



Building Telegram Bots

Develop Bots in 12 Programming
Languages using the Telegram
Bot API

—

Nicolas Modrzyk

Apress®

www.EBooksWorld.ir

Building Telegram Bots

Develop Bots in
12 Programming Languages
using the Telegram Bot API

Nicolas Modrzyk

Apress®

Building Telegram Bots: Develop Bots in 12 Programming Languages using the Telegram Bot API

Nicolas Modrzyk
Tokyo, Tokyo, Japan

ISBN-13 (pbk): 978-1-4842-4196-7
<https://doi.org/10.1007/978-1-4842-4197-4>

ISBN-13 (electronic): 978-1-4842-4197-4

Library of Congress Control Number: 2018965498

Copyright © 2019 by Nicolas Modrzyk

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image, we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the author nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Nikhil Karkal
Development Editor: Matthew Moodie
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science+Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4196-7. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author	ix
About the Technical Reviewers	xi
Acknowledgments	xiii
Introduction	xv
Chapter 1: Week 1: Ruby	1
Chatting with the BotFather	1
Setting Up Ruby	7
Your First Telegram Bot	8
Understanding Received Messages Fields	14
First Reply	15
Chapter 2: Week 2: Nim	17
Installing Nim	17
Nim Plug-in for Visual Studio Code	18
Hello, Nim	20
Second Nim Program (Still Not Bot)	23
Creating Visual Studio Code Build Tasks	25
Installing Nim Packages with Nimble	29
First Nim Bot	31
Replying to Nim Bot	33
Cats and Dogs Nim Bot	34

TABLE OF CONTENTS

Chapter 3: Week 3: Crystal	37
Setting Up Crystal	38
Short Walk in the Playground.....	39
Going Visual Studio Code Again	41
Creating a Crystal Project	46
Echo Bot.....	50
Command Bot	52
Chapter 4: Week 4: Rust	57
Rust Installation and First Steps	58
Installation.....	58
First Rust or Two	62
Hello Rust	62
Fibonacci	64
Ride the cargo.....	66
We Have Time	66
Multiple Cargo Targets	70
Rust Bot Number 1: Reply to Me	74
Rust Bot Number 2: Where Is Tokyo?	77
Rust Bot Number 3: Chained Reaction	82
Compiling for Release	84
Chapter 5: Week 5: D	85
Installation and First D Steps	86
Some Bits of D on Concurrency	91
Simple Threading.....	91
Thread with a State	92
Shared State.....	94

A Few More Examples of D	95
Sort Me Tender, Sort Me True.....	95
My Love for Fibonacci	98
Telegram Bots in D	100
Meet dub.....	100
First D Bot.....	105
More Bot API Usage	107
Chapter 6: Week 6: C++	111
Requirements, Installation, and First Bot	111
Install tgbot-cpp	112
Install OpenCV	113
File Download Program	113
Echo Bot.....	119
C++ Bots.....	123
Bot with Inline Keyboard	123
Photo Bot	126
OpenCV in action.....	128
OpenCV Sample Program	128
OpenCV Bot	131
Chapter 7: Week 7: Clojure	135
Initial Setup and First Clojure Bot	136
Visual Studio Code.....	141
The Project Metadata in project.clj.....	142
The Clojure Code in core.clj.....	143
The Token!	146
Debugging Telegram Messages.....	146

TABLE OF CONTENTS

Creating a Reverse Bot.....	148
Inline Handler	149
A Simple Weather Bot	150
OpenCV and Telegram: Origami Bot	154
Chapter 8: Week 8: Java	159
Installation	159
The Project Structure.....	161
The build.gradle file.....	161
Visual Studio Code Setup	164
First Java Bot	165
Send Some Text.....	167
Send a Photo	169
Bot with Invoice Capabilities.....	170
Asking Permission	170
Sending an Invoice Message	173
Chapter 9: Week 9: Go	181
Installation of Go	182
Let's Go	184
Let's Fib.....	190
First Bot in Go	193
Just Sending Pictures	197
Chapter 10: Week 10: Elixir	201
Installation	202
Using iex.....	203
Using mix.....	204
Running iex with mix	205

Project Structure of a mix Project.....	206
config.exs	207
mix.exs	208
Dependencies	209
telegrambox.ex.....	212
(Back to) Dependencies.....	213
Get Something	216
GetMe	216
GetChat.....	217
GetFile	218
Using Elixir's System	219
SendPhoto	220
Telegram Bot.....	221
Bot1: Anything Goes.....	221
Bot2: Fibonacci	222
Chapter 11: Week 11: Node.js.....	225
Meet RunKit	226
Creating an Account	226
First Code on RunKit.....	229
A Certain Je Ne Sais Koa	230
Publishing Some Koa.....	232
Telegram Bot with Webhooks.....	234
More on the Telegraf Library	239
Image-to-Chat Example.....	239
RegExp, Inline Keyboards, and Embedded Emojis.....	240

TABLE OF CONTENTS

Running Node.js Locally.....242

 Setting Up Node.js243

 Using Local Tunnel.....244

Chapter 12: Week 12: Python247

 Installation248

 A Few Python Programs.....250

 Fibonacci 1250

 Fibonacci 2252

 Fibonacci 3253

 Fibonacci 4254

 Fibonacci 5255

 First Telegram with Python256

 First Bot: Send a Random Photo258

 First OpenCV Bot: Changing the Color Space of a Picture.....260

 Second OpenCV Bot: Count Faces.....262

 TensorFlow to Close the Show265

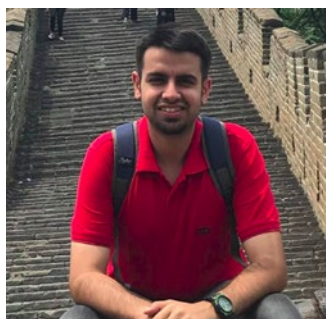
Index.....271

About the Author



Nicolas Modrzyk has more than 15 years of IT experience in Asia, Europe, and the United States and is currently CTO of an international consulting company in Tokyo, Japan. He is the author of four other published books, mostly focused on the Clojure language and expressive code. When not bringing new ideas to customers, he spends time with his two fantastic daughters, Mei and Manon, and playing live music internationally.

About the Technical Reviewers



Dushyant Rathore is currently working as a firmware engineer with Western Digital. His experience includes full-stack web development, machine learning, decentralized applications, and others. Dushyant has worked on several kinds of projects related to IoT, chatbots, web sites, scrapers, command-line tools, and machine learning projects, among others, at various startups. He has participated in national and international hackathons and has won a few of them. He is a big cloud computing enthusiast.



Sham Satyaprasad has been a full-stack software developer for more than four years, having completed a master's degree in embedded systems from Manipal University. He prides himself on writing highly efficient, readable, and maintainable code and strongly believes that coding is an art as much as it is science. Sham has recently developed a keen interest in NLP, ML, and data science and has been busy wrapping his head around these topics.

Acknowledgments

It's been yet another typing race to get this book out on time and beat the odds. Divya and Nikhil, thank you so much for teaming up again. I'm really looking forward to our next collaboration!

I received support from so many people that it would take another book just to write the list of names. Nevertheless...

Thank you to my sister, Emily, my brother, Gregor, Mum and Dad, family, cousins, uncles, aunts, friends, Abe-san, Sawada-san, Gucci, Marshall, Momo, my soulmate Sachiko, soccer friends (I would break a knee for you), the Irish crew still enjoying Guinness (one more for me!), the awesome people in America (who always find the LPs I'm missing), Chris and the Biners, the French team that's always there to support me, even without being asked, and the people deep in my heart, for your *never-ending* support. I could not have finished this without you. I appreciate you all so much.

And, of course, thank you to my two wonderful daughters, Mei and Manon, for keeping up and always doing your best, even during hard times. You rock! I love you.

Introduction

With a hundred ways to do a dozen things, why not try it all?

—Julian Casablancas

Have you ever wondered how you could accomplish more by doing less, how you could have a sort of double who does all the work while you enjoy some cool beachside or spend more outdoor time with your beautiful children? I always have.

I am a big fan of the Telegram chat platform. Let's call it a platform, because it is more than a simple chat service with which you can stay in touch with people who matter to you most. It also enables you to think in ways you haven't before.

For example, living in Tokyo, you always care about what time the last train home is going to depart. I guess most people in big cities around the globe probably have that same concern. Checking the clock only every so often can result in a terrible and/or expensive taxi ride, so I started wanting something that automatically offered me a bunch of options to get home.

The first bot I wrote was to tell me the schedule of the last few trains home and some different options, from the easiest to reach before the last few departures to the very latest, which I would have to dash to catch. That saved me quite a bit of money.

The second bot I wrote was slightly more IoT-oriented. It used a webcam to send me via Telegram pictures of people who rang my doorbell.

The third one, I also remember, was kind of stupid. It was to use a mini projector to display the most recent message coming through a Telegram chat room. (It's very entertaining to view random messages during a small party at home.)

INTRODUCTION

But there are so many things for which to try to build a bot—search for a plane ticket, check your fridge, etc. Having a bot is a simple way to facilitate all the things you do daily, using the same kind of simple Telegram chat rooms to get answers to questions related to daily life or to issue commands and conquer the world.

This relatively short book is about learning how to write Telegram bots in several different programming languages. Why not use one and stick to it? you might ask. Well, because there's not one answer to all questions, and what's right for others might not be suitable for you. Exploring different programming languages is also a fun way to examine the strengths of each language while performing the same tasks. Each of the Telegram concepts can be introduced one after the other, in a simple fashion.

Or, you could just jump in and choose the language you want to try and get started in no time. Some people want it to happen; some wish it would happen; others make it happen. So, enjoy reading this book, and make it happen.

CHAPTER 1

Week 1: Ruby

Mindful Monday Humans, may your coffee kick in before reality does.

—Napz Cherub Pellazo

Ruby took the world by storm a while ago, owing to the ease and concision of the code you can write with it. Most programmers have a sweet spot for Ruby, and when their shell scripts and day-to-day lives get too messy, they are usually very quick to switch to Ruby.

This first chapter is a bit special, because on top of creating a client for our bot, I must also introduce you to how to create the bot itself. Throughout the book, this first bot will be reused at will, although, of course, the same steps used to make it could be used to create a bot army and conquer the world!

Chatting with the BotFather

To register your own Telegram bot, you must talk to the father of all bots. This bot father has a name, BotFather; Brad or Vladimir just doesn't cut it. He can be reached via Telegram as @BotFather.

BotFather does not sleep and can be reached at any time of day. BotFather does take showers and always looks fresh. Here is the last profile picture we have of this handsome bot (Figure 1-1).



Figure 1-1. BotFather’s latest profile picture

Finding BotFather is not so difficult; you just have to type his name, “@BotFather,” in the Telegram list of people in the search box of your Telegram client (Figure 1-2).

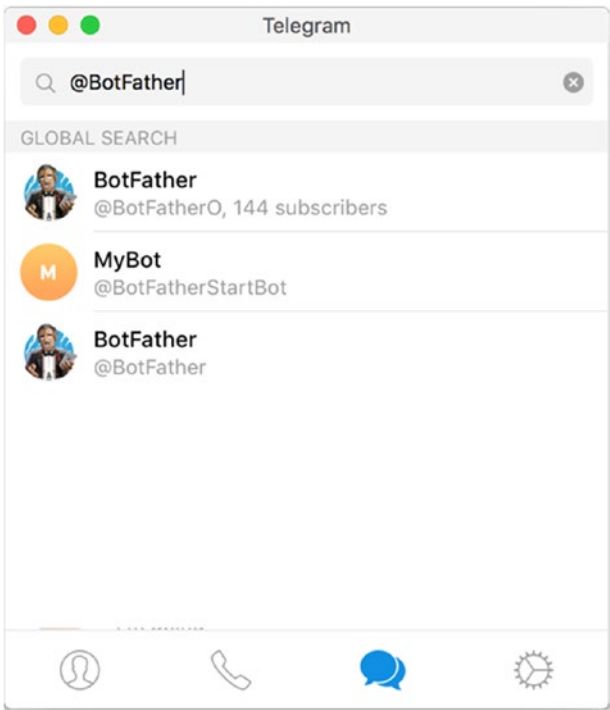


Figure 1-2. Looking for BotFather

In the preceding list, the name is the one at the bottom. Next, start a new chat with BotFather by clicking it.

Once the chat is started, you will also be welcomed by our handsome bot, with a cordial message about APIs, free help, and an invitation to start the chat (Figure 1-3).

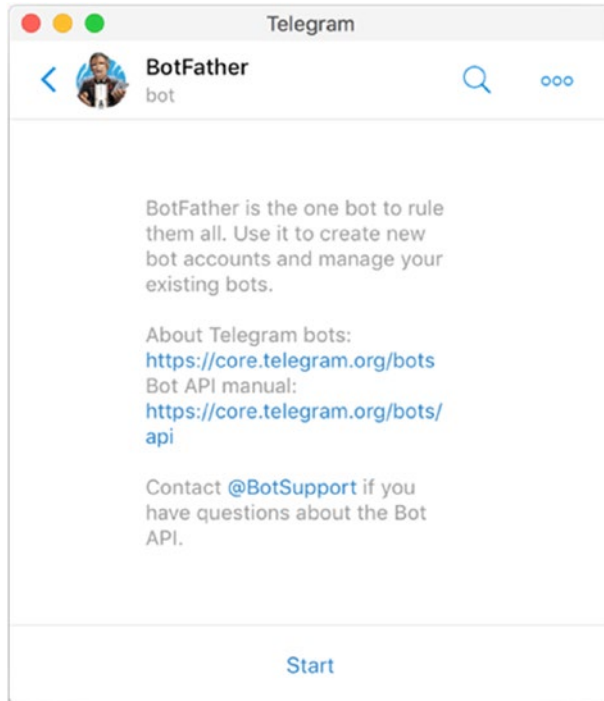


Figure 1-3. Ready?

Once the chat has begun (by pressing the Start button that you can see at the bottom), you are welcomed by BotFather with a bunch of options on how to create or edit your list of bots (Figure 1-4).

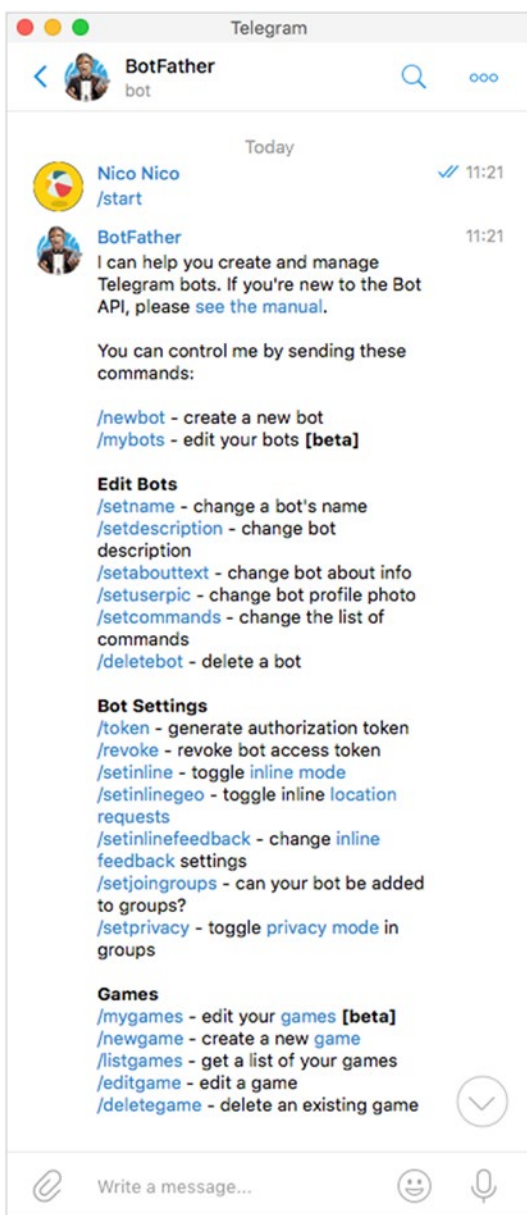


Figure 1-4. Say hello to BotFather

Great! I won't review the full list of options now but will start just by creating our new bot. This is done here by typing in the `/newbot` command and then following a simple conversation, such as the one in Figure 1-5.

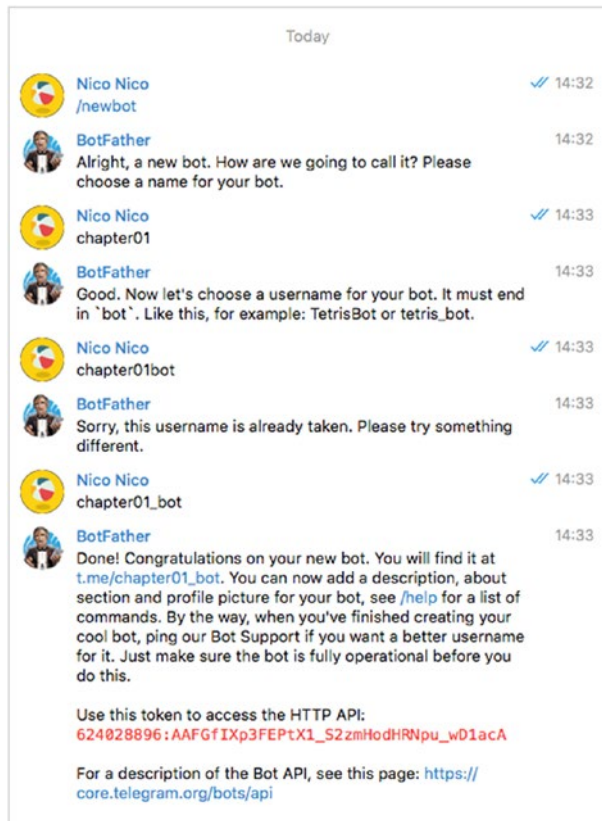


Figure 1-5. Ask BotFather, please, please, for a new bot

Your bot is now ready to use. Can you see in red something like a secret code? This is the bot token, which is a chain of characters that will be used to uniquely identify and authenticate your bot against the Telegram platform. **Do not give away this token.** Don't write it in a book or allow it to hang somewhere on GitHub, especially now that Microsoft owns it.

In our case, in the preceding chat, the token that was generated and given to use is the one following:

624028896:AAFGfIXp3FEptX1_S2zmHodHRNpu_wD1acA

If your token, like this one, ever becomes compromised, you can use the `/token` command with the bot father, to generate a new token, as shown in the conversation in Figure 1-6.

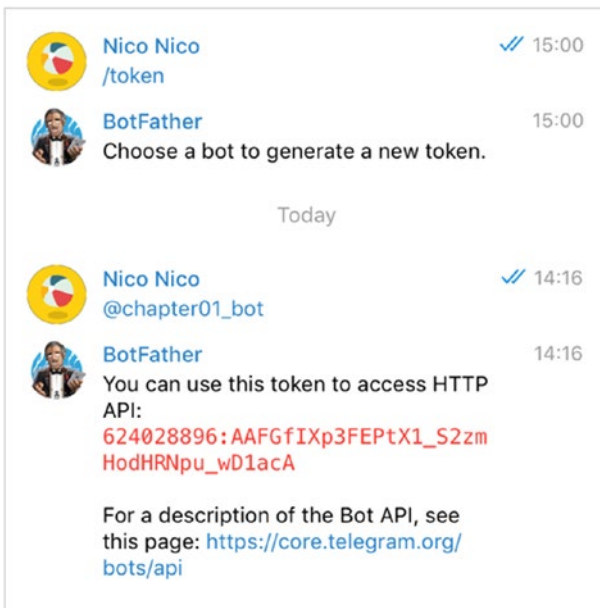


Figure 1-6. Chat to generate a new token

Alright, the registration of our Telegram bot is all done. So, let's switch to a little bit of coding in Ruby.

Setting Up Ruby

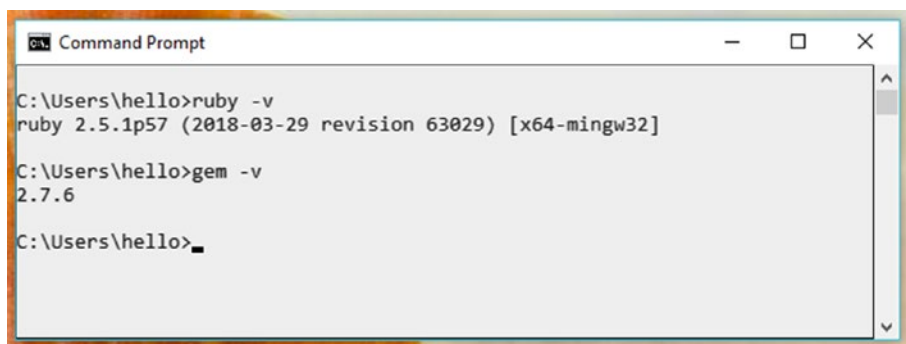
Ruby, on most Unix-like platforms, including OS X, is already installed, or it can be installed using a package manager. For those running lesser operating systems, like Windows, you can download and install the Ruby installer (Figure 1-7) from the Ruby download page at www.ruby-lang.org/en/downloads/. Download the most recent version.



Figure 1-7. Looking for the Windows Ruby installer

After the installation is complete, if you open a terminal (on macOS), or a command prompt on Windows (Figure 1-8), and can type in the following commands without getting an error, you are all set:

```
ruby -v
gem -v
```



```
C:\Users\hello>ruby -v
ruby 2.5.1p57 (2018-03-29 revision 63029) [x64-mingw32]

C:\Users\hello>gem -v
2.7.6

C:\Users\hello>
```

Figure 1-8. *Checking ruby and gem versions*

If you have never used Ruby before, you may be wondering what the gem executable is? It's simply a Ruby-specific installer for libraries (just like npm is for Node and pip is for Python), so when you require some open source library that has been written by someone else, you would use gem to get it on your machine and the ruby executable to run it.

Apart from Ruby, to have some coding fun outside Notepad, you also need a text editor, so I propose to use Microsoft's Visual Studio Code (<https://code.visualstudio.com/>), but, of course, any of your favorite text editors will do.

Alright, let's get started and code our first bot.

Your First Telegram Bot

To get to talk to our first bot, we will use the Ruby library named telegram_bot. There are a few other famous libraries that you can find on RubyGems (<https://rubygems.org/>), but I find this library to be an easy one to start and get going with, and I hope you come to agree with me about this in time.

Actually, you can check for yourself and find your favorite Telegram library, by querying the RubyGems web site (Figure 1-9).

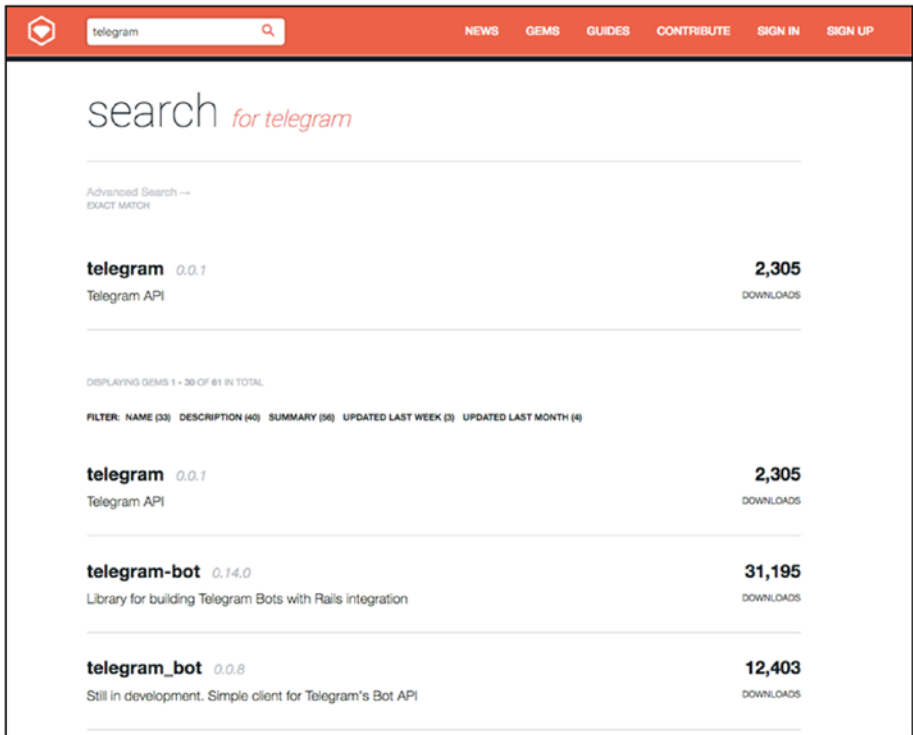


Figure 1-9. Looking for gems

To use a library in Ruby, you install it on your machine first, to make it available to your computer, by using the `gem install` command, and then in your Ruby code, you use the `*require*` function, to make that library available to your Ruby program.

Let's create a new folder for this first bot. Change the directory and then install the Telegram library with `gem`, as shown following:

```
mkdir chapter-01
cd chapter-01
gem install telegram_bot
```

At the terminal, the output should be something similar to this:

```
SuperPinkicious:chapter-01 niko$ gem install telegram_bot
Successfully installed telegram_bot-0.0.8
Parsing documentation for telegram_bot-0.0.8
Done installing documentation for telegram_bot after 0 seconds
1 gem installed
```

The gem is now installed and ready to be used in your code.

Now, you are going to write some code to wake up your bot and make it come alive. In a new file in that folder, which you can name `step0.rb`, for example, let's write the following lines of Ruby code:

```
require 'telegram_bot'

bot = TelegramBot.new(token: ENV['BOT_TOKEN'])
bot.get_updates() do |message|
  puts message.to_s
end
```

What that code does is

- Make the `telegram_bot` library, installed via gem, available to your program
- Create a new Ruby bot object, using the Telegram token exposed via an external variable. This is usually the recommended way to share your bot code without giving your bot token to everyone.
- Get the bot instance instantiated, to listen for incoming messages, using the bot object `get_updates()` method
- Ensure that, now, whenever a message is sent to the bot, the bot will print it on the console

To run the preceding written program on your machine, you pass the name of the program file, `step0.rb`, to the ruby executable. Let's do it.

At the terminal, execute the following command:

```
ruby step0.rb
```

Observe the output (Figure 1-10).



```
SuperPinkicious:chapter-01 niko$ ruby step0.rb
Traceback (most recent call last):
  8: from step0.rb:5:in `<main>'
  7: from /Users/niko/projects/homebrew/lib/ruby/gems/2.5.0/gems/telegram_bot-0.0.8/lib/telegram_bot/bot.rb:32:in `get_updates'
  6: from /Users/niko/projects/homebrew/lib/ruby/gems/2.5.0/gems/telegram_bot-0.0.8/lib/telegram_bot/bot.rb:32:in `loop'
  5: from /Users/niko/projects/homebrew/lib/ruby/gems/2.5.0/gems/telegram_bot-0.0.8/lib/telegram_bot/bot.rb:33:in `block in get_updates'
  4: from /Users/niko/projects/homebrew/lib/ruby/gems/2.5.0/gems/telegram_bot-0.0.8/lib/telegram_bot/bot.rb:83:in `get_last_messages'
  3: from /Users/niko/projects/homebrew/lib/ruby/gems/2.5.0/gems/telegram_bot-0.0.8/lib/telegram_bot/bot.rb:72:in `get_last_updates'
  2: from /Users/niko/projects/homebrew/lib/ruby/gems/2.5.0/gems/telegram_bot-0.0.8/lib/telegram_bot/bot.rb:68:in `request'
  1: from /Users/niko/projects/homebrew/lib/ruby/gems/2.5.0/gems/telegram_bot-0.0.8/lib/telegram_bot/bot.rb:68:in `new'
/Users/niko/projects/homebrew/lib/ruby/gems/2.5.0/gems/telegram_bot-0.0.8/lib/telegram_bot/api_response.rb:13:in `initialize': uninitialized constant TelegramBot::ApiResponse:ResponseError (NameError)
SuperPinkicious:chapter-01 niko$
```

Figure 1-10. Sometimes, it just does not work

Oops! That did not go so well. We forgot to pass the Telegram token to our program.

This can be done on Linux or OS X with the following:

```
export BOT_TOKEN='585672177:AAHswpmdA2zP52ZWojMdteGa0xQ8KeynWvE'
```

And on Windows with

```
set BOT_TOKEN=585672177:AAHswpmdA2zP52ZWojMdteGa0xQ8KeynWvE
```

Let's run the program again. This time, it looks like the command is not finishing... This is expected, as the bot is now actually waiting for messages.

Let's be the one to start the conversation, so let's send a greeting message.

In the Telegram window, search for the bot and start chatting (Figure 1-11).

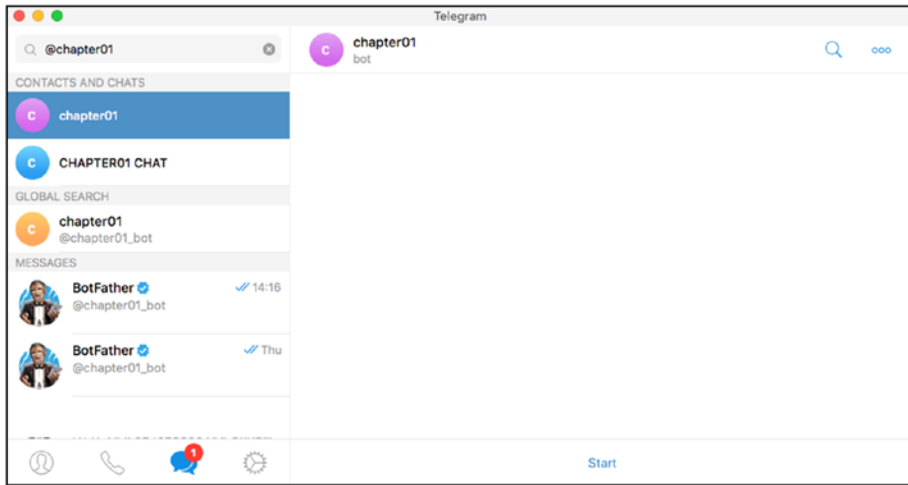


Figure 1-11. First message to our bot

By pressing that Start button, the bot is already receiving a message (Figure 1-12)!

```
C:\Users\hello\projects\chapter01>ruby step0.rb
#<TelegramBot::Message:0x0000000003713d58>
```

Figure 1-12. First message from our bot!

Hmmm, that was not very readable. With Ruby, you can display a more legible version of any object, by using the `to_yaml` function. Let's update the code and see what happens.

We call `require 'yaml'`, to import it into the Ruby namespace, and now we can call the `to_yaml` on the message the bot has received.

```
require 'telegram_bot'
require 'yaml'
bot = TelegramBot.new(token: ENV['BOT_TOKEN'])
```

```

bot.get_updates() do |message|
  # puts message.to_s
  puts message.to_yaml
end

```

You would have to type Ctrl-C to terminate the running version of the bot first and then start the new bot, by executing the ruby command again. Figure 1-13 shows the outcome.

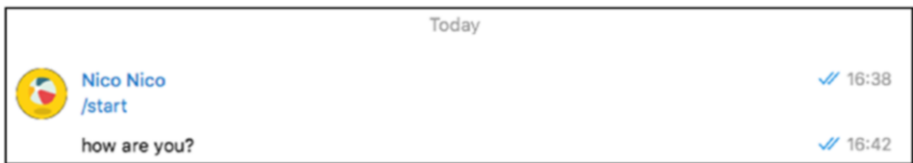


Figure 1-13. *How are you?*

In the console, or in the terminal where the bot was started, you now can see a more detailed version of the received message.

```

endSuperPinkicious:chapter-01 niko$ ruby step0.rb
--- !ruby/object:TelegramBot::Message
message_id: 191
from: !ruby/object:TelegramBot::User
  id: 121843071
  first_name: Nico
  last_name: Nico
  username: hellonico
chat: !ruby/object:TelegramBot::Channel
  id: 121843071
  username: hellonico
  title:
date: !ruby/object:DateTime 2018-08-31 07:42:26.000000000 Z
text: how are you?
reply_to_message:

```

Actually, whatever the programming language used, this message format is going to be quite consistent, so it's a good idea to have a look at the message fields.

Understanding Received Messages Fields

Table 1-1 briefly explains the fields we have just received via the Telegram bot.

Table 1-1. *Fields Received from the Telegram Bot*

Field	Sample Value	Explanation
message_id	191	The unique identifier of the bot message
from	!ruby/object:TelegramBot::User	The user who sent the message
chat	!ruby/object:TelegramBot::Channel	The chat/channel information
date	!ruby/object:DateTime 2018-08-31 07:42:26.000000000 Z	When the message was sent
text	how are you?	The text of the message
reply_to_message	<empty>	The message that this message was a reply to

For standard messages, there is not much beyond what you would expect from a chat message object. You will probably use the date, text, and from fields most of the time.

First Reply

Again, Ruby's concision makes it quite easy to create strings from objects. With the following, you can use blocks of executable code directly within a string: `#{}.`

This makes it very powerful for templated messages, and in our case, for bot programming. Within the `get_updates()` call block, let's now write and send a reply.

```
message.reply do |reply|
  reply.text = "Hello, #{message.from.first_name}!"
  reply.send_with(bot)
end
```

After restarting the bot (Ctrl-C, `ruby step0.rb`), we can start a more talkative version of this Ruby bot (Figure 1-14).

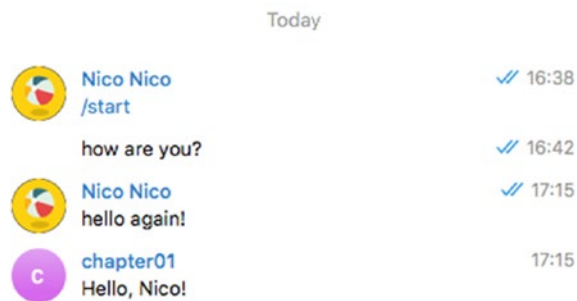


Figure 1-14. *Hello, Nico!*

That worked pretty smoothly, and I am happy to announce that you have moved to level 2 of this bot master course.

You have seen many things in this first chapter, the following among them:

- How to register a bot to the bot father
- How to put the bot token to use in our program
- Starting a bot and listening to messages
- Reviewing the fields of incoming messages
- Replying to an incoming message

Next week, we will build on those same concepts in the following chapter using a different programming language.

CHAPTER 2

Week 2: Nim

Jenner: I learned this much: take what you can, when you can.

Justin: Then you have learned nothing.

—The Secret of Nimh

Nim has been on my list of adopted languages for the last two years. Visually, it looks and feels very similar to Python, but it can also be compiled to a binary, via C, C++, Objective-C, and even JavaScript. I have seen compiled binary Nim programs running *very* fast, and I really wished Nim would reach the next level of adoption.

Today, we're going to focus on building bots, runnable as a binary, built with Nim. We will, of course, start by setting up a Nim environment on your local machine.

Installing Nim

Nim can be downloaded manually from the following location: <https://nim-lang.org/docs/re.html>. It is also available through your platform package manager.

Make sure you install both Nim (the compiler) and Nimble (the package manager). At the time of writing, these are the latest available versions for both executables.

```
SuperPinkicious:APRESS niko$ nim -v
Nim Compiler Version 0.18.0 [MacOSX: amd64]
Copyright (c) 2006-2018 by Andreas Rumpf
```

```
active boot switches: -d:release -d:useLinenoise
SuperPinkicious:APRESS niko$ nimble -v
nimble v0.8.10 compiled at 2018-03-11 16:16:29
git hash: couldn't determine git hash
```

The setup used throughout this book, and thus for Nim programming too, is based on Visual Studio Code, and you will be shown the basics to get running with Nim from within Visual Studio Code.

Nim Plug-in for Visual Studio Code

Before writing a full-blown Telegram bot, you are going to be introduced to Nim, by writing some Hello World code. Let's start by creating a file named `goodmorning.nim` from within Visual Studio Code, to write our first few lines of code (Figure 2-1).

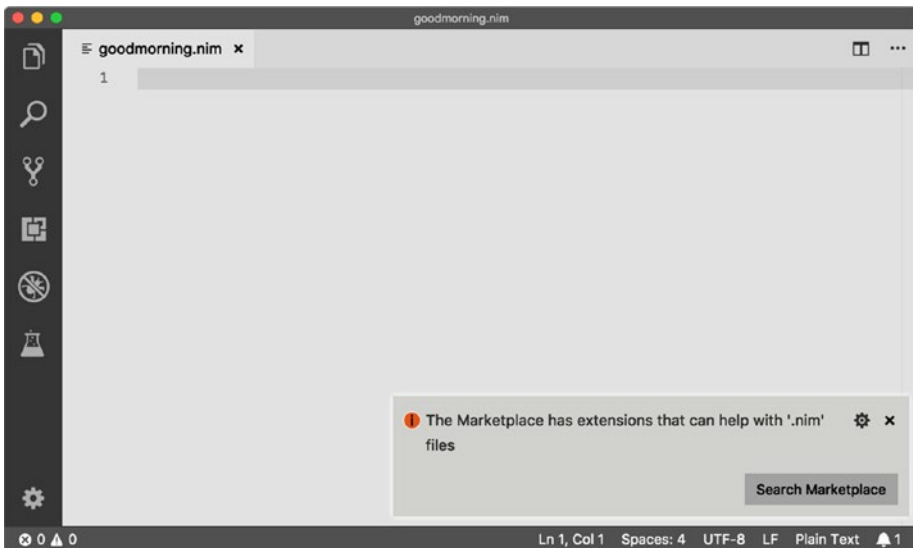


Figure 2-1. goodmorning.nim file

As you can see at the bottom right of Figure 2-1, Visual Studio Code recognizes the extension of the new file and proposes that you have a look at the Marketplace, to find an extension that supports Nim files editing. This is the same for every other language, so you may have to repeat these steps for other languages afterward.

The plug-in to install is the one written by Konstantin Zaitsev, shown in Figure 2-2.

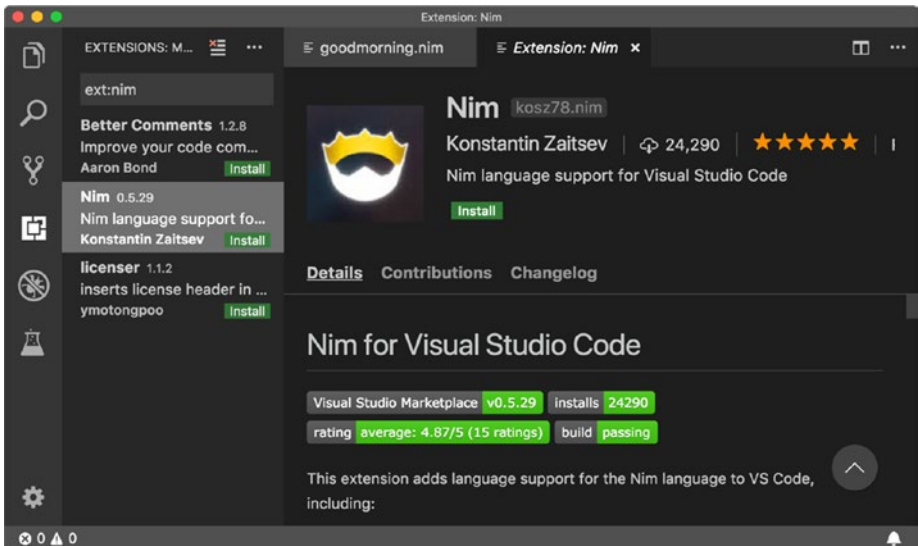


Figure 2-2. *Nim plug-in*

Once Visual Studio Code has installed the plug-in, you must click the Reload button, to activate the plug-in (Figure 2-3).

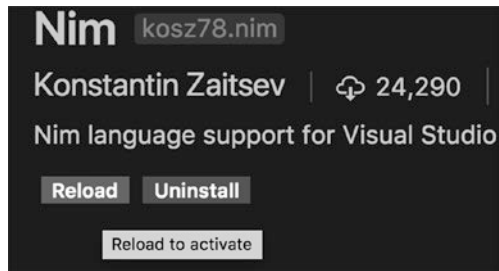


Figure 2-3. *Activating the Nim plug-in*

Now, let's write some code!

Hello, Nim

If the install was successful, you now have a Nim-enabled editor, meaning syntax highlighting and compilation from within the editor. `echo` is the simplest Nim function to print to the standard output, and we'll use that to print some formal greetings (Figure 2-4).



Figure 2-4. *hello nim world greeting*

Once you have the code written in the file, you are now ready to execute it.

Unlike Ruby, in which you can pass in code to be executed to the ruby executable, Nim is a compiled language, so it is turned into a binary file first. This is a form that the computer understands and can execute by itself, including third-party libraries, then that file is executed.

To active the command that performs these two steps in Visual Studio Code, you use either one of the following:

1. Command+Shift+P or
2. Ctrl+Shift+P

This will pop up a menu with all the available Visual Studio Code commands. The one you want to use to compile and run Nim code located in the opened file is named `Nim: Run Selected file`, and it will be the first option, once you start typing “nim” in the search-like text box that popped up (Figure 2-5).

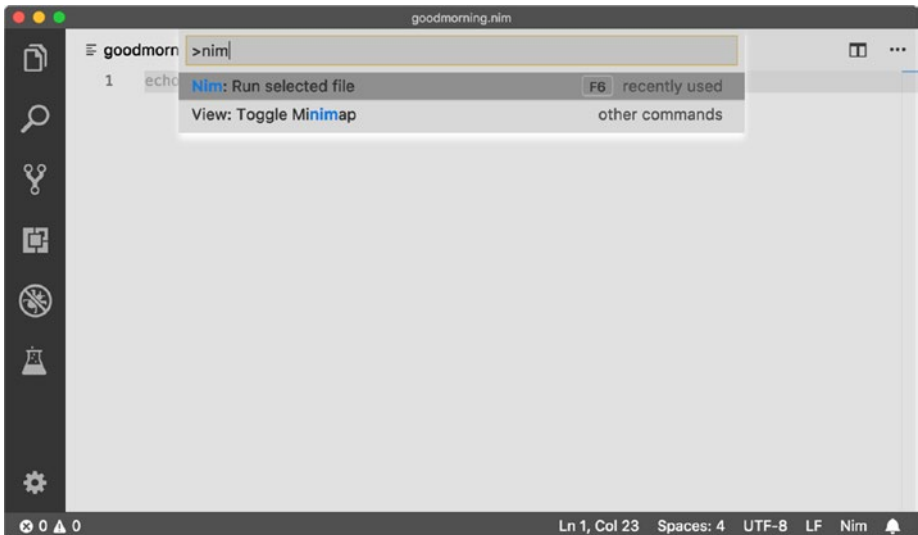


Figure 2-5. Run selected file from the pop-up menu

Executing the command will print out a few lines of compilation in the terminal or command prompt that has just been opened when triggering the command. At the bottom, you can see the greeting message. The code has been compiled and executed from within the text editor (Figure 2-6).

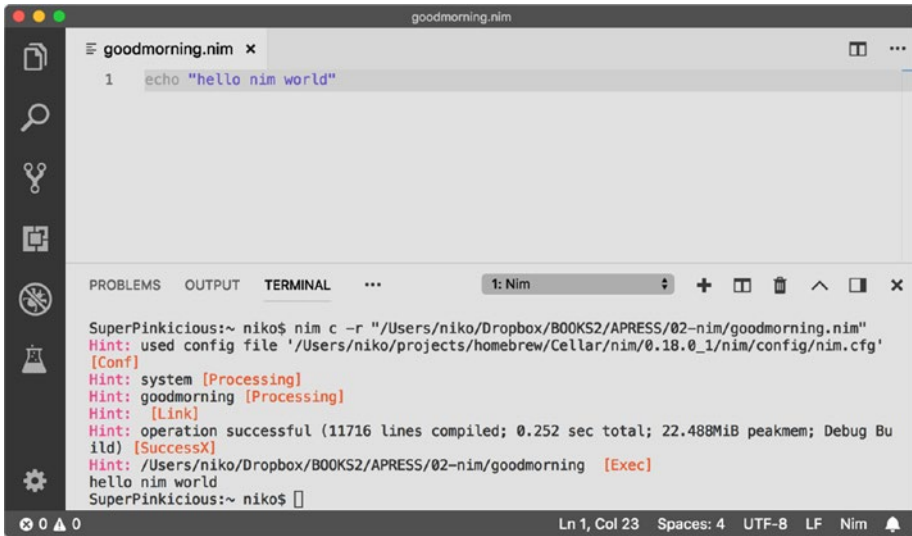


Figure 2-6. *hello nim world*

Note that the actual command that was executed by the plug-in is something like

```
nim c -r goodmorning.nim
```

This is a shortcut for

- C: compile
- -r: run
- goodmorning.nim: the file to compile and run

In the folder in which the `goodmorning.nim` file was located, you will see that a new file has been created, named `goodmorning`, without a file extension.

You can re-execute the compiled file by itself, of course, without recompiling directly from the command line again.

```
SuperPinkicious:02-nim niko$ ./goodmorning
hello nim world
```

Try changing the code a bit and recompile a few times, to master this compile step from Visual Studio Code. Next let's move beyond greetings, to more meaningful code.

Second Nim Program (Still Not Bot)

In this second example, we will send an HTTP request to a remote site and print the result on the command line. This new program requires some code from a different Nim module, `httpclient`, which contains the object that can send HTTP requests and reads the return value. You will use the keyword `import` to make this happen.

The rest of the code, matching closely a Python-like syntax, is quite easy to read.

```
import httpclient

const URL = "https://api.github.com/search/repositories" &
            "?q=cat" &
            "&sort=stars" &
            "&order=desc"

let
  client = newHttpClient()

echo client.getContent(URL)
```

We see first the creation of a constant String, URL, which will be the target of the HTTP request. `let` defines a scoped variable, the HTTP client. Finally, we retrieve the body coming from the HTTP get request, with `getContent` called on the HTTP client.

Let's run this now, by calling the `Nim: Run selected file` command from the palette (Figure 2-7).



Figure 2-7. *Running the HTTP client*

Unfortunately, things did not seem to fare too well, and a message with red text is showing, as in Figure 2-8.

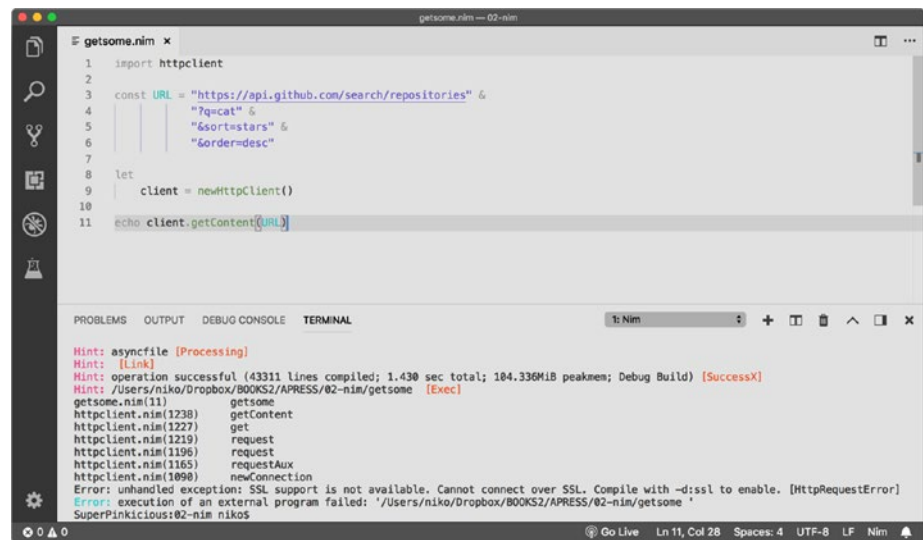


Figure 2-8. *Failed execution*

Fortunately, it tells you explicitly what's wrong, which is quite unusual for a programming language. Go ahead and ask Java developers.

What happened is that the HTTP connection requires SSL, as we are connecting over HTTPS. SSL is not included by default during the compilation step of Nim source files.

To include SSL support, we must define a symbol, using the `-d:ssl` flag, for the `nim` command, so that it will include the required code to execute the HTTPS request.

If you want to try this on the command line or in a terminal, you can use the following full command:

```
nim c -d:ssl -r getsome.nim
```

The preceding command will compile and run the code, as expected, finally outputting many lines of JSON returned from the get request. But...I know how you feel. You would like to run this command from Visual Studio Code itself. This is done using Visual Studio Code Custom tasks.

Creating Visual Studio Code Build Tasks

To create your own task in Visual Studio Code, follow a few steps from the command palette, which will create a JSON file from a template (Figures 2-9, 2-10, 2-11).

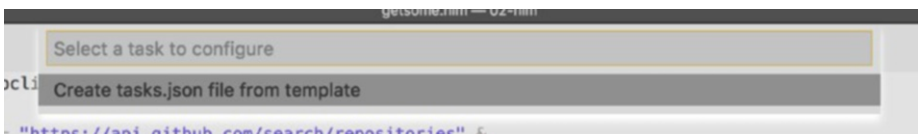


Figure 2-9. *Creating the tasks.json file from the template*



Figure 2-10. *Configuring the default build task*

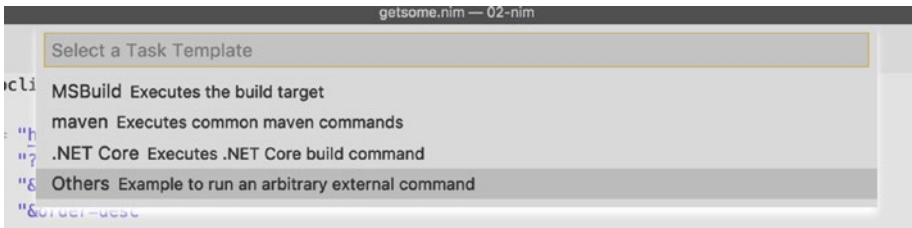


Figure 2-11. *Running an arbitrary external command*

At this end of the sequence, you will have a new file, `.vscode/tasks.json`, with something similar to the following content:

```
{
  // See https://go.microsoft.com/fwlink/?LinkId=733558
  // for the documentation about the tasks.json format
  "version": "2.0.0",
  "tasks": [
    {
      "label": "echo",
      "type": "shell",
      "command": "echo Hello",
      "problemMatcher": []
    }
  ]
}
```

Basically, this task executes a shell command that displays “hello” on the terminal or command prompt. Not a very useful command by itself, in general, but, hey, who knows.

Now, we want to use that to the default build command, so let’s trigger the palette again with `Command+Shift+B`, `Ctrl+Shift+B`.

This will ask you a few more questions along the lines of those shown in Figures 2-12 and 2-13.

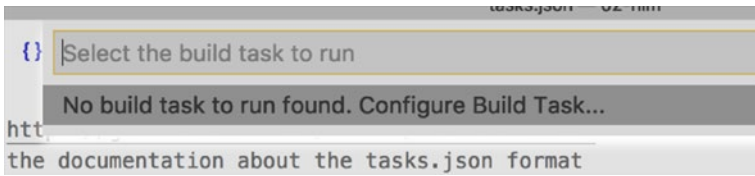


Figure 2-12. *Configuring the build task*

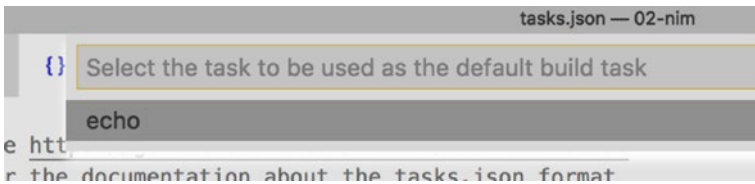


Figure 2-13. *echo*

The custom build task hello is now configured within Visual Studio Code. The next time you type “Command+Shift+B,” a terminal will show up with the output of the echo command (Figure 2-14).



Figure 2-14. *Hello*

“Hello” is usually good enough for a first time, but, yes, now it’s probably the time to get back to compiling and executing Nim files.

So, let's replace the content of the `.vscode/tasks.json` file with the following:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "nim",
      "command": "nim",
      "args": [
        "c",
        "-d:ssl",
        "-r",
        "${file}"
      ],
      "options": {
        "cwd": "${workspaceRoot}"
      },
      "group": {
        "kind": "build",
        "isDefault": true
      }
    }
  ]
}
```

Note that you have just created a task labeled `nim`, for which the command is the one you saw just a few pages ago.

```
nim c -d:ssl -r ${file}
```

`${file}` in Visual Studio Code is a parameter that contains the currently opened file. Now, we can execute the `getsome.nim` directly from the editor (Figure 2-15).

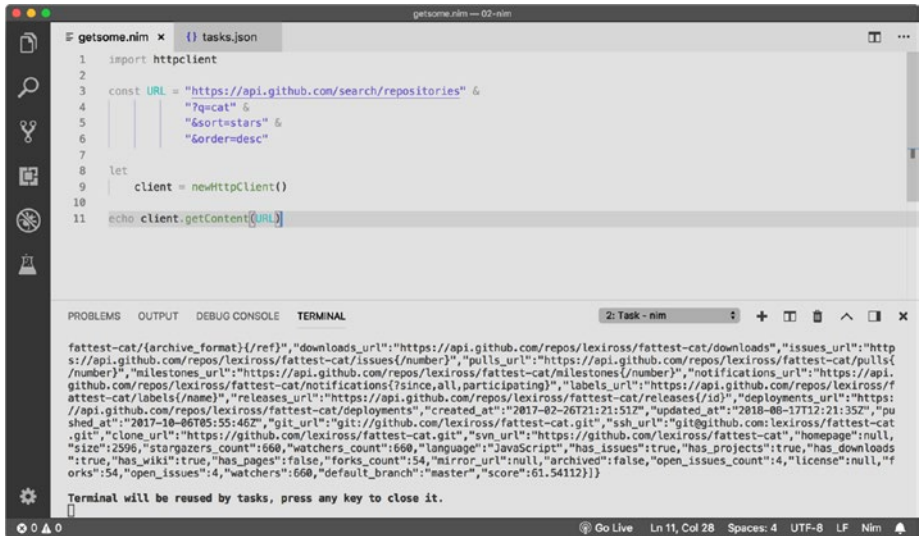


Figure 2-15. Executing `getsome` directly from Visual Studio Code

Great! This custom task was actually required to compile and run code that includes the third-party Telegram library we are going to use. But how do we install external packages again?

This is done via Nimble. Let's see how that works.

Installing Nim Packages with Nimble

Remember that you installed Nimble at the beginning of this chapter. You are going to use it now to install a wrapper for the Telegram API.

You can use Nimble to find packages, using the search subcommand, as shown following:

```
nimble search telegram
```

This returns a list of libraries matching the search keywords, plus some information related to each library.

```
SuperPinkicious:02-nim niko$ nimble search telegram
```

```
telebot:
```

```
url:          https://github.com/ba0f3/telebot.nim (git)
tags:         telebot, telegram, bot, api, client, async
description:  Async Telegram Bot API Client
license:      MIT
website:      https://github.com/ba0f3/telebot.nim
```

```
nim_telegram_bot:
```

```
url:          https://github.com/juancarlospaco/nim-
              telegram-bot (git)
tags:         telegram, bot, telebot, async, multipurpose, chat
description:  Generic Configurable Telegram Bot for Nim, with
              builtin basic functionality and Plugins
license:      MIT
website:      https://github.com/juancarlospaco/nim-telegram-bot
```

To build our bot, we will use the first, most widely used library, telebot, and this is done using the Nimble install command.

```
nimble install telebot
```

This results in the following output:

```
SuperPinkicious:02-nim niko$ nimble install telebot
Downloading https://github.com/ba0f3/telebot.nim using git
Verifying dependencies for telebot@0.4.2
Installing telebot@0.4.2
Success: telebot installed successfully.
```

You can, of course, check that the package is installed on your machine, with

```
nimble list -i
```


or remove the package with `uninstall`:

```
nimble uninstall telebot
```

Now that everything you require is on your machine, let's finally start writing our bot in Nim!

First Nim Bot

In Nim, you define a function or procedure using `proc`. `proc` takes a function name, here, `updateHandler`; some parameters, here, `b` of type `telebot` and `u` of type `Update` (both from the `telebot` module); and sometimes some metadata (or `pragma`), here, `{.async.}`.

You can find the documentation for any module on nim-lang.org. For example, the `asyncdispatch` module is here:

<https://nim-lang.org/docs/asyncdispatch.html>

The reason we use the `async` module here is to write our Telegram bot.

Note Asynchronous procedures remove the pain of working with callbacks. They do this by allowing you to write asynchronous code the same way as you would synchronous code.

The `slurp` function takes the content of a file and turns it into a string—here, the `bot API_KEY` that you receive from BotFather in the previous chapter. And, yes, you should paste your key in a new `secret.key` file. That gives the following few lines of code for a simple bot that returns “hello” to whatever you sent to him.

```
import telebot, asyncdispatch, options

const API_KEY = slurp("secret.key")

proc updateHandler(b: Telebot, u: Update) {.async.} =
```

```

let
  response = u.message.get
  message = newMessage(response.chat.id, "hello")
  discard await b.send(message)

let bot = newTeleBot(API_KEY)
bot.onUpdate(updateHandler)
bot.poll(300)

```

Starting a new chat with your existing bot, or a new bot registered with the bot father, would result in a conversation such as that shown in Figure 2-16.

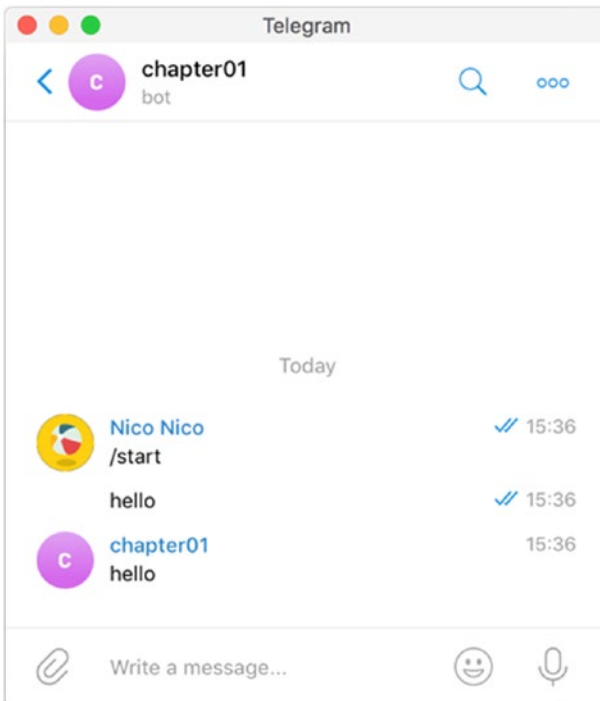
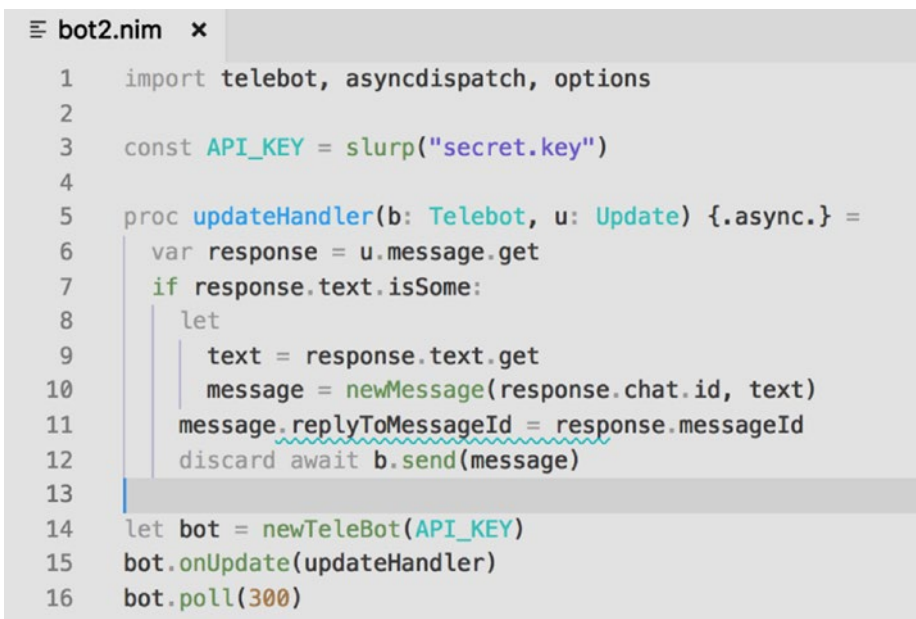


Figure 2-16. *hello*

Replying to Nim Bot

Now that you have the whole setup at hand, creating the second Nim bot is going to be fairly simple. This new bot will inline the reply, as when you are replying directly to a message sent by another user. This is done by setting the `replyToMessageId` property on the created message object.

Note that if you keep the message object declared in the `let` block, the compiler will “compilain” (pun intended) that the message cannot be updated, because it is immutable (Figure 2-17).



```

1  import telebot, asyncdispatch, options
2
3  const API_KEY = slurp("secret.key")
4
5  proc updateHandler(b: Telebot, u: Update) {.async.} =
6      var response = u.message.get
7      if response.text.isSome:
8          let
9              text = response.text.get
10             message = newMessage(response.chat.id, text)
11             message.replyToMessageId = response.messageId
12             discard await b.send(message)
13
14  let bot = newTeleBot(API_KEY)
15  bot.onUpdate(updateHandler)
16  bot.poll(300)
  
```

Figure 2-17. Compilation error that results when using `let` instead of `var`

This is because it was created with `let`, which makes the object immutable. To work around this problem, you simply declare the message object with `var` instead.

```

import telebot, asyncdispatch, options

const API_KEY = slurp("secret.key")

proc updateHandler(b: Telebot, u: Update) {.async.} =
  var response = u.message.get
  if response.text.isSome:
    let
      text = response.text.get
      var message = newMessage(response.chat.id, text)
      message.replyToMessageId = response.messageId
      discard await b.send(message)

let bot = newTeleBot(API_KEY)
bot.onUpdate(updateHandler)
bot.poll(300)

```

The bot now tells you which message it is replying to (Figure 2-18).



Figure 2-18. *hello, hello*

Cats and Dogs Nim Bot

This next bot is only a slight update from the previous bot. Instead of creating a message with `newMessage`, we will use `newPhoto`, which allows us to embed a picture in the message sent to the user.

For this to work, you must have pictures of animals located along the Nim file (`cat.jpg` and `dog.jpg`, for example).

In the Nim code, the path to the file is prefixed with `file://`, and we also add a caption to the picture, by setting the caption field of the photo message.

This gives

```
import telebot, asyncdispatch, options

const API_KEY = slurp("secret.key")

proc updateHandler(bot: TeleBot, update: Update) {.async.} =
  var response = update.message.get
  if response.text.isSome:
    let
      animal = response.text.get
      file = "file://" & animal & ".jpg"
      var message = newPhoto(response.chat.id, file)
      message.caption = animal
      discard await bot.send(message)

let
  bot = newTeleBot(API_KEY)
  bot.onUpdate(updateHandler)
  bot.poll(300)
```

After starting this bot, when you send keywords to it, it will reply with a loaded picture of the given keyword, along with a caption of the animal (Figure 2-19).

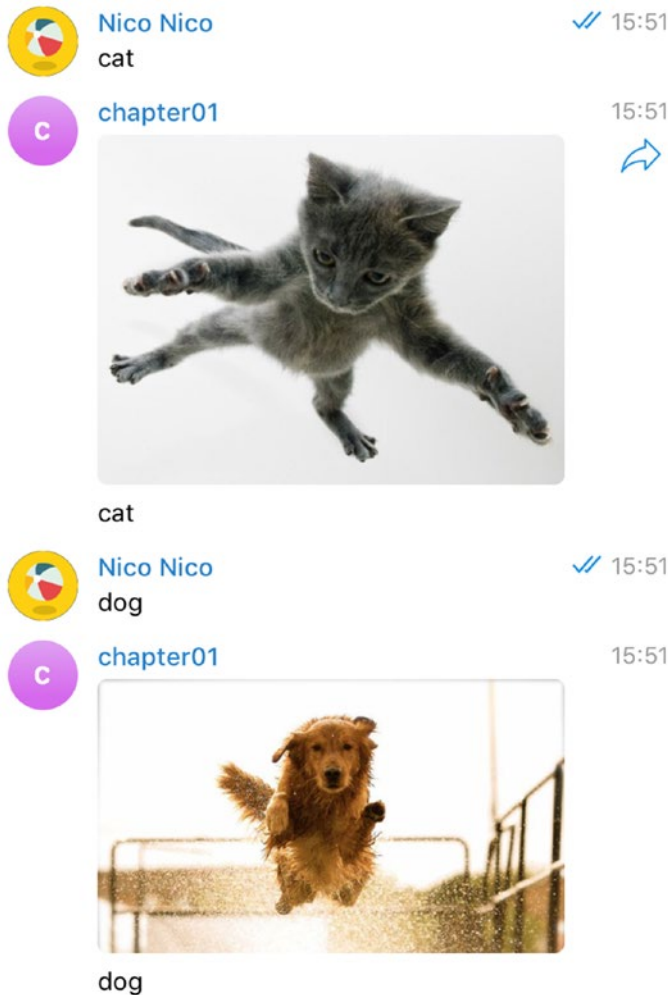


Figure 2-19. Cat and dog

Of course, if you enter an animal that does not have a corresponding file, the bot won't be able to answer.

So, a small exercise would be to send a response to the user, such as "The file could not be found."

There was a lot of time spent on setting up Visual Studio Code in this Nim chapter, but now you are more ready than ever to tackle bots in other languages.

CHAPTER 3

Week 3: Crystal

*Comets, importing change of times and states,
Brandish your crystal tresses in the sky,...*

—William Shakespeare

Henry VI, Part 1, Act 1, Scene 1

Crystal is a new programming language with a syntax heavily influenced by Ruby, to the point that it feels just like writing Ruby. The main difference with Ruby is that Crystal is compiled to C and is, therefore, quite blazingly fast.

Crystal comes with many advantages, such as full support for easy concurrency models, macros, heavy type checks, and one of my favorites: full support for dependencies, via direct Git access, without the need for packaging, and the always-breaking central repository.

In this chapter, in addition to playing with Crystal to develop bots, you will also be introduced to writing your first bot commands. You will remember what commands are. We used them when talking to the bot father, to create our bots. Yes, those are the same! (See Figure 3-1.)

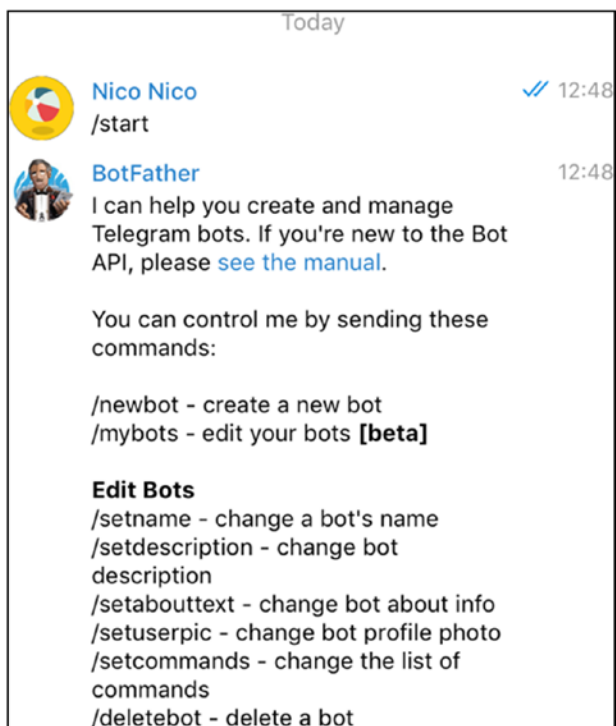


Figure 3-1. *BotFather commands*

Setting Up Crystal

There are already a bunch of installation steps on the Crystal web site, so we will keep those to a minimum here.

<https://crystal-lang.org/docs/installation/index.html>

Note, however, that for Windows users, Crystal can only be used with the Windows Subsystem for Linux (WSL), which requires a recent version of Windows 10. On macOS, Crystal is easily installed via the homebrew, as follows:

```
brew install crystal
```


When you install Crystal with the homebrew, you can see that the Crystal compiler is itself written in Crystal, `.cr` being the common extension for Crystal files.

```
bin/crystal build -D without_openssl -D without_zlib -o .build/crystal src/compiler/crystal.cr --release --no-debug
```

On my favorite Arch Linux-based Manjaro Linux, Pacman has it straight out of the box as well.

```
pacman -S crystal shards
```

On other distributions, you must install a new repository, before running `yum`, `apt-get`, and the likes.

Once Crystal is installed, you should check that `crystal`, but also `shards`, the Crystal package manager, is installed.

```
SuperPinkicious:withcrystal niko$ crystal --version
Crystal 0.26.1 (2018-09-06)
LLVM: 6.0.1
Default target: x86_64-apple-macosx
```

```
SuperPinkicious:withcrystal niko$ shards --version
Shards 0.8.1 (2018-09-06)
```

Short Walk in the Playground

Once installed, Crystal can actually be enjoyed as a stand-alone, in what is called a playground. You can start up a training environment as a web server, using the `play` command.

```
crystal play
```

This will start a web server on the default port 8080. You can also start the playground on a different port, using the `-p` flag.

```
crystal play -p 8090
```

You can type code inside the left panel and directly see the results of the line-by-line execution on the right-hand side (Figure 3-2).

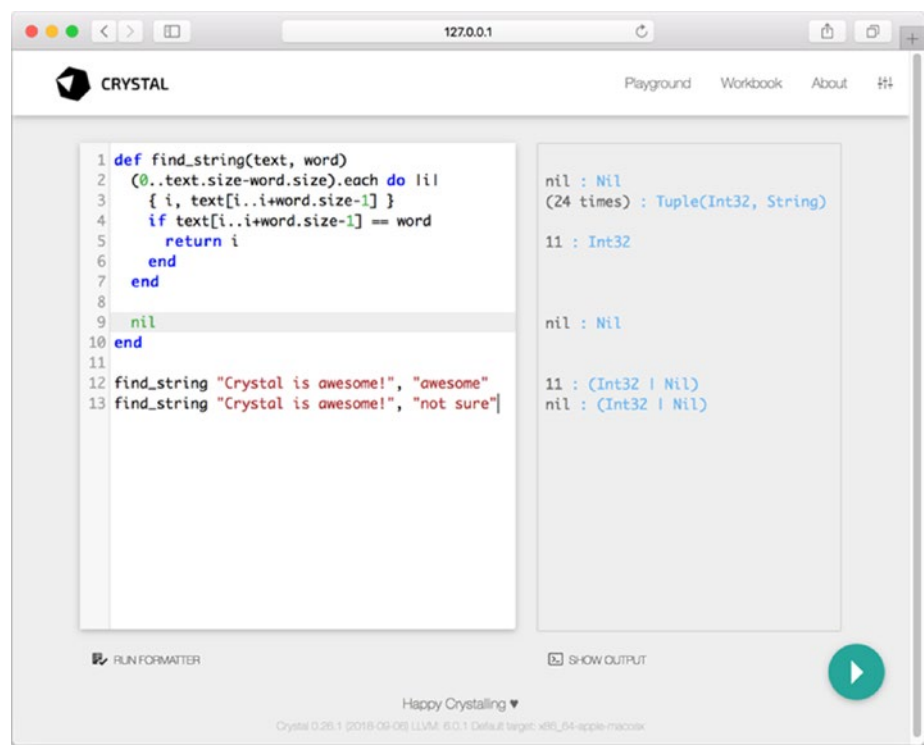


Figure 3-2. *Crystal embedded development environment*

The sheer speed of Crystal can be appreciated quite quickly by running some CPU-intensive function, for example, good old Fibonacci (Figure 3-3).



Figure 3-3. *Fibonacci*

The listing with the benchmark code included gives the following:

```
require "benchmark"

def fib(n : UInt64)
  raise "fib not defined for negative numbers" if n < 0
  new, old = 1_u64, 0_u64
  n.times {new, old = new + old, new}
  old
end

def time(&block)
  puts Benchmark.measure(&block).real
end

time { fib(1000000000_u64) }
```

When executed, as here, on a simple MacBook, it took way fewer than five seconds. So, here we are, Crystal is a language of simplicity and speed.

But let's go back and appreciate it while working and typing code in with Visual Studio Code.

Going Visual Studio Code Again

You are going to write a small program fetching URLs of GitHub projects, based on a GitHub search query. The data returned from an HTTP request to GitHub would be in JSON format, so we will have to decode that too, using some JSON parsing.

In a new file, `requesting.cr`, let's write the code that will send the request and display it in the console.

```
require "http/client"
require "json"
```

```

def search_github(q : String)
  url = "https://api.github.com/search/repositories?q=#{q}"
  HTTP::Client.get(url) do |response|
    res = response.body_io.gets.to_s
    json = JSON.parse(res)
    json["items"][0]["url"]
  end
end

puts search_github("dog")

```

As you can appreciate, the syntax is really, really close to Ruby. Also, note the optional typing of the parameter(s), `q : String`.

The HTTP client and the JSON parser are part of the core language, and thus, no third-party library is required to run the snippet.

Once you have typed the code in the file, Visual Studio Code will, as usual, propose that you find a plug-in in the Marketplace (Figure 3-4).

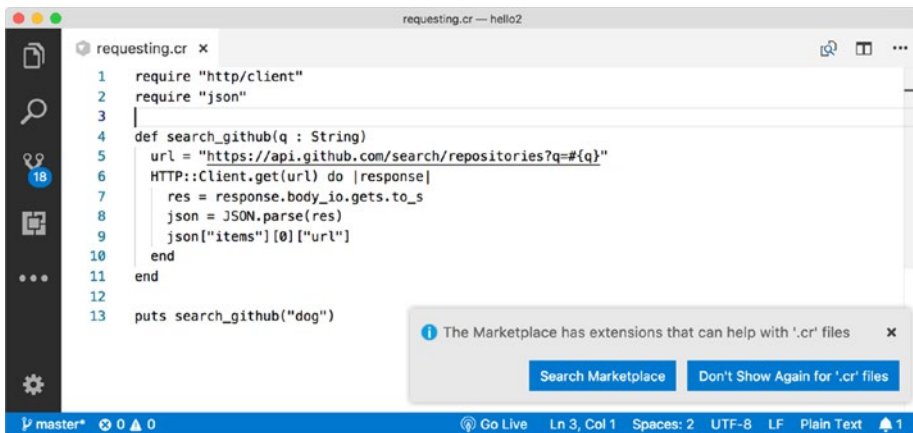


Figure 3-4. Looking for crystals in the Marketplace

I have been having a great experience with the Crystal plug-in of Gerardo Ortega (Figure 3-5), but, of course, feel free to try others.

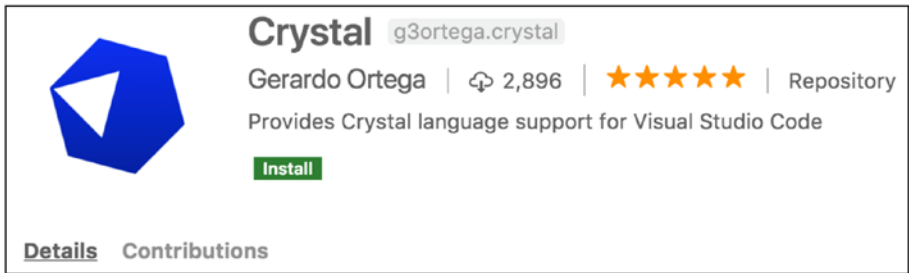


Figure 3-5. *Crystal plug-in*

You may remember from Chapter 2 that you also need a `tasks.json` file, very similar to the one that was used for Nim, to run and execute code, but this time for Crystal. The command to execute/run a file from the command line uses the `crystal` command directly.

```
crystal run <file_name>
```

So, let's write the equivalent command in Visual Studio Code's task file.

```
{
  "version": "2.0.0",
  "tasks":
  [
    {
      "label": "cr",
      "command": "crystal",
      "args": [
        "run",
        "${file}"
      ],
    },
  ],
}
```

```

        "options": {
            "cwd": "${workspaceRoot}"
        },
        "group": {
            "kind": "build",
            "isDefault": true
        }
    }
]
}

```

Note Actually, on macOS, you must modify the `LIBRARY_PATH` variable, so that Crystal can compile the C code properly.

This is done by adding the following line to the Visual Studio Code user settings.

```

"terminal.integrated.env.osx": {
    "LIBRARY_PATH": "/Users/niko/projects/homebrew/lib"
}

```

The settings can be edited, as shown in Figure 3-6, Figure 3-7, and Figure 3-8.

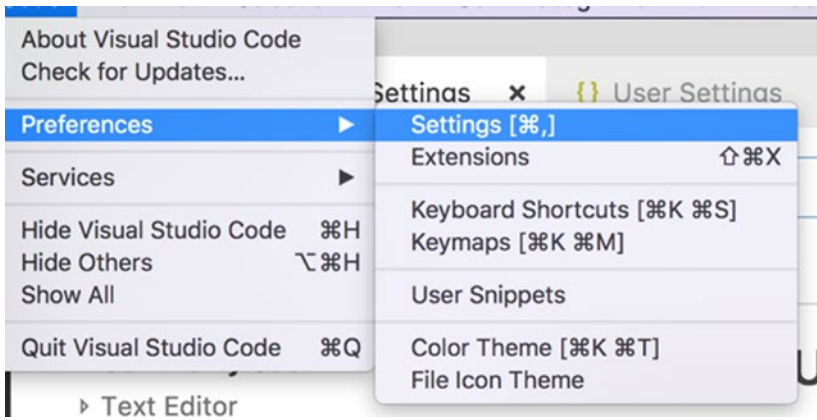


Figure 3-6. Access the settings

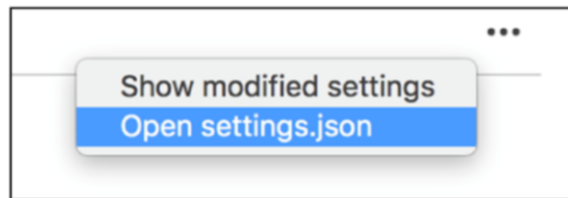


Figure 3-7. Edit the json file

```

"terminal.integrated.env.osx": {
  "LIBRARY_PATH": "/Users/niko/projects/homebrew/lib"
}

```

Figure 3-8. Add the line

Note that when you execute from the command line, you will also have to set the same variable somewhere in your shell environment, using the following:

```
export LIBRARY_PATH=/Users/niko/projects/homebrew/lib
```

This is a unique workaround for macOS and can be safely ignored on other environments.

You are all set now, so a familiar Command+Shift+B (Ctrl+Shift+B) will execute the Crystal code.

The integrated console/terminal of Visual Studio Code will execute the command and display the result of the GitHub query (Figure 3-9).

```
> Executing task: crystal run /Users/niko/Dropbox/BOOKS2/APRESS/03-crystal/hello2/src/requesting.cr <
https://api.github.com/repos/typicode/husky
Terminal will be reused by tasks, press any key to close it.
```

Figure 3-9. First Crystal command

Now, to write the Telegram bot, we are going to require a third-party library. For this to work, we will create a Crystal project, using the built-in command.

Creating a Crystal Project

To create a layout of files for a new Crystal project, you can use the `init` subcommand.

```
SuperPinkicious:crystal niko$ crystal init app mybot
```

```
create mybot/.gitignore
create mybot/.editorconfig
create mybot/LICENSE
create mybot/README.md
create mybot/.travis.yml
create mybot/shard.yml
create mybot/src/mybot.cr
create mybot/spec/spec_helper.cr
create mybot/spec/mybot_spec.cr
```

```
Initialized empty Git repository in /Users/niko/APRESS/crystal/
mybot/.git/
```


Now, if you open the folder in Visual Studio Code and re-create the `tasks.json` to run Crystal code, you should be able to open and execute the (almost empty) `mybot.cr` file (Figure 3-10).

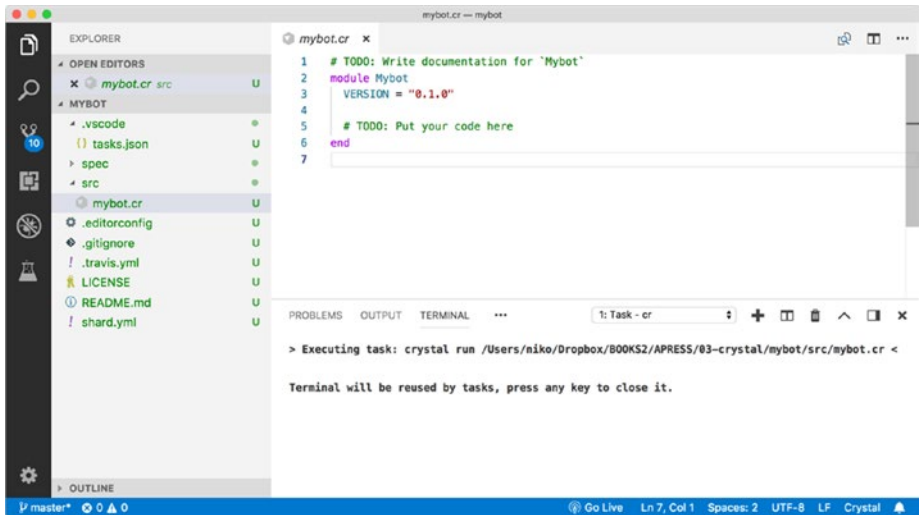


Figure 3-10. Empty—really, really empty—bot

That is probably not the most exciting program you’ve ever seen. Let’s update the code of `mybot.cr` just a little bit, by adding a print statement at the bottom of the file.

```
puts Mybot::VERSION
```

You can execute this directly from Visual Studio Code, of course (Figure 3-11).

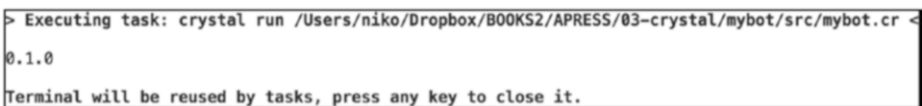


Figure 3-11. Printing the version

You can also execute from the command line, using the run subcommand.

```
SuperPinkicious:mybot niko$ crystal run src/mybot.cr
0.1.0
```

The interesting part is that you can also compile the same code to a binary file, using the build command.

```
crystal build src/mybot.cr
```

Then, directly invoking the compiled file will execute the same print statement, but this time, as a binary file, without dependencies on the Crystal build system.

```
./mybot
>0.1.0
```

Any code you compile with the build command gives you an executable you can reuse at will. This includes the soon-to-arrive Telegram bot.Sweet.

Now, the reason we really wanted to create a Crystal project in the first place was to be able to add the bot API dependency to write Crystal code. In the project folder, there was a file named `shard.yml`. This is the metadata file responsible for handling the project, similar to `pom.xml` or `build.gradle` in Java or the `package.json` file in JavaScript.

The content of the file `shard.yml` is shown following, with the added `telegram_bot` dependency toward the end.

```
name: mybot
version: 0.1.0
authors:
  - Nicolas Modrzyk <hellonico at gmail.com>
targets:
  mybot:
    main: src/mybot.cr
```

```
crystal: 0.26.1
license: MIT

dependencies:
  telegram_bot:
    github: hangyas/telegram_bot
```

The dependency being defined in the `shard.yml` file of the project, you can now retrieve that dependency, using the package manager for Crystal, the `shards` command.

```
SuperPinkicious:mybot niko$ shards
Fetching https://github.com/hangyas/telegram_bot.git
Installing telegram_bot (0.1.0 at HEAD)
```

You will notice a lock file has been created, containing the dependency, and the hash of the commit that was referenced for each dependency.

```
version: 1.0
shards:
  telegram_bot:
    github: hangyas/telegram_bot
    commit: 722bab24876d13a661f513b09c4569916f7a81c1
```

That's pretty much all you require to write your Crystal-based Telegram bot, so let's see all this in action, with a simple echo bot.

Echo Bot

With all that you have seen so far in this chapter and the previous one, the upcoming code should not be too complicated to read. We start by setting up the name and secret Telegram API key of the bot in the constructor, the initialize function. We then move on to implement the handle (message) function, which gets a message as a parameter and replies using the text that was read in the incoming message. Last, we create a new bot, and define its connection as polling.

```
require "telegram_bot"

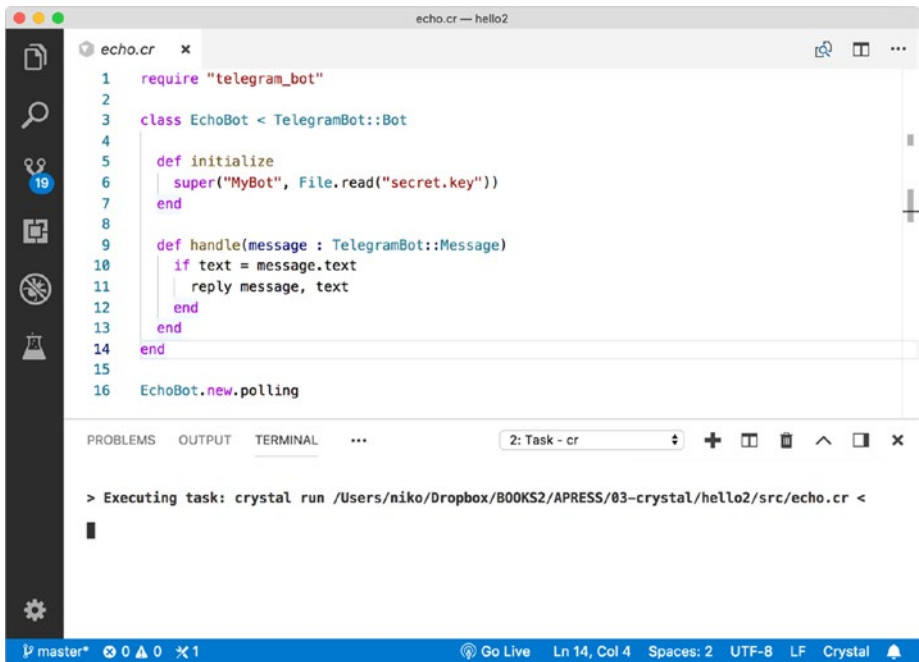
class EchoBot < TelegramBot::Bot

  def initialize
    super("MyBot", File.read("secret.key"))
  end

  def handle(message : TelegramBot::Message)
    if text = message.text
      reply message, text
    end
  end
end

EchoBot.new.polling
```

Then we execute the bot, by using the Command(Ctrl)+Shift+B key shortcut again (Figure 3-12).



```

1  require "telegram_bot"
2
3  class EchoBot < TelegramBot::Bot
4
5      def initialize
6          super("MyBot", File.read("secret.key"))
7      end
8
9      def handle(message : TelegramBot::Message)
10         if text = message.text
11             reply message, text
12         end
13     end
14 end
15
16 EchoBot.new.polling

```

PROBLEMS OUTPUT TERMINAL ... 2: Task - cr

> Executing task: crystal run /Users/niko/Dropbox/B00KS2/APRESS/03-crystal/hello2/src/echo.cr <

master* 0 0 1 Go Live Ln 14, Col 4 Spaces: 2 UTF-8 LF Crystal

Figure 3-12. *Starting the bot*

Back to chatting to this new bot. You can confirm that a nice parrot has been found, and whatever you say will be repeated (Figure 3-13).

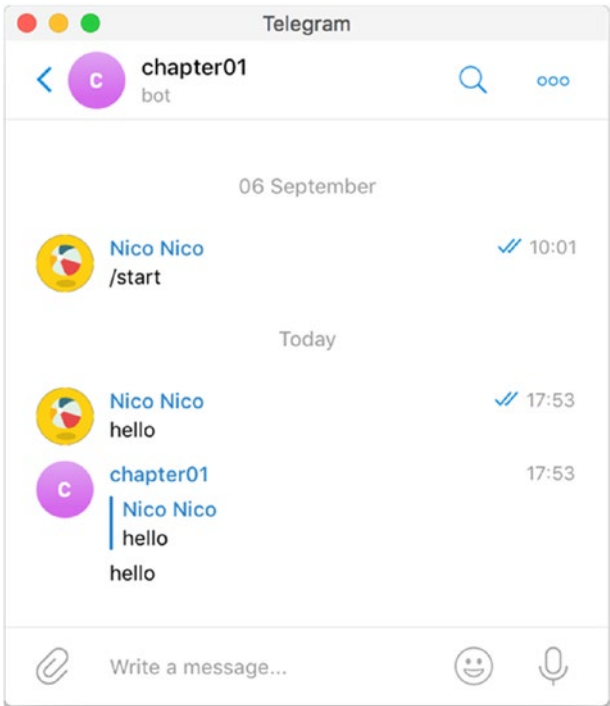


Figure 3-13. *echo, echo, hello, hello*

That was an easy bot. Let’s move on to something more exciting, with the command bot.

Command Bot

Telegram understands the notion of commands. To put it simply, a command is a message to a bot. We know, however, that this accepts a bunch of parameters, and this helps us to write code that is easier to read, without the need for complex parsing of the message.

When it’s an incoming message, the bot will try to find an implemented command for it first. If none is found, it will use the standard handle function.

Our first bot will implement a hello command, which takes no parameter, and just reply to the message, as the handle function would have done.

The code is the following:

```
require "telegram_bot"

class CommandOneBot < TelegramBot::Bot
  include TelegramBot::CmdHandler

  def initialize
    super("MyBot", File.read("secret.key"))

    cmd "hello" do |msg|
      send_message msg.chat.id, "world"
    end
  end
end

my_bot = CommandOneBot.new
my_bot.polling
```

The only difference with the code you have seen up to now is the `cmd` block. The block takes a string for identifier, and its simplest form, the message, as parameter of the block. Sending a command from the chat simply requires a prefix and the name of the command sent to the bot (Figure 3-14).



Figure 3-14. *hello command*

Now, on to our second bot. The second bot is going to use the parameters passed to the command to send an image as a reply.

This time, instead of using `send_message`, we'll use `send_photo`, to send an image back to the chat, depending on the parameter.

```
require "telegram_bot"

class CommandOneBot < TelegramBot::Bot
  include TelegramBot::CmdHandler

  def initialize
    super("MyBot", File.read("secret.key"))

    cmd "animals" do |msg, params|
      send_photo msg.chat.id, File.new("#{params[0]}.jpg")
    end
  end
end

my_bot = CommandOneBot.new
my_bot.polling
```

Now, provided you have a few images in the project folder to match the animals' names, the `/animals` command will fetch the image as a file and send it back to the chat room (Figure 3-15).

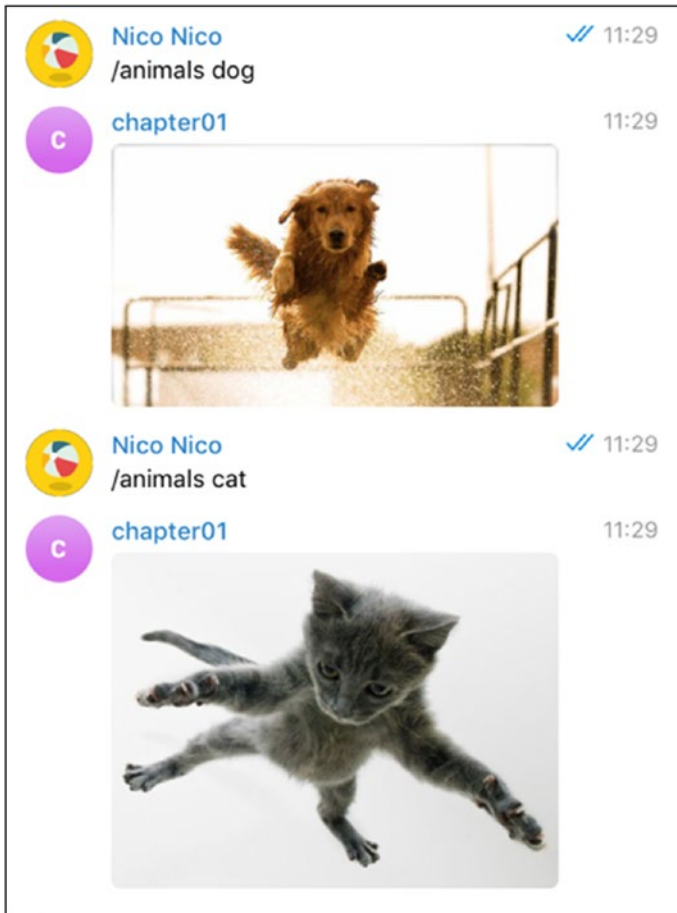


Figure 3-15. */animals are cute!*

Now, you can try a few different commands, using other bot API functionalities.

Most of the Telegram bot APIs are implemented in Crystal's `telegram_bot`, and following is a list of the different functions that can be used to send types of messages and media to the chat.

- `send_message`
- `send_photo`

- `send_audio`
- `send_document`
- `send_sticker`
- `send_video`
- `send_voice`
- `send_video_note`
- `send_media_group`
- `send_location`
- `send_venue`
- `send_contact`
- `send_chat_action`
- `send_game`
- `send_invoice`

But now you'll have to wait until next week, before you can see inline messages in action.

CHAPTER 4

Week 4: Rust

No great movement designed to change the world can bear to be laughed at or belittled. Mockery is a rust that corrodes all it touches.

—Milan Kundera

If you have not heard about the Rust programming language yet, I think you should. Rust won first place for “most-loved programming language” in the [Stack Overflow](#) Developer Survey in 2016, 2017, and 2018.

As noted on the Rust web site (www.rust-lang.org), the language has a long list of advantages, such as the following:

- Zero-cost abstractions
- Move semantics
- Guaranteed memory safety
- Threads without data races
- Trait-based generics
- Pattern matching
- Type inference
- Minimal runtime
- Efficient C bindings

Rust is sponsored by Mozilla, but more than just being backed by a single company, it is also coded by a very supportive and active community. Rust is very much concerned with robust threaded environments and related memory safety concerns. These are especially suited to solving problems related to developing a concurrent end point for bots, as this chapter will explore.

Rust Installation and First Steps

Rust is maintained and managed on your local machine using `rustup`. Yes, this is an environment manager specific to Rust. It is similar to Ruby Version Manager (RVM) but fully part of the Rust tool chain and not a third-party team, so it feels very integrated.

Installation

The following two links are the entry points to get `rustup` and, thus, Rust installed on your machine:

www.rust-lang.org/en-US/install.html

www.rust-lang.org/en-US/other-installers.html

Depending on whether you are on macOS/Linux (Figure 4-1) or Windows (Figure 4-2), you will get a different page, but the installation process is quite similar.

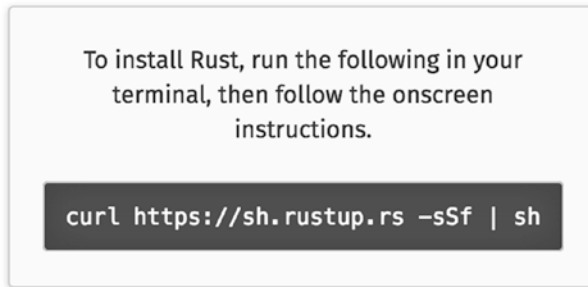


Figure 4-1. *Installation box on OSX/Linux*

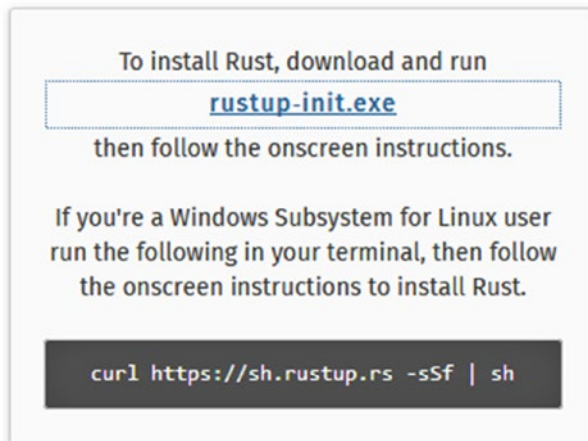


Figure 4-2. *Installation box on Windows*

On Windows, you also require some extra tooling, such as the Microsoft C++ build tools (Figure 4-3).

Figure 4-3. C++ tooling required on Windows

As indicated in the console output, the Microsoft build tools can be found on the following web site: <https://aka.ms/buildtools>. You can download and install (Figure 4-4 and Figure 4-5) by following the download link.

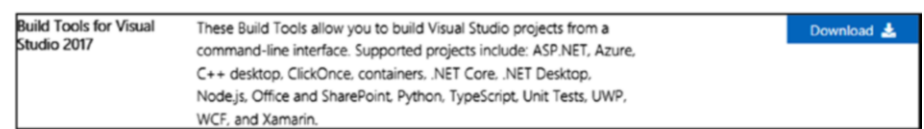


Figure 4-4. Build tools for Visual Studio

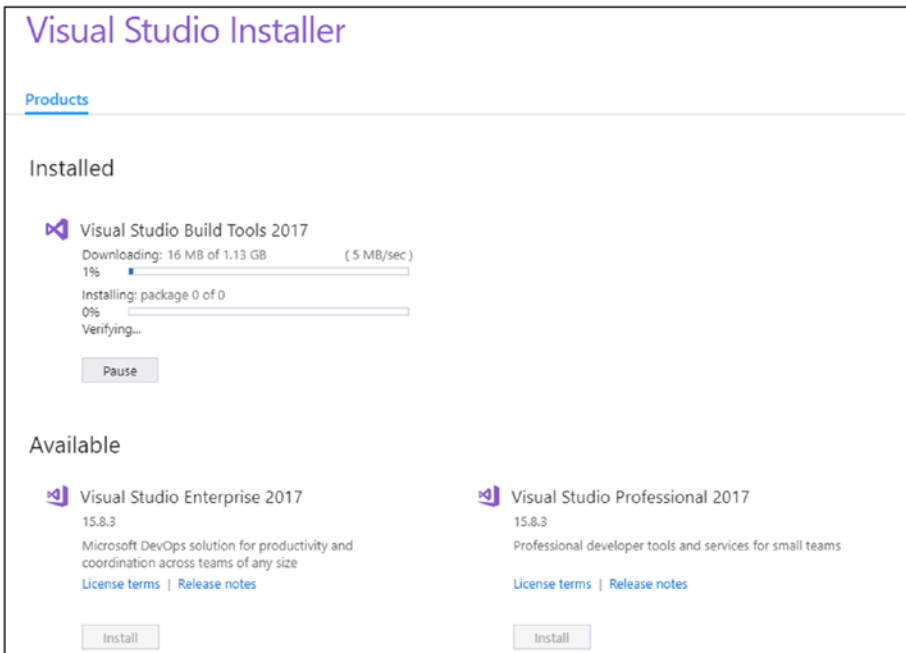


Figure 4-5. *Installing the build tools*

Rust has a fast-paced six-week update cycle, so by the time you read this, the version will be different, but at the time of writing, this is the Rust current version, obtained with the `rustup show` command:

```
SuperPinkicious:rust-01 niko$ rustup show
Default host: x86_64-apple-darwin
stable-x86_64-apple-darwin (default)
rustc 1.28.0 (9634041f0 2018-07-30)
```

Now let's see how to compile your first Rust program.

First Rust or Two

The first Rust program will print a simple Hello World message. This isn't complicated, but it will help you get used to the Rust compilation and execution tool chain.

Hello Rust

This, our first Rust program,

- Declares a `main` function, using `fn`
- Prints to the standard output with the `println!` function (note the exclamation point)
- Ends statements with a semicolon (`;`)

This gives you the following simple and short code snippet:

```
fn main() {  
    println!("Hello World!");  
}
```

You can save the snippet in a file named `first.rs`, for example.

As a side note, if you type the preceding snippet in Visual Studio Code, the editor will recognize the `rs` file extension and will ask you to install a related plug-in, to handle `rs` files (Figure 4-6).

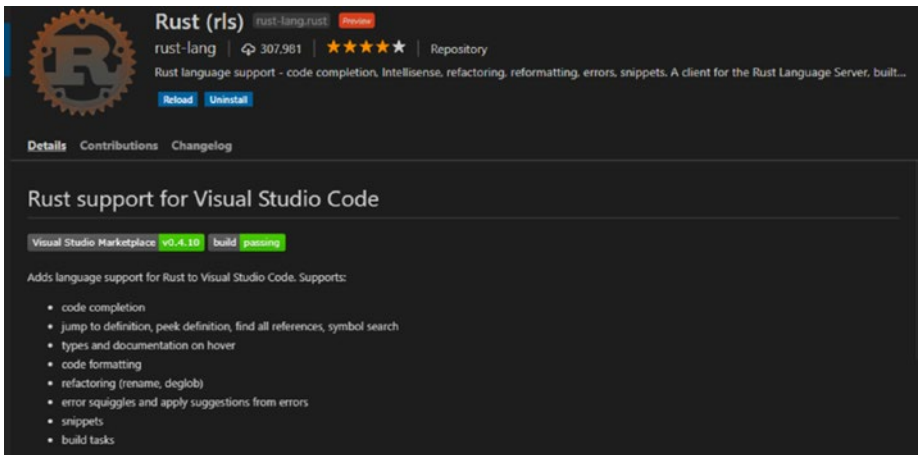


Figure 4-6. *Visual Studio Code Rust plug-in*

Rust being a compiled language, you must first handle a compilation step before execution of the code. This is done using the `rustc` command, which is installed when you install `rustup`. The command is as follows:

```
rustc first.rs
```

In the same folder in which you run the command, you will notice that you now have a new binary file, without the `.rs` file extension. Executing the binary is just a matter of typing the command name.

```
$ ./first
Hello World!
```

This works both for Windows and macOS/Linux, but, of course, you cannot bring one and make it run onto the other as is. You'll have to compile things again. While `rustup` can help you cross-compile for other platforms, that is beyond the scope of this book.

For other ramp-up examples, make sure to review the excellent Rust by Example web site (<https://doc.rust-lang.org/rust-by-example/hello.html>), which has all you need to get started with the language's constructs.

Going forward, let's review how good old Fibonacci numbers can be expressively generated with Rust.

Fibonacci

Rust has very convenient pattern-matching branching, which can be written with the appropriately named `match`. We will use the construct to write the Fibonacci generators.

In a new file, `fibonacci.rs`, we declare the `fibonacci` function, including the parameter types for input and output—here, 64-bit integers for both inputs and outputs—to avoid overflows.

Also, note, in the following code snippet, how you use `_` to match any other value for `n`.

```
fn fibonacci(n: u64) -> u64 {
    match n {
        0 => 0,
        1 => 1,
        _ => fibonacci(n - 1) + fibonacci(n - 2)
    }
}
```

Let's put this code with a main function, so that we can call the program with an input from the command line.

```
use std::env;

fn main() {
    if let Some(arg1) = env::args().nth(1) {
        let myint = arg1.parse().unwrap();
```

```

        println!("{}", fibonacci(myint));
    } else {
        println!("missing input")
    }
}

```

In the preceding code snippet, note how explicit null checks are avoided by using `Some` and how the string parameter is turned into an integer, `myint`, using `parse` and `unwrap`.

`unwrap` is used to turn the `Some(something_inside)` to `something_inside`. Here, we have the `aif/else` block to confirm whether we have a value as an argument.

If you are not sure whether you have a value after `Sum`, you can also use `unwrap_or`, to give a default value.

Compiling gives you an executable, as in the previous Hello World example.

```
rustc fibo.rs
```

Running shows that the Fibonacci generator program works as expected.

```

$ ./fibo 10
55
$ ./fibo 15
610
$ ./fibo 30
832040

```

While Rust can be enjoyed on its own, you usually resort to cargo, the Rust build tool, to create a project structure and manage project dependencies.

Ride the cargo

cargo is a command-line tool that is also installed and set up by rustup. cargo is responsible for creating new projects, managing the project structure, adding dependencies, generating binaries, running code, testing code, etc.

A list of the most important cargo subcommands is shown following:

Some common cargo commands are (see all commands with `--list`):

<code>build</code>	Compile the current project
<code>clean</code>	Remove the target directory
<code>doc</code>	Build this project's and its dependencies' documentation
<code>new</code>	Create a new cargo project
<code>init</code>	Create a new cargo project in an existing directory
<code>run</code>	Build and execute <code>src/main.rs</code>
<code>test</code>	Run the tests
<code>bench</code>	Run the benchmarks
<code>update</code>	Update dependencies listed in <code>Cargo.lock</code>
<code>search</code>	Search registry for crates
<code>publish</code>	Package and upload this project to the registry
<code>install</code>	Install a Rust binary
<code>uninstall</code>	Uninstall a Rust binary

As a first cargo test, let's write a program that displays Coordinated Universal Time (UTC) and local times, using a third-party library named `chrono`.

We Have Time

Obviously, before we add a library to a project, we first require a project. This is done via the new subcommand of cargo.

```
$ cargo new we-have-time
Created binary (application) `we-have-time` project
```

The type of project generated by cargo is a binary application by default, but you can also prepare a library without a runnable entry point. Once you have the project generated, you can see a very basic project structure and a few files and folders.

```
$ tree -L 2
.
├── Cargo.toml
└── src
    └── main.rs
```

In the preceding, `Cargo.toml` is the project metadata, and `main.rs` is the same kind of Rust source file we have seen up to now. `Cargo.toml` is the location to add the dependencies.

```
$ cat Cargo.toml
[package]
name = "we-have-time"
version = "0.1.0"
authors = ["Nicolas Modrzyk <myemail@gmail.com>"]

[dependencies]
```

Now as we would like to play with dates and time zones, let's use cargo again, to search for a time library.

```
$ cargo search time
    Updating registry `https://github.com/rust-lang/crates.
    io-index`
time = "0.1.40"           # Utilities for working with time-
                           related functions in Rust.
```

```
exonum-time = "0.9.2"      # The time oracle service for
                             Exonum.
chrono = "0.4.6"           # Date and time library for Rust
faster_path = "0.0.2"      # Alternative to Pathname
specs_time = "0.5.1"       # time resource for specs
... and 1738 crates more (use --limit N to see more)
```

chrono makes it easy to display dates in different time zones, so we'll just use that library. To make chrono available to your project, you add its name and version to the Cargo.toml file, in the dependencies section.

This gives

```
[dependencies]
chrono = "0.4.6"
```

After adding a library, or to validate the code written in your project, you can usually make use of cargo check. This also comes in handy after adding a dependency, to make sure it has been downloaded and installed properly on your local machine.

```
$ cargo check
Compiling num-traits v0.2.5
Compiling num-integer v0.1.39
Checking libc v0.2.43
Checking time v0.1.40
Checking chrono v0.4.6
Checking we-have-time v0.1.0 (file:///Users/niko/Dropbox/
BOOKS2/APRESS/04-rust/we-have-time)
Finished dev [unoptimized + debuginfo] target(s) in 7.69s
```

Now let's move on to coding and update the Rust code, located in main.rs, to make use of the chrono library.

1. First, we must tell the program that we are using an external library, which Rust calls `crate`. This is done using the `extern crate chrono` notation. This is done only once in the lifetime of a program.
2. Next, we must import the Rust symbols in the current file/namespace, and this is done via `use`.
3. `let` defines immutable variables, each with its own type.
4. And, as before, `println!` prints this on the standard output.

This gives the following code snippet (again in the `main.rs` file):

```
extern crate chrono;
use chrono::prelude::*;

fn main() {
    let utc: DateTime<Utc> = Utc::now();
    println!("{}", utc);
    let local: DateTime<Local> = Local::now();
    println!("{}", local);
}
```

Because we are using cargo to manage our project, there are now a few shortcuts we can take advantage of to play with the run, such as running code directly, by using `cargorun`.

```
$ cargo run
Compiling num-traits v0.2.5
Compiling num-integer v0.1.39
Compiling libc v0.2.43
Compiling time v0.1.40
Compiling chrono v0.4.6
```

```
Compiling we-have-time v0.1.0 (file:///we-have-time)
Finished dev [unoptimized + debuginfo] target(s) in 9.11s
Running `target/debug/we-have-time`
2018-09-12 05:41:56.872180 UTC
2018-09-12 14:41:56.872309 +09:00
```

See the two dates displayed at the end of the compilation steps? These are the standard output of the program.

`run` is the de facto subcommand for writing and running Rust programs quickly. To create the final version, `cargo` has the built-in subtask `build`. `build` is responsible for creating a stand-alone binary file.

```
$ cargo build
Finished dev [unoptimized + debuginfo] target(s) in 0.16s
```

Note, too, that `cargo` supports incremental compilation and will only recompile code that was changed since the last execution.

Running the binary gives the same output—everything is working so well. This is actually refreshing.

```
$ ./target/debug/we-have-time
2018-09-12 06:01:54.100082 UTC
2018-09-12 15:01:54.100176 +09:00
```

You will also see later how to generate a release version of your program, using the same `cargo build` command, but with extra flags.

For now, let's sidestep and see how we can declare multiple targets within the same `cargo` project, meaning different entry points for execution.

Multiple Cargo Targets

You may have noticed that each of the commands `run` is taking the `main.rs` file as the default input. For some time, I have been looking for a way to run different files, and you actually cannot pass a file directly to `cargo`. In

the `Cargo.toml` file, you must define what is known as a binary target. This is done by defining a block in the `.toml` file, using `[[bin]]`.

At a minimum, the block takes a name and a path, so let's define two binary targets, one with the existing `main.rs` file and another with a new Rust source code file: `main2.rs`.

```
[[bin]]
name = "main1"
path = "src/main.rs"

[[bin]]
name = "main2"
path = "src/main2.rs"
```

For the purpose of multiple targets, we will keep the `main2.rs` source code ultra-minimal here, only outputting the current date in the UTC time zone, but, of course, it could be as complicated as you can code it.

```
extern crate chrono;

use chrono::prelude::*;

fn main() {
    let utc: DateTime<Utc> = Utc::now();
    println!("{}", utc);
}
```

Now, when running cargo commands, you can specify which binary target to run by specifying the `--bin` flag to cargo, as shown twice in the following code, once with target binary `main1`

```
$ cargo run --bin main1
    Finished dev [unoptimized + debuginfo] target(s) in 0.16s
    Running `target/debug/main1`
```

CHAPTER 4 WEEK 4: RUST

2018-09-12 06:26:11.634208 UTC

2018-09-12 15:26:11.634336 +09:00

and a second time with target binary main2

```
$ cargo run --bin main2
```

```
Finished dev [unoptimized + debuginfo] target(s) in 0.05s
```

```
Running `target/debug/main2`
```

2018-09-12 06:26:13.387135 UTC

As a bonus, here is the `tasks.json` you can use within Visual Studio Code to run any of the two binary targets from visual code.

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "main1",
      "command": "cargo",
      "args": [
        "run",
        "--bin",
        "main1"
      ],
      "options": {
        "cwd": "${workspaceRoot}"
      },
      "group": {
        "kind": "build",
        "isDefault": true
      }
    },
  ],
}
```

```

{
  "label": "main2",
  "command": "cargo",
  "args": [
    "run",
    "--bin",
    "main2"
  ],
  "options": {
    "cwd": "${workspaceRoot}"
  },
  "group": {
    "kind": "build",
    "isDefault": true
  }
}
]
}

```

In addition, by using Command(Ctrl)+Shift+B, you can now select which cargo binary target to run (Figure 4-7).

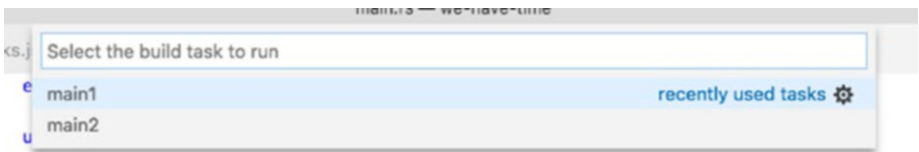


Figure 4-7. Binary target from Visual Studio Code

Now you have all you need to start enjoying writing bots in Rust. So, let's bot.

Rust Bot Number 1: Reply to Me

To create bots, we'll again make use of a Telegram wrapper. To find one, remember how you used cargo to search for a dependency: the cargo search command.

```
$ cargo search telegram
    Updating registry `https://github.com/rust-lang/crates.
    io-index`
telegram = "0.2.0"                # Unofficial Telegram API
                                   Library
telegram-bot = "0.6.1"           # A library for creating
                                   Telegram bots
telegram-bot-raw = "0.6.1"       # Telegram Bot API types
teleborg = "0.1.32"             # A Telegram bot API.
... and 35 crates more (use --limit N to see more)
```

We'll go for the telegram-bot version "0.6.1". So, having created a new project with cargo, we will update Cargo.toml accordingly.

```
[dependencies]
telegram-bot = "0.6"
tokio-core = "0.1.17"
futures = "0.1.23"
```

Ah, yes, you'll also have to spawn threads (futures) and have a cool messaging system (tokio-core), so we add those two in the dependencies section too.

In the same Cargo.toml file, we will also define a first binary target.

```
[[bin]]
name = "bot1"
path = "src/bot1.rs"
```

I will let you define more binary targets (and update the `tasks.json`), as you delve further into this chapter.

The core purpose of the bot is, as usual, to get an update from the Telegram servers, by polling, and handle the message associated with the update. Here, we retrieve the text of a pure text message, as well as references to the first name of the person who wrote the message. `match`, which you have seen with Fibonacci, is again used to match on the message type—here, `MessageKind::Text`.

Finally, once we have all the elements, we must send a reply (`first_name`, test data), then call `text_reply` inside a newly spawned thread—here, in fire-and-forget mode.

```
if let UpdateKind::Message(message) = update.kind {
    match message.kind {
        MessageKind::Text {ref data, ..} => {
            api.spawn(message.text_reply(
                format!("Hi, {}! You just wrote
                    '{}'",
&message.from.first_name, data)
            ));
        },
        _ => {println!("which message?")}
    }
}
```

In case the type of the message is not a text message, we just ignore it altogether, with the default section `_` of the `match` block. The rest of the code involves some gymnastics, to get the bot to be in a threaded environment with message passing, as described in more detail on the [telegram_botgithub](#) page.

This is not that important to write the bot, so we'll just assume we have to write it. Finally, the full listing of `bot1.rs` is shown following:

```
extern crate futures;
extern crate telegram_bot;
extern crate tokio_core;

use std::env;

use futures::Stream;
use tokio_core::reactor::Core;
use telegram_bot::*;

fn main() {
    let mut core = Core::new().unwrap();

    let token = env::var("TELEGRAM_BOT_TOKEN").unwrap();
    let api = Api::configure(token).build(core.handle()).unwrap();

    let future = api.stream().for_each(|update| {
        if let UpdateKind::Message(message) = update.kind {
            if let MessageKind::Text {ref data, ..} = message.
            kind {
                println!("<{}>: {}", &message.from.first_name,
                    data);

                api.spawn(message.text_reply(
                    format!("Hi, {}! You just wrote '{}'",
                        &message.from.first_name, data)
                ));
            }
        }
    })
}
```

```

        Ok(())
    });

    core.run(future).unwrap();
}

```

Now you can start this bot from the command line or from Visual Studio Code (Figure 4-8).

```

> Executing task: cargo run --bin bot1 <

Compiling bots v0.1.0 (file:///Users/niko/Dropbox/B00KS2/APRESS/04-rust/bots)
Finished dev [unoptimized + debuginfo] target(s) in 4.72s
Running `target/debug/bot1`
<Nico>: yey

```

Figure 4-8. Starting the Rust bot from Visual Studio Code

And confirm the usual output from a Telegram chat, as shown in Figure 4-9.

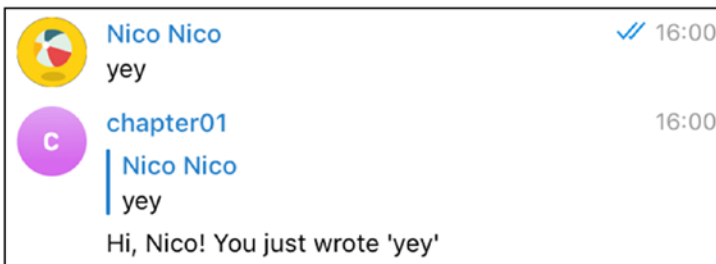


Figure 4-9. Your first Rust bot

Rust Bot Number 2: Where Is Tokyo?

It's always nice to have a bot that tells you where something is or where to head. The next bot will send you a live location, with a time-out directly in the chat.

While the main function is almost identical to that in the previous example, in terms of preparing the bot and setting up the channeling, the meat of the following exercise, is actually to send the live location.

This is done using the `location_reply` function on the Rust object message. We can also specify directly here how long the location live stream will run.

In the following code, we stream the location of Tokyo (lat 35.652832° N, long 139.839478° E) for 60 seconds.

Again, most of this Rust Telegram API is based on preparing futures, an object from the futures library ready to be executed in a separate thread and spawned using tokio handles.

The core bot handle and message answers are done this way. Following is the code snippet for the location bot.

```
extern crate futures;
extern crate telegram_bot;
extern crate tokio_core;

use std::env;

use futures::{Future, Stream};
use tokio_core::reactor::{Handle, Core};
use telegram_bot::*;

fn where_is_tokyo(api: Api, message: Message, handle: Handle) {
    let api_future = || Ok(api.clone());
    let future = api.send(message.location_reply(35.652832,
    139.839478).live_period(60)).join(api_future());
    handle.spawn(future.then(|_| Ok(())))
}

fn main() {
    let token = env::var("TELEGRAM_BOT_TOKEN").unwrap();
    let mut core = Core::new().unwrap();
```



```

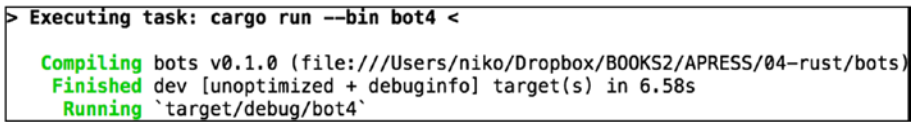
let handle = core.handle();
let api = Api::configure(token).build(core.handle()).unwrap();

let future = api.stream().for_each(|update| {
    if let UpdateKind::Message(message) = update.kind {
        where_is_tokyo(api.clone(), message.clone(), handle.clone())
    }
    Ok(())
});

core.run(future).unwrap();
}

```

With the bot started from Visual Studio Code again (task setup is left to the reader), we can see the compilation steps succeeding and running the compiled target directly (Figure 4-10).



```

> Executing task: cargo run --bin bot4 <
Compiling bots v0.1.0 (file:///Users/niko/Dropbox/B00KS2/APRESS/04-rust/bots)
Finished dev [unoptimized + debuginfo] target(s) in 6.58s
Running target/debug/bot4

```

Figure 4-10. Starting location bot

I did not specify any filtering or specific message handling, so sending any message to the bot will do. Here, a brief “hello” message notifies the bot that we want to know where it is (Figure 4-11).

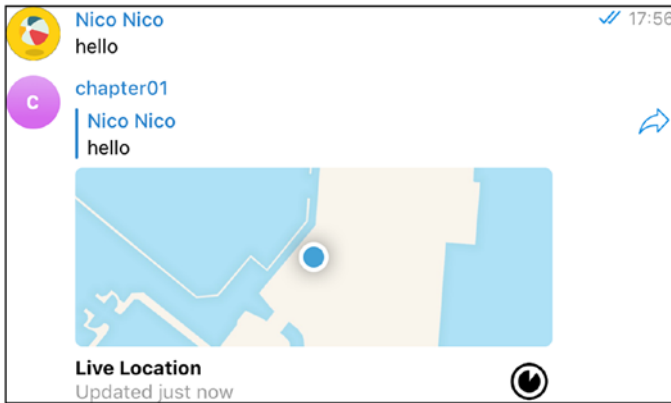


Figure 4-11. *Live from Tokyo*

Notice, too, in the figure how the clock-looking timer is slowly decreasing, depending on the amount of time we told the live location to run for. A fun exercise using the first version of this bot is actually to edit the location along the way.

Now, let's say we want to send a location in central Tokyo first, and then slowly move east across Japan. We'll use a bunch of the same patterns to run a call asynchronously, join on the newly spawn thread with a time-out, make sure the call is finished, join again, and move on to the next location update.

This is handled as follows:

```
let future = api.send(message.location_reply(35.652832,
139.839478).live_period(60))
    .join(api_future()).join(timeout(10))
    .and_then(|((message, api), _)| api.send(message.edit_
live_location(35.652832, 138.839478)))
    .join(api_future()).join(timeout(10))
    .and_then(|((message, api), _)| api.send(message.edit_
live_location(35.652832, 137.839478)))
    .join(api_future()).join(timeout(10))
    .and_then(|((message, api), _)| api.send(message.edit_
live_location(35.652832, 136.839478))));
```

On the Telegram desktop client, it appeared that the Telegram map was not updating in real time very well, but on the native Android and iOS apps, location was updated in real time. The updates on an Android-based handset are shown in Figures 4-12 and 4-13.

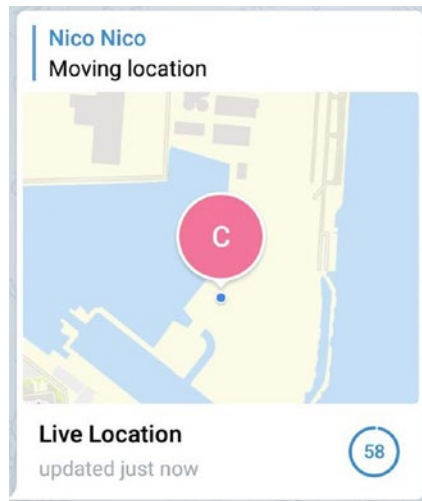


Figure 4-12. Tokyo

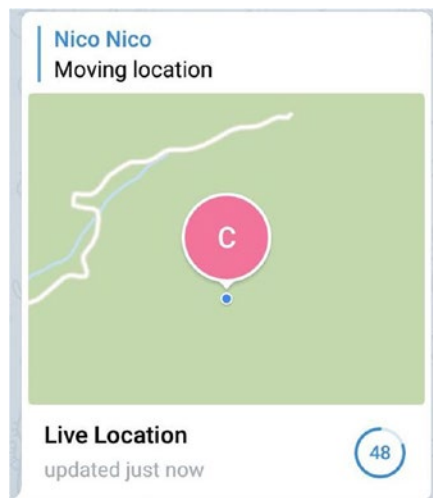


Figure 4-13. East of Tokyo

Rust Bot Number 3: Chained Reaction

In the same vein, you could also chain messages for your chat flow, as needed. This new `multiple_messages` function presents multiple ways to send messages and chain them to the chat, one after the other.

Here, we'll employ similar constructs, with some changes to how the text is parsed, or not, before sending the message. `text_reply` can be substituted with `message.from.text`, to avoid this reply-looking message sent in many examples.

```
fn multiple_messages(api: Api, message: Message, handle:
&Handle) {

    let simple =
        api.send(message.text_reply("Simple message"));
    let markdown =
        api.send(message.text_reply("- Markdown message \n-
line 2\n- line 3").parse_mode(ParseMode::Markdown));
    let html =
        api.send(message.text_reply("<b>Bold HTML message
</b>").parse_mode(ParseMode::Html));
    let private =
        api.send(message.from.text("Private text"));
    let private_html =
        api.send(message.from.text(format!("<b>Private text
</b>")).parse_mode(ParseMode::Html));
    let preview =
        api.send(message.text_reply("Message with preview
https://telegram.org"));

    handle.spawn({
        let future = simple
            .and_then(|_| markdown)
```

```

        .and_then(|_| private)
        .and_then(|_| private_html)
        .and_then(|_| preview)
        .and_then(|_| html);

    future.map_err(|_| ()).map(|_| ())
  })
}

```

Now, finally, when chatting with this new Telegram bot, you can observe all the messages popping up, queued one by one in Figure 4-14.

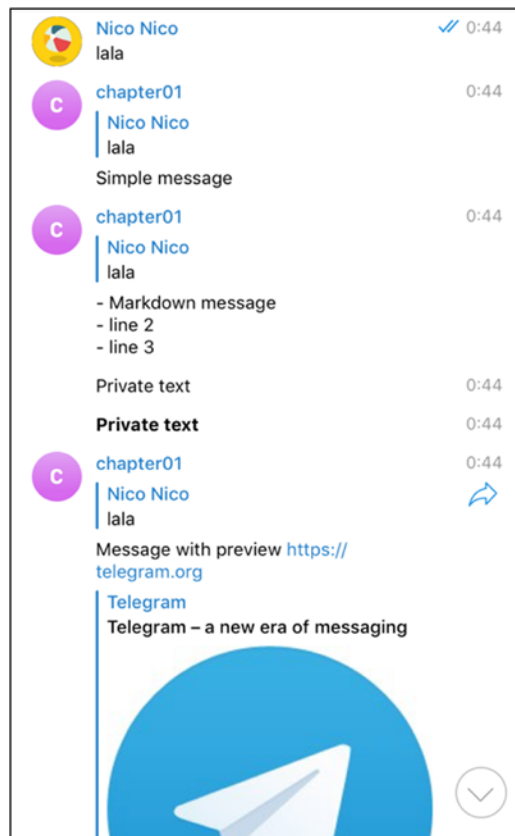


Figure 4-14. All different types of send requests from Rust

Once you have fully finished developing your Rust bot, do not forget to generate a binary for it.

Compiling for Release

Compiling for release with cargo is easier than it seems. It is done simply by passing the `--release` flag to the cargo `build` command. This will have the side effect of rebuilding all the dependencies in release mode as well.

```
$ cargo build --bin bot2 --release
```

In the following, you can see how this build command in release mode is followed by a bunch of compiling lines:

```
Compiling nodrop v0.1.12
  Compiling cfg-if v0.1.4
  Compiling libc v0.2.42
  Compiling byteorder v1.2.3
  Compiling scopeguard v0.3.3
  Compiling memoffset v0.2.1
  Compiling lazy_static v1.0.2
  Compiling lazycell v0.6.0
  Compiling slab v0.4.0
  Compiling futures v0.1.23
  ...
```

The optimized compiled code can be run in the same way as its debug counterpart.

```
$ ./target/release/bot2
```

You should also appreciate the difference in size between the debug and release versions of the program. The release version is at least one-third the size of the debug one.

CHAPTER 5

Week 5: D

“As for difficulties,” replied Ferguson, in a serious tone, “they were made to be overcome.”

—Jules Verne

Five Weeks in a Balloon

To be honest, I haven’t yet used the D language for heavyweight projects, but after a few small coding challenges, I have been very impressed by D’s sheer speed of execution.

D is a statically typed language, which very much resembles C++ and Java, with some additional conveniences. The learning curve would not be that steep for those experienced in either of those two languages.

On the list quoted on the D language web site (<https://tour.dlang.org/>), the following features are highlighted:

- *High-level* constructs for great modeling power
- *High-performance*, compiled language
- Static typing
- Direct interface to the operating system APIs and hardware
- Blazingly fast compile times
- Memory-safe subset (SafeD)

- *Maintainable, easy-to-understand code*
- Gradual learning curve (C-like syntax, similar to Java and others)
- Compatible with the C application binary interface
- Limited compatibility with the C++ application binary interface
- Multi-paradigm (imperative, structured, object-oriented, generic, functional programming purity, and even assembly)
- Built-in error detection (contracts, unittests)

D also comes with a highly multi-threaded web framework named `vibe.d`, which I won't cover in detail here, but you should have a look at the simplicity of the servers built with it.

Before I move on to bot programming, we'll look over some basic D constructs and code examples, including, of course, some Fibonacci.

In addition, once you have mastered some D coding and project dependencies management, we will be moving to the core of every chapter of this book, Telegram bot coding.

Installation and First D Steps

The D language comes with all sorts of packages and installers, which are available from its web site at <https://dlang.org.html> (Figure 5-1).

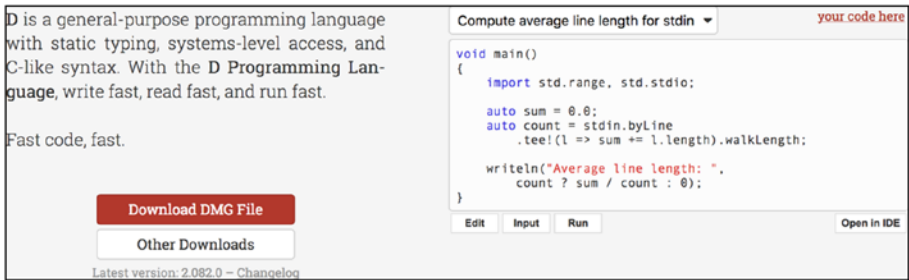


Figure 5-1. D language download page, with samples

Once you have finished downloading and installing, you should have two main binaries available to you, namely:

- dmd: The reference D compiler
- dub: The build tool that goes along with dmd

Our first D program is going to exhibit some courtesy, by saying hello, as every first program should.

Import statements are quite intuitively done, using `import`, and the main entry point of the program is defined via the `main()` function. Note that there is no need to define an array of string parameters to this main function. The return value of the main function is also optional.

Printing to the standard output is done with `writeln`, which is located in the `std.stdio` package, which we will import beforehand.

You can save the following sample code in a `hello.d` file, with `.d` being the default extension for D source code.

```
import std.stdio;

void main() {
    writeln("hello D!");
}
```

D source code is compiled with the `dmd` command. In its two simplest forms, the command can be run as either one of the following:

```
dmd <source_file>
dmd -run <source_file>
```

The first form will generate a binary file of the same name as the entry source file. You can then execute the binary file directly on the host machine.

The second form does not generate intermediate files and does a compilation, followed by an execution cycle, in one go.

You would usually use the first form to prepare your code for a release. The second form is the one that is preferred when compiling and running code from the command line or Visual Studio Code.

For now, let's get the code to compile and run at the same time, using the second form, with the `-run` flag.

```
$ dmd -run hello.d
hello D!
```

The hello message is displayed on the standard output, and our program is now courteous.

While we are it, let's set up Visual Studio Code as well, by installing the D language plug-in, shown in [Figure 5-2](#).

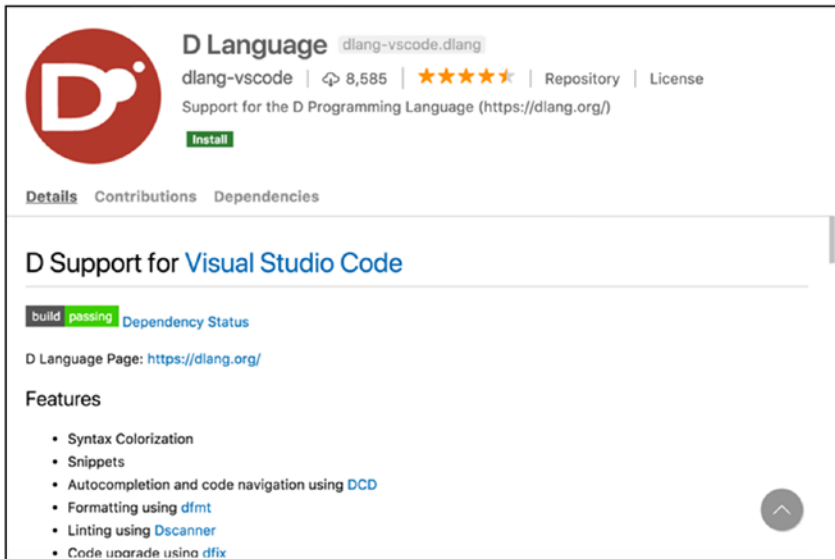


Figure 5-2. Plug-in for D language

Along with the plug-in, you need a `tasks.json` file, to run the `build` command from inside the editor. As in the preceding code, we will be calling the `dmd` compiler with the `-run` flag, so that the code is executed directly.

```
{
  "version": "2.0.0",
  "tasks":
    [
      {
        "label": "dmd",
        "command": "dmd",
        "args": [
          "-run",
          "${file}"
        ],
      },
    ],
}
```

```

    "options": {
      "cwd": "${workspaceRoot}"
    },
    "group": {
      "kind": "build",
      "isDefault": true
    }
  }
]
}

```

Everything is in place for some happy coding. One more Command(Ctrl)+Shift+B to test our build setup from the editor (Figure 5-3), and we're good to go.

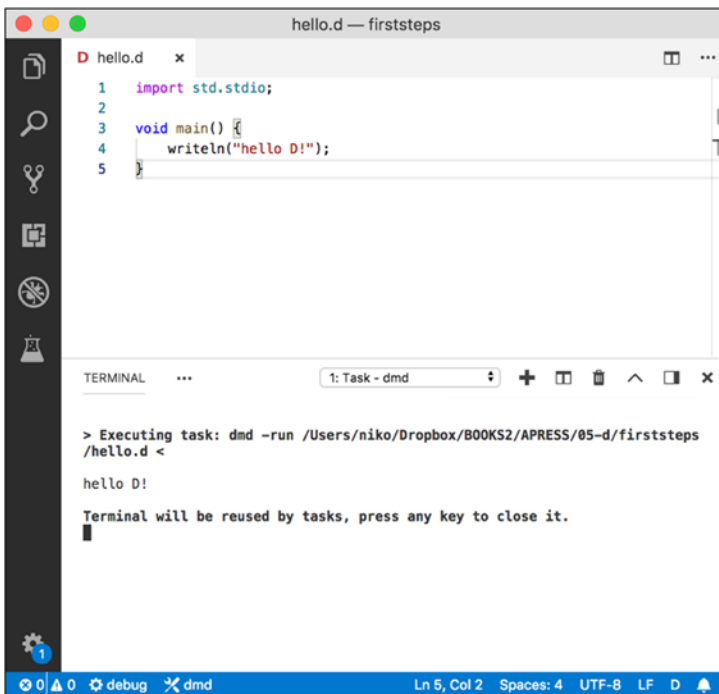


Figure 5-3. Running D source files directly from Visual Studio Code

Some Bits of D on Concurrency

Coming from the annoyance of Java with threading, you'll be thrilled by the concurrency features of D. Threads in D can be created and *spawned*, just as with regular D functions.

Simple Threading

The namespace responsible for concurrency is appropriately named `std.concurrency`, and we will use it for this first threaded test.

To spawn a thread from a function, we will simply use `spawn`, which takes a reference to a function as a parameter. This is done using an ampersand (&) in front of any standard D function. The rest of the code snippet is quite easy to follow, so here is the program with threads:

```
import std.concurrency : spawn, thisTid;
import std.stdio : writeln;

void worker()
{
    writeln("Thread ", thisTid);
}

void main()
{
    int TOTAL_WORKERS=5;
    for (int i = 0; i < TOTAL_WORKERS; ++i) {
        spawn(&worker);
    }
}
```

Executing from Visual Studio Code gives an output similar to the one following, in which each thread simply prints its own id.

```
> Executing task: dmd -run APRESS/05-d/firststeps/threads1.d <
Thread Tid(109655000)
Thread Tid(109655200)
Thread Tid(109655300)
Thread Tid(109655400)
Thread Tid(109655500)
```

From this short example, there are a few things worth noting.

- Every thread has a separate and uniquely assigned id, labeled `thisTid`.
- The function passed to `spawn` is actually a reference to the function, using `&worker`.

Next, let's see how to keep an internal state in each thread.

Thread with a State

In this second example, we define a static field inside the function that will be spawned for the thread. The defined `threadState` integer is stored locally to the thread as `ThreadLocal`. Also note that the integer `threadState` is created only once. Every subsequent time that the thread is calling itself via `worker()`, the thread state is not reinitialized.

Here is the code snippet with a local state for a thread:

```
import std.concurrency : spawn, thisTid;
import std.stdio : writeln;

void worker()
{
    static int threadState = 0;
```

```

        writeln("Thread ", thisTid,": My state = ", threadState++);
        if (threadState < 5) worker();
    }

void main()
{
    for (int i = 0; i < 5; ++i) {
        spawn(&worker);
    }
}

```

And here is the slightly shortened output:

```

> Executing task: dmd -run APRESS/05-d/firststeps/threads1.d <

Thread Tid(109a2d000): My state = 0
Thread Tid(109a2d000): My state = 1
...
Thread Tid(109a2d300): My state = 3
Thread Tid(109a2d500): My state = 2
Thread Tid(109a2d400): My state = 3
Thread Tid(109a2d300): My state = 4
Thread Tid(109a2d500): My state = 3
Thread Tid(109a2d400): My state = 4
Thread Tid(109a2d500): My state = 4

```

You can see that each thread has a separate state, and that state is incremented each time the thread is looping, by calling workers again.

Now, let's see how we can share a state between all those threads and perform some safe multi-threaded sharing of data.

Shared State

The last example in our thread subsection for the D language closes in shared state, with a common counter that is increased by each thread separately. To create such a shared piece of data, the D language requires that you specifically mark the data as shared.

Once this is done, you can use operations from the `core.atomic` namespace, to safely update the value of that shared piece of data. The atomic operation we will use is a combination of `atomicOp` and `+=`, written as follows:

```
atomicOp!"+="
```

This creates a thread-safe version of the function `+=`.

Let's see how this is actually used, in the following snippet.

```
import std.concurrency : spawn, thisTid;
import std.stdio : writeln;
import core.atomic : atomicOp;

shared int flag = 0;

void worker()
{
    atomicOp!"+="(flag, 1);
    writeln("Thread ", thisTid,":global>",flag);
}

void main()
{
    for (int i = 0; i < 20; ++i) {
        spawn(&worker);
    }
}
```


`atomicOp!"+="` receives two parameters, the shared data variable to update and the parameter to the atomic function `+=` just created.

When you run the preceding program, the output should be something like the following. Note the safeness of the threaded code.

```
> Executing task: dmd -run APRESS/05-d/firststeps/threads3.d <
Thread Tid(10e05c000):global>1
Thread Tid(10e05c200):global>2
Thread Tid(10e05c300):global>3
Thread Tid(10e05c400):global>4
Thread Tid(10e05c500):global>5
```

Obviously, the D language has full-featured threading capabilities not exposed in the simple examples in this chapter, but they should be easy enough, so that you can perform multi-threaded tasks with them.

Now let's sidestep threaded code and return to some simpler single-threaded examples.

A Few More Examples of D

First, let's go back to school for the basics, with examples of sorting.

Sort Me Tender, Sort Me True

Sorting arrays in D is just a matter of using the `sort` function from the `std.algorithm` namespace. Also, in the following snippet, note that the first use of `writeln`, which stands for "write format," allows you to turn various data structures and types into strings for printing out.

```
import std.stdio;
import std.algorithm;
```

```
void main()
{
    int[] arr1 = [4, 9, 7];
    writeln("%s\n", sort(arr1));
}
```

Executing the sorting preceding code surprisingly (!) gives you a sorted array.

```
> Executing task: dmd -run APRESS/05-d/firststeps/simplesort0.d
<
[4, 7, 9]
```

Sorting in reverse order can be done by passing the sorting function directly as a parameter to `sort`.

```
import std.stdio;
import std.algorithm;

void main()
{
    int[] arr1 = [4, 9, 7];
    writeln("%s\n", arr1.sort!("a > b"));
}
```

After execution, the reversed array is displayed in the output.

```
> Executing task: dmd -run APRESS/05-d/firststeps/simplesort0.d
<
[9, 7, 4]
```

I also stumbled on this great example, which must definitely be shared when presenting D to a crowd. The following code uses something named the chain function from the `std.range` namespace to sort a series of three different arrays, each time reusing the memory allocated for sorting.

```

import std.stdio;
import std.algorithm;
import std.range;

void main()
{
    int[] arr1 = [4, 9, 7];
    int[] arr2 = [5, 2, 1, 10];
    int[] arr3 = [6, 8, 3];
    sort(chain(arr1, arr2, arr3));
    writeln("%s\n%s\n%s\n", arr1, arr2, arr3);
}

```

After execution, while the output is as expected, the memory consumption should be quite considerably less when sorting big arrays.

```

> Executing task: dmd -run APRESS/05-d/firststeps/simplesort1.d
<

```

```

[1, 2, 3]
[4, 5, 6, 7]
[8, 9, 10]

```

When sorting `arr1`, `arr2`, and `arr3`, a program can sort in place, which takes less memory, but is slower, or use a temporary memory space. Using `chain`, you can tell the compiler to reuse that temporary memory to sort subsequent arrays, thus offering the advantage of both memory usage and CPU speed.

Last, before moving on to bot programming, let's see examples of implementing Fibonacci in D.

My Love for Fibonacci

In the never-ending search for the most effective Fibonacci code snippet, we may have something of a showstopper with the D language. This first snippet is the usual iterative implementation, in which the algorithm loops through the different value and iteratively computes the elements of the Fibonacci sequence.

```
auto fib(const int n)
{
    import std.bigint;
    if (n == 0)
        return BigInt(0);
    if (n == 1)
        return BigInt(1);
    BigInt next;
    for (BigInt i = 0, j = 1, count = 1; count < n; ++count)
    {
        next = i + j;
        i = j;
        j = next;
    }
    return next;
}

void main()
{
    import std.stdio;
    writeln(fib(100_000));
}
```

When you execute the preceding, you'll find that it's very fast. But the next implementation (found in the forums, implemented by Biotronic) is just blazingly fast.

```
auto fib(ulong n) {
    import std.bigint : BigInt;
    import std.meta : AliasSeq;
    import std.typecons : tuple;
    BigInt a = 0;
    BigInt b = 1;
    auto bit = 63;
    while (bit > 0) {
        AliasSeq!(a, b) = tuple(
            a * (2*b - a),
            a*a + b*b);
        if (n & (BigInt(1) << bit)) {
            AliasSeq!(a, b) = tuple(b, a+b);
        }
        --bit;
    }
    return a;
}
```

This version of generating the Fibonacci numbers uses the following:

- A different base algorithm to generate the Fibonacci numbers, named the Fast squaring algorithm
- AliasSeq, from the std.meta namespace, working with templates, which allows for compile-time improvement, with the help of the compiler.

Without going into all the details of the algorithm, this version clocks incredibly fast, even on a standard MacBook. You will appreciate the following times for `fib(1_000_000)`:

2 secs, 415 ms, 140 μ s, and 4 hnsecs

And 10 million clocks in at less than 50 seconds on the same machine!

But enough with displays of speed performance. Let's get a project template ready for some Telegram bot fun with D.

Telegram Bots in D

To write a telegram bot in D, we will require the usual build tool that can handle dependencies. In D, this tool is named `dub`, and I am sure you love the tool name as much as I do.

`dub` was installed (or should have been installed) along with D, earlier in the chapter.

`dub` is responsible for

- Handling your project metadata needs
- Managing dependencies
- Compiling and running code

Meet dub

`dub init <projectname>` will almost act as a bot and ask you a few questions, in order to generate files for your new project.

Note here that we are going to announce directly during project creation that we need `telega` as a dependency. `telega` is a wrapper for the Telegram bot API in D.

```
$ dub init hellobot
Package recipe format (sdl/json) [json]: json
Name [hellobot]:
Description [A minimal D application.]:
Author name [niko]:
License [proprietary]:
Copyright string [Copyright © 2018, niko]:
Add dependency (leave empty to skip) []: telega
Added dependency telega ~>0.0.3
Add dependency (leave empty to skip) []:
Successfully created an empty project in 'hellobot'.
Package successfully created in hellobot
```

dub will then have created your basic project structure, which at this stage contains only the project metadata in `dub.json` and the main source file in `app.d`, as shown following:

```
$ tree
.
├── dub.json
└── source
    └── app.d
```

1 directory, 2 files

Inspecting the `dub.json` file, you can see that it contains the required project metadata and the specified `telega` dependency, as well as the most recent version found.

```
{
  "name": "hellobot",
  "authors": [
    "niko"
  ],
```

```

"dependencies": {
  "telega": "~>0.0.3"
},
"description": "A minimal D application.",
"copyright": "Copyright © 2018, niko",
"license": "proprietary"
}

```

Caution That would almost work—except in those instances in which it doesn’t—and at the time of writing, it did not. You may have to refer to a specific build of *telega* (or any other dependency, for that matter).

Installing a custom version of a dependency is done in two steps.

1. Checking out the package you want, in this case, *telega*

```
git clone https://github.com/nexor/telega.git
```

2. Telling *dub*, with the following snippet, that you have a new version of that package:

```
[niko@niko-pc hellobot]$ dub add-local ../../RESEARCH/telega
```

```
Registered package: telega (version:
0.0.3+commit.6.g87f37af)
```

The project should then be good to go. You can validate that the dependencies are installed properly, using `dub list`.


```
$ dub list | grep telega
telega 0.0.3+commit.6.g87f37af: APRESS/RESEARCH/
telega/
telega 0.0.3: /Users/niko/.dub/packages/telega-0.0.3/
telega/
```

Of course, if you do not receive an error message, there's no need for all of this. The default subcommand of dub is `dub run`, and this tells your project to run the code located in the `source/app.d` file, with the dependency resolved from `dub.json`.

Right after creating your project, the default D source file should look like this:

```
import std.stdio;

void main()
{
    writeln("Edit source/app.d to start your project.");
}
```

Then you can execute the preceding default D file within the project with

```
dub run
```

That will pick up the file `source/app.d` inside your project folder. Finally, your output for your project is ready!

```
hellobot ~master: building configuration "application"...
Linking...
```

To force a rebuild of up-to-date targets, run again with `--force`.

```
Running ./hellobot
```

```
Edit source/app.d to start your project.
```

As usual, you can run all this from Visual Studio Code, with a simple `tasks.json` file. The file needed to run the bot project with `dub` from Visual Studio Code is shown following:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "dub",
      "command": "dub",
      "args": [
        "run"
      ],
      "options": {
        "cwd": "${workspaceRoot}"
      },
      "group": {
        "kind": "build",
        "isDefault": true
      }
    }
  ]
}
```

Note that finding other packages using `dub` is as easy as running `dubsearch`. So, for Telegram, we simply looked for packages using the following command:

```
$ dub search telegram
```

```
==== registry at https://code.dlang.org/ (fallback ["registry
at http://code.dlang.org/", "registry at https://code-mirror.
```

```

dlang.io/", "registry at https://code-mirror2.dlang.io/",
"registry at https://dub-registry.herokuapp.com/"] ) ===
paper_plane_bot (0.0.5) Telegram Dlang news notify bot
telega (0.0.3)           Telegram Bot API implementation
tg-d (0.0.2)           Telegram Bot API client library for the
                        D programming language
delegram (0.1.3)       dlang Telegram bot api
dtd (0.0.1)           A D wrapper to TDLib (Telegram Database
                        library)

```

In this case, `telega`, was the most updated package, with a wide range of the full Telegram bot API already implemented. You also can try to find packages for logging, for example.

```

$ dub search log | grep color
colored-logger (0.1.0)           Colored logger for TTY

```

Now on to our first D bot!

First D Bot

This first bot will perform the usual echo of a chat message.

A few things to note here:

- The main import that you haven't seen before is the one from the `telega` library, the `botapi`. We create a bot client with the key contained in the `secret.key` text file.
- Updates come in batches of messages, so in the `listenUpdates` function, we create a loop to handle those messages one by one.
- To tell the bot API we finished handling the update, we call `updateProcessed`, with the original update as parameter.

Cutting out the extra code from the telega example gives the following snippet:

```
import std.typecons;
import std.file;
import std.stdio;
import telega.botapi;

int main()
{
    auto botToken = readText("secret.key");
    listenUpdates(botToken);
    return 0;
}

void handleUpdate(BotApi api, Update msg) {
    if (!msg.message.isNull) {
        writeln("Update %s:%s", msg.message.chat.id, msg.
            message.text);
        api.sendMessage(update.message.chat.id, msg.message.text);
    }
    api.updateProcessed(update);
}

void listenUpdates(string token)
{
    auto api = new BotApi(token);
    while(true) {
        auto updates = api.getUpdates();
        writeln("Got %d updates", updates.length);
        foreach (update; updates) {
            handleUpdate(api, update);
        }
    }
}
```

Running the bot with the usual Visual Studio Code setup and `dub run` will obviously leave you breathless (see Figure 5-4).



Figure 5-4. *Echo text messages bot in D*

More Bot API Usage

For something more fun, you can now replace the insides of the `handleUpdate` function.

The rewritten function following gives you some examples of

- Sending a message
- Sending a photo
- Sending a document
- Sending a location

Note, too, that while we are replying to the same chat room, using the `update.message.chat.id`, we could also reply to a different chat room, keeping records of the id we have seen so far.

Also note that most of the implementation of `telega` uses public URLs to retrieve objects and does not yet allow you to send in local files.

```
void handleUpdate(BotApi api, Update update) {
    if (!update.message.isNull) {
        api.sendMessage(update.message.chat.id, update.message.text);

        api.sendPhoto(update.message.chat.id, "https://pbs.twimg.
            com/tweet_video_thumb/Dm_YlIgXgAAbxZu.jpg");
    }
}
```

```

api.sendDocument(update.message.chat.id, "https://ibiblio.
org/ebooks/Poe/Black_Cat.pdf");

api.sendLocation(update.message.chat.id, 45.8992, 6.1294);
}
api.updateProcessed(update);
}

```

Rerunning the bot with the new `handleUpdate` function gives you a series of message, image, document, and location, as shown in Figure 5-5.

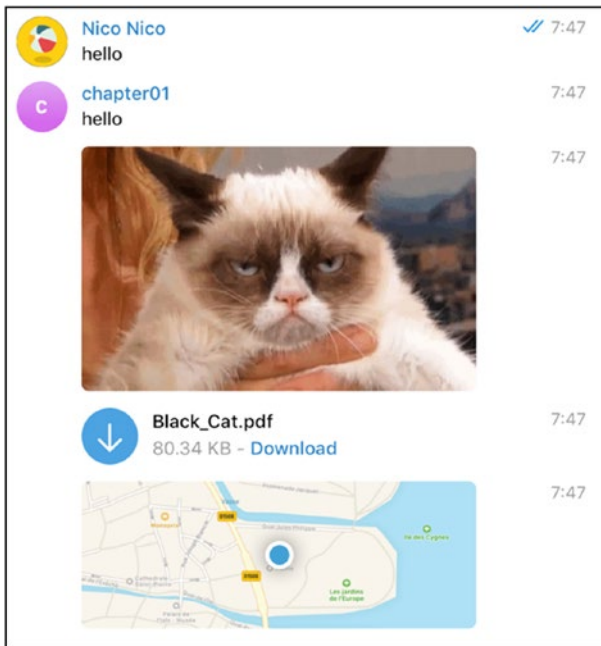


Figure 5-5. Message, image, document, and location from your D bot

For reference, the current set of methods implemented by telega is listed on the project site, but you may want to quickly put into action the functions presented following.

- `sendMessage`
- `forwardMessage`
- `sendPhoto`
- `sendAudio`
- `sendDocument`
- `sendVideo`
- `sendVoice`
- `sendVideoNote`
- `sendMediaGroup`
- `sendLocation`
- `editMessageLiveLocation`
- `stopMessageLiveLocation`
- `sendVenue`
- `sendContact`
- `setChatPhoto`

As for this book, it's now time to forge ahead to next week.

CHAPTER 6

Week 6: C++

Writing in C or C++ is like running a chainsaw with all the safety guards removed.

—Bob Gray

It was inevitable to have a chapter on C++, especially after dealing with D. What is surprising, however, is that there is a working Telegram bot API—wrapper—on the market, and what’s even more surprising is that it works very well.

C++ is the language often thought of as a reference to what not to code, but I think it’s always a plus to have it in your toolbox.

This chapter will focus on sending pictures from the usual echo bot to finally applying basic OpenCV calls on received images. OpenCV, if you are not familiar with this framework, is a universally known imaging library that can do everything from change colors and sizes to find objects in pictures and videos.

Requirements, Installation, and First Bot

Working with C++ usually requires a fair bit of work to set up. To work with Telegram, we will be using `tgbot-cpp`, and to work with `opencv`, you’ll also be installing the imaging framework on your local machine.

cmake is a low-level tool used to organize your project metadata and third-party libraries, as well as generating a file that makes the build tool itself.

boost is a set of well-maintained and clean libraries for C++.

Install tgbot-cpp

Cloning the tgbot-cpp repository is going to be the first step in this chapter. The code of the C++ wrapper for Telegram can be found on GitHub and following.

```
git clone https://github.com/reo7sp/tgbot-cpp.git
```

To compile and install the library, you need a few tools, such as make, cmake, ssl development libraries, and boost.

On Debian/Ubuntu, this would be

```
apt-get install g++ make binutilscmakelibssl-dev libboost-  
system-dev
```

Then, once in the tgbot-cpp cloned directory, the bot library can be easily installed with this set of three commands:

```
cmake .  
make  
sudo make install
```

The install defaults to /usr/lib/, and you can check that the Telegram library is properly installed, by looking up the presence of the following file:

```
/usr/local/lib/libTgBot.a
```

Install OpenCV

The imaging library OpenCV is a lot easier to install these days, and while options are still a bit hard to figure out, we will leave them out for this week and keep the default settings to compile and install the library. It's really just a matter of following the installation instructions from the OpenCV web site. For reference, here are the steps for Linux and macOS:

```
# clone the opencv repository
git clone https://github.com/opencv/opencv.git
# create build folder
mkdir build && cd build
# generate make file from CMake directives
cmake .
# compile code ... takes ~15min the first time.
make -j8
# install as library
sudo make install
```

Remember where you have installed opencv and its build folder, as you will need it later, when compiling C++ programs.

To verify our setup and get excited a bit, let's start by writing a simple program that downloads a file, using the system command `curl`.

File Download Program

We will be reusing this piece of code later in the OpenCV bot, so make sure you understand how things work. To write a project using `cmake`, we first create a simple folder structure, with the `CmakeLists.txt` file, and a source folder, with the C++ code in that source folder.

```
[niko@niko-pcsamplecmake]$ tree
```

```
.
├── CMakeLists.txt
└── src
    └── main.cpp
```

1 directory, 2 files

The CMakeLists.txt contains your project metadata, and main.cpp, the C++ code. The following CMakeLists.txt file is a simplified version of the one we will use to run tgbot-cpp later on. Let's have a look.

```
cmake_minimum_required(VERSION 2.8.4)

project(samplecmake)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -Wall")
set(Boost_USE_MULTITHREADED ON)

find_package(Threads REQUIRED)
find_package(OpenSSL REQUIRED)
find_package(Boost COMPONENTS system REQUIRED)
find_package(CURL)

include_directories(/usr/local/include ${OPENSSL_INCLUDE_DIR}
${Boost_INCLUDE_DIR})

if (CURL_FOUND)
include_directories(${CURL_INCLUDE_DIRS})
add_definitions(-DHAVE_CURL)
endif()

add_executable(samplecmakesrc/main.cpp)

target_link_libraries(samplecmake ${CMAKE_THREAD_LIBS_INIT}
${OPENSSL_LIBRARIES} ${Boost_LIBRARIES} ${CURL_LIBRARIES})
```

In Table 6-1, I list the commands used in the cmake definition files, in addition to why they are used.

Table 6-1. *Understanding cmake file*

Command	Use
cmake_minimum_required	Defines which version of cmake to use
project	Defines the project name
set	Sets variables used during the build
find_package	Finds external libraries, i.e., finds cmake files in search paths and retrieves symbols
include_directories	More folders to retrieve header files and packages
add_executable	Creates an executable, here, samplecmake, from a given source file
target_link_libraries	Identifies libraries to link to the given executable, here again, samplecmake

With this CMakeLists.txt, you can run the cmake command and prepare the project for compilation. Here is the shortened output of the command execution:

```
$ cmake .

-- The C compiler identification is GNU 8.2.1
-- The CXX compiler identification is GNU 8.2.1
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
...
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
```

```
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Found OpenSSL: /usr/lib/libcrypto.so (found version
  "1.1.0i")
-- Boost version: 1.67.0
-- Found the following Boost libraries:
--   system
-- Found CURL: /usr/lib/libcurl.so (found version "7.61.1")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/niko/Dropbox/BOOKS2/
  APRESS/06-cplusplus/samplecmake
```

If you are writing code from Visual Studio Code again, there is a specific and dedicated plug-in, created by Microsoft, that you can install (Figure 6-1).

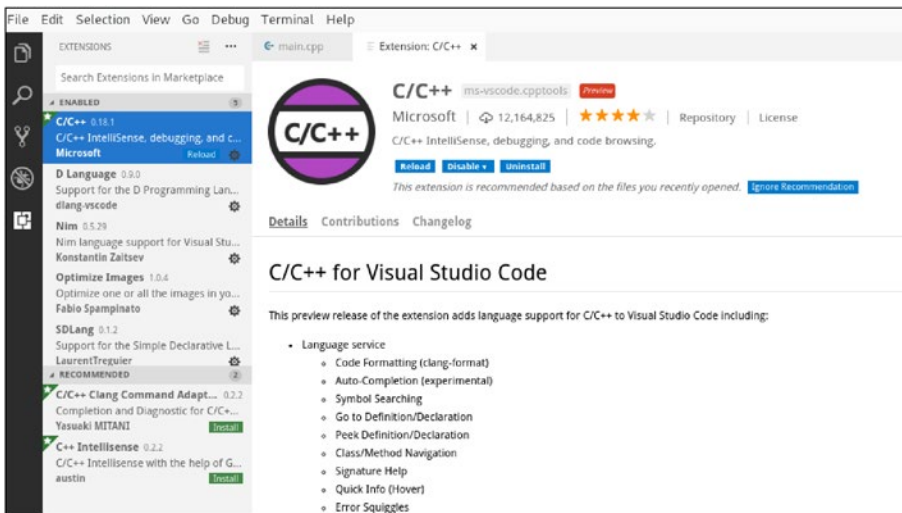


Figure 6-1. C/C++ plug-in for Visual Studio Code

With the plug-in installed, you can start typing C++ code and your first C++ program of the day.

Here is a list of reminders before proceeding to look at the code:

- `#include` is used to include namespaces
- Using namespace will prevent you from having to insert the namespace in front of each symbol of given namespaces, thus, `std::string` will become `string`, for example.
- `intmain(intargc, char* argv[])` is the main entry point, and you can retrieve the number of args, as well as an array of parameters. Note that the first parameter is at index 0 and is the command name itself. Here, it will be `samplecmake`, as it was defined in the project metadata file, `CMakeLists.txt`.
- `auto url` means that we do not specify the type of the variable ourselves; the compiler does it. Here, it will actually be a `char*`, which is a pointer to an array of characters.
- We will actually download the file using a system command, `curl` (which should be installed on your machine).
- The shell command `curl` is called using the system function, with `url` as the parameter to the `curl` command.

This now gives the slightly readable code snippet following, which should be saved in a file named `main.cpp`.

```
#include <string>
using namespace std;
```

```
int main(int argc, char* argv[])
{
    auto url = argv[1];
    printf("Downloading: %s\n", url);
    string command = string("curl --silent -O ");
    const char* cmd = command.append(url).c_str();
    system(cmd);
    return 0;
}
```

To create the executable, cmake has generated a Makefile, a file that the make command understands. Running make will create an executable.

make

To test our program, we can pass a URL with an image (or any file, for that matter) for our newly compiled and linked program.

```
./samplecmake http://stuffpoint.com/cats/image/245258-cats-cute-white-cat.jpg
```

This will download the cat shown in Figure 6-2.

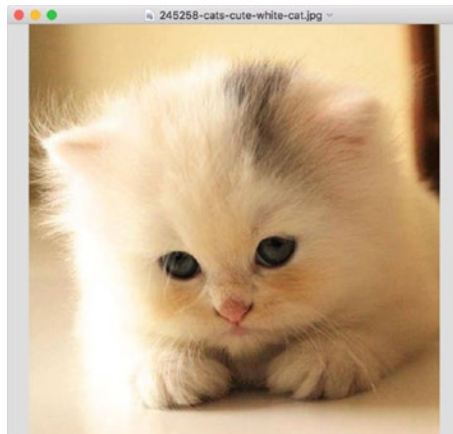


Figure 6-2. *Lovely cat*

Go ahead and try it with a few other URLs and see the files being downloaded in the project folder, as expected.

Now on to our first bot!

Echo Bot

The echo bot will, of course, echo whatever we send it. We will build on the download file project, so you can either reuse it or create a new folder with the same file and structure.

In the new `CMakeLists.txt`, you'll immediately notice three things:

- The project name has been changed.
- The name of the executable has been changed to `echobot`.
- The `target_link_libraries` section contains a reference to our installed `tgbot-cpp` library, which is packaged inside `/usr/local/lib/libTgBot.a`.

The rest is identical and should hold few surprises.

```
cmake_minimum_required(VERSION 2.8.4)
project(echobot)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -Wall")
set(Boost_USE_MULTITHREADED ON)

find_package(Threads REQUIRED)
find_package(OpenSSL REQUIRED)
find_package(Boost COMPONENTS system REQUIRED)
find_package(CURL)
include_directories(/usr/local/include ${OPENSSL_INCLUDE_DIR}
    ${Boost_INCLUDE_DIR})
if (CURL_FOUND)
```



```
include_directories(${CURL_INCLUDE_DIRS})
add_definitions(-DHAVE_CURL)
endif()

add_executable(echobotsrc/main.cpp)

target_link_libraries(echobot /usr/local/lib/libTgBot.a
    ${CMAKE_THREAD_LIBS_INIT} ${OPENSSL_LIBRARIES}
    ${Boost_LIBRARIES} ${CURL_LIBRARIES})
```

Here again, let's run the cmake command

```
cmake .
```

to generate the necessary project files.

Now, on to the code. The forthcoming code is mostly taken from the tgbot-cpp project samples, but there are a few things to notice.

- The bot token is read from an environment variable.
- You can register as many callbacks as you want, using `.getEvents.onCommand` or `.getEvents.onAnyMessage`. Here, the bot is defined to respond to a command named `start` and on any message sent.
- On each callback, you must specify a vector of pointers that will be used in the callback, hence, `[&bot]`. We will add some more later.
- On each callback, you get a pointer to the message, which has the same structure as you have seen previously.
- Finally, the bot is set up to do polling, as usual, but it has to be made explicit with this C++ library.
- And the snippet for the echo bot is now shown following:

```

#include <tgbot/tgbot.h>

using namespace std;
using namespace TgBot;

int main() {
    string token(getenv("TOKEN"));
    printf("Token: %s\n", token.c_str());

    Bot bot(token);

    bot.getEvents().onCommand("start", [&bot](Message::Ptr message)
    {
        bot.getApi().sendMessage(message->chat->id, "Hi!");
    });
    bot.getEvents().onAnyMessage([&bot](Message::Ptr message) {
        printf("User wrote %s\n", message->text.c_str());
        if (StringTools::startsWith(message->text, "/start")) {
            return;
        }
        bot.getApi().sendMessage(message->chat->id, "Your message is: "
        + message->text);
    });

    signal(SIGINT, [](int s) {
        printf("SIGINT got\n");
        exit(0);
    });

    try {
        printf("Bot username: %s\n", bot.getApi().getMe()->username.
        c_str());
        bot.getApi().deleteWebhook();
    }

```

```

TgLongPoll(longPoll(bot);
    while (true) {
printf("Long poll started\n");
longPoll.start();
    }
    } catch (exception& e) {
printf("error: %s\n", e.what());
    }
    return 0;
}

```

The same steps are used to compile and run. Before running the newly created executable, you must expose the bot token as `TOKEN` in the current shell.

```

export TOKEN=...
./echobot

```

On start, the bot, will yield a short output, to say it has started polling normally.

```

Token: ...
Bot username: chapter01bot
Long poll started

```

A sample chat session with this new bot is shown in [Figure 6-3](#).

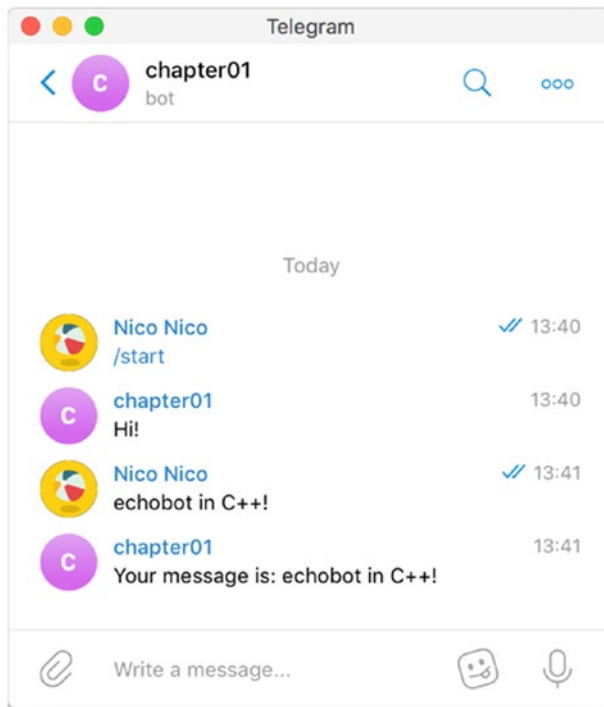


Figure 6-3. *C++echobot*

C++ Bots

Now on our mission to deploy as many Telegram bots as possible, we'll create two bots.

- One bot with a Telegram inline keyboard
- Another bot that sends pictures to the chat

Bot with Inline Keyboard

This second bot is also taken from the samples of the project and is added here for reference. Inline keyboards are pretty much buttons sent to the chat room that are assigned callbacks when pressed by the user,

just like a bot father. If you have started a project from scratch, there is no need to add anything new to `CMakeLists.txt`, apart from changing the project's name.

What is interesting in the bot here is the way the keyboard is registered. The rest of the code is identical to that for the echo bot and is left out for clarity.

The keyboard is created using `InlineKeyboardMarkup`, and then rows of buttons are created with the `InlineKeyboardButton` constructor. Each button has a callback, here, `check`, that must be registered in a `onCallbackQuery` block, just like commands and messages.

Finally, you send the keyboard to the chat, using the longer version of `sendMessage`, which accepts the created keyboard as parameter.

```
bot.getApi().sendMessage(chatid,response,false,0,keyboard,"Mark
down");
```

This gives the following snippet, which should be located inside the main method of your program:

```
// Thanks Pietro Falessi for code

InlineKeyboardMarkup::Ptr keyboard(new InlineKeyboardMarkup);

    vector<InlineKeyboardButton::Ptr> row0;
InlineKeyboardButton::Ptr checkButton(new InlineKeyboardButton);
checkButton->text = "check";
checkButton->callbackData = "check";
    row0.push_back(checkButton);

    keyboard->inlineKeyboard.push_back(row0);

bot.getEvents().onCommand("check", [&bot, &keyboard]
(Message::Ptr message) {
    string response = "ok";
```

```

bot.getApi().sendMessage(message->chat->id, response, false, 0,
keyboard, "Markdown");
});
bot.getEvents().onCallbackQuery([&bot, &keyboard]
(CallbackQuery::Ptr query) {
    if (StringTools::startsWith(query->data, "check")) {
        string response = "ok";
bot.getApi().sendMessage(query->message->chat->id, response,
false, 0, keyboard, "Markdown");
    }
});

```

Now let's compile and see this new bot in action (Figure 6-4).

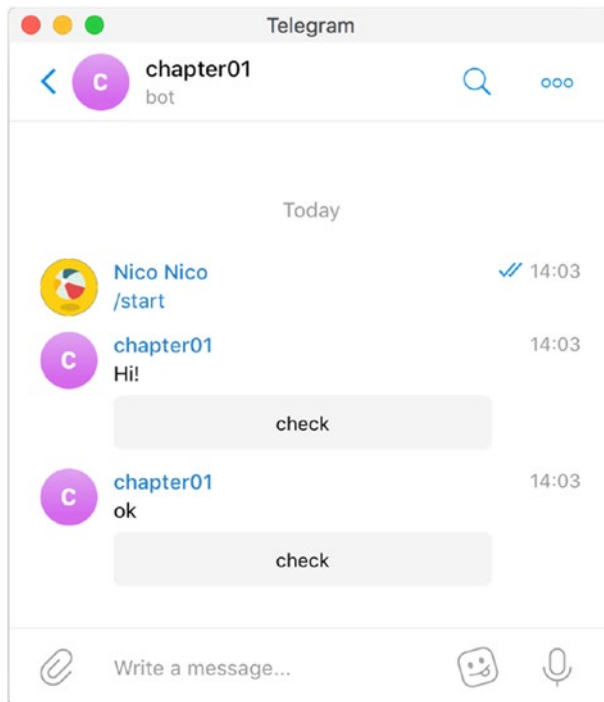


Figure 6-4. Bot with inline keyboard

Photo Bot

The photo bot is here to present how to send pictures to the chat, using the C++ API. The Telegram API being the same for any language, you'll notice how the method name is similar to ones used in other languages.

Here, we send an example.jpg image, located at the root of the project folder.

```
const string photoFilePath = "example.jpg";
const string photoMimeType = "image/jpeg";
    bot
    .getEvents()
    .onCommand("photo", [&bot, &photoFilePath, &photoMimeType]
(Message::Ptr message) {
    bot.getApi().sendPhoto(message->chat->id, InputFile::fromFile(p
hotoFilePath, photoMimeType));
    });
```

No extra change in the libraries is required, so just make and run will do here too. You can see the photo bot in action in [Figure 6-5](#).

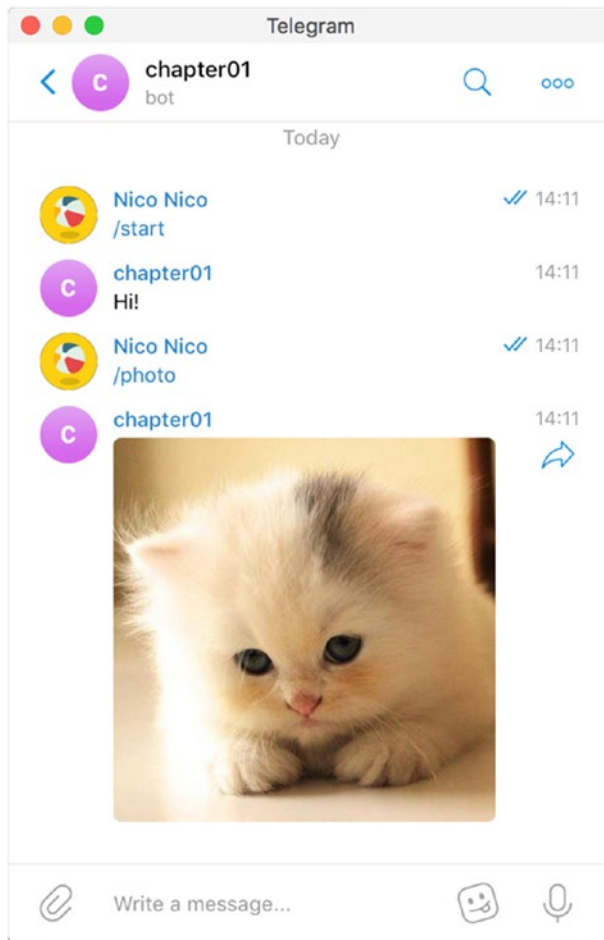


Figure 6-5. *Send me a cat, c++ bot*

Now let's repeat this technique of sending pictures from the photo bot, combined with the OpenCV example, to perform transformations on the pictures sent to the chat and resend them directly to the cat.

OpenCV in action

Before creating a bot, let's first try to perform a simple OpenCV transformation.

OpenCV Sample Program

This time, you will have to update the `CMakeLists.txt` a bit. Notably, you will have to include the place where the OpenCV header and library files are located and also specify to include them when linking the final binary.

So, building on the previous `CMakeLists.txt`, the following will occur:

- Change the project name! Although not required, it is always useful for us humans.
- Add the `opencv/build/include` folder to the `include_directories`.
- Add a `find_package` directive targeting OpenCV (be careful of the upper-/lowercase).
- In the `target_link_libraries`, add the different required `.so` files from the installed version of OpenCV. (The libraries could also be pulled from the `opencv/build/lib` folder, of course.) Here, three library files are used for OpenCV: `core`, `imgcodecs`, and `highgui`. In the future, you may need others, if you do some video, so you should adjust this accordingly.

Before running `cmake` on this new project, let's review the `CMakeLists.txt` file.

```
cmake_minimum_required(VERSION 2.8.4)
project(opencvdemo)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -Wall")
set(Boost_USE_MULTITHREADED ON)
```

```

include_directories($HOME/projects/opencv/build/include )
include_directories(/usr/local/include ${OPENSSL_INCLUDE_DIR}
${Boost_INCLUDE_DIR})

if (CURL_FOUND)
include_directories(${CURL_INCLUDE_DIRS})
add_definitions(-DHAVE_CURL)
endif()

find_package(Threads REQUIRED)
find_package(OpenSSL REQUIRED)
find_package(Boost COMPONENTS system REQUIRED)
find_package(CURL)
find_package(OpenCV)

add_executable(opencvdemo src/main.cpp)

target_link_libraries(opencvdemo /usr/local/lib64/libopencv_
core.so /usr/local/lib64/libopencv_imgcodecs.so /usr/local/
lib64/libopencv_highgui.so /usr/local/lib/libTgBot.a ${CMAKE_
THREAD_LIBS_INIT} ${OPENSSL_LIBRARIES} ${Boost_LIBRARIES}
${CURL_LIBRARIES})

```

Once cmake is run, we can focus on writing just a small piece of OpenCV programming code that will process a picture specified as input and transform it into its gray version. In OpenCV, this is done through an intermediate Mat object, which is a matrix object used to represent the image in-memory.

You'll need a new include statement, with opencv2/highgui, and, yes, we all love the fact that even though its opencv4, the namespace is still versioned as 2.

`imread` and `imwrite` are the two `opencv` methods to read and write pictures, and the second parameter of the `imread` tells it how many colors to use when decoding the picture. It gives the very short snippet following:

```
#include "opencv2/highgui.hpp"
using namespace cv;

int main(int argc, char* argv[])
{
    auto bw = imread(argv[1],0);
    imwrite("saved.jpg", bw);
    return 0;
}
```

Now to compile and run this short program using `make`.

```
make
./opencvdemo cat.jpg
```

The input picture, `cat.jpg`, is shown in Figure 6-6.



Figure 6-6. *Colored input cat*

After applying the OpenCV change of color, the gray version of the cat is shown in Figure 6-7.



Figure 6-7. *If you can see a difference, you are reading this book in colors*

Now that we know how to use `opencv` from our program, let's integrate all the pieces, to create a bot that transforms the picture sent to the chat.

OpenCV Bot

The `cmake` setup for the full OpenCV bot is the same as the previous one, so there's no need to change the `CMakeLists.txt`, apart from the project name. The thing to know ahead of programming this bot is that when the chat receives a message with a picture, the message contains a `fileId` corresponding to the file that was saved.

On the Telegram architecture, those files are hosted on different servers, thus a different API. The new location is `api.telegramorg/file/bot`.

The URL to retrieve files is then constructed from this base URL, appended with the bot token and, finally, the `fileId`.

To read a file and process the picture or file, we must perform a different request to retrieve that file. So, in this example, you will reuse the trick of running the `curl` command from the C++ code to perform this request.

The following function, `applyOpenCV`, downloads a static file from the Telegram server applies the `opencv` process of turning the picture to shades of gray, and, finally, saves it to a file named `saved.jpg`.

```
const string telegram_url = "https://api.telegram.org/file/bot";
const string tmp_file = "download.jpg";
const string saved_file = "saved.jpg";

string applyOpenCV(string token, string path) {
    string command = string("curl --silent");
    command
.append(telegram_url)
.append(token)
.append("/")
.append(path)
.append(" -o ")
.append(tmp_file);

    const char* cmd = command.c_str();
    system(cmd);

    Mat bw = imread(tmp_file,0);
    imwrite(saved_file, bw);
    return saved_file;
}
```

Now, we just need an entry point to call this function. This will be called when a photo is detected in the chat.

So, in the bot callback handling the chat message, to detect whether a picture is present, we can check the number of photo sizes included in the message. Once we know there is a picture (or pictures), we can retrieve its (their) `fileId` and then call the `applyOpenCV` function we have just defined with the `filePath` and the token.

Also, note, as I previously touched on, the vector of the reference is being added a new pointer on token. If you forget to do this, the callback does not have access to the token variable, and the compilation fails.

```
bot.getEvents().onAnyMessage([&bot, &token](Message::Ptr
message) {
    if(message->photo.size() != 0) {
        PhotoSize::Ptr s = message->photo[2];
        if(s!=NULL) {
            string fileId = message->photo[2]->fileId;
            File::Ptr file = bot.getApi().getFile(fileId);
            string filepath = applyOpenCV(token.c_str(), file-
            >filePath.c_str());
            bot.getApi().sendPhoto(
                message->chat->id,
                InputFile::fromFile(filepath, "image/jpeg"));
        }
    }
}
```

The rest of the bot code is identical to that for the other bots. After executing, make and starting the bot, you can see the picture being sent to the chat change to black and white, as shown in Figure 6-8.

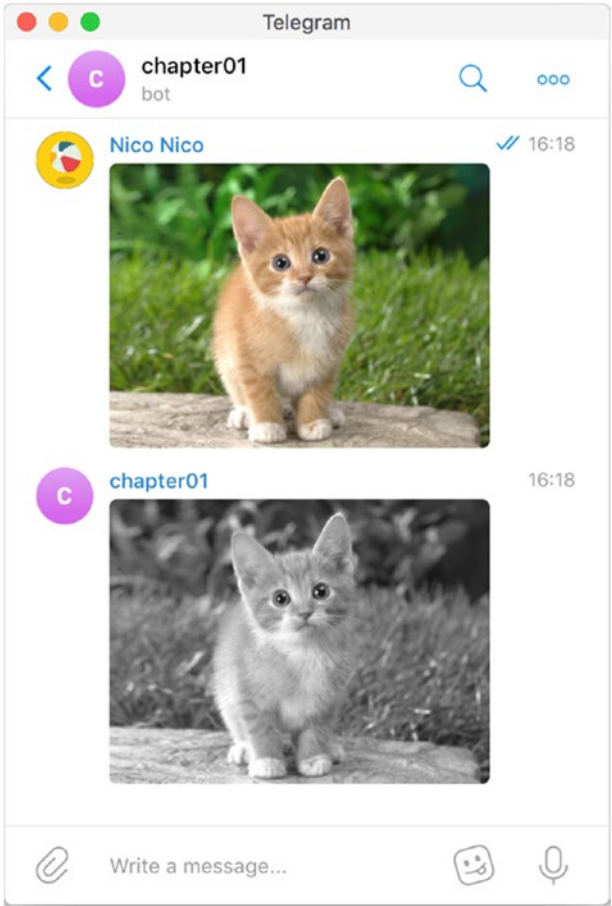


Figure 6-8. *Colored and gray cats*

Obviously, the next step is to try a few more OpenCV transformations and feed them into your bot. Creating a bot to identify specific objects in each picture is now within your reach.

CHAPTER 7

Week 7: Clojure

Sometimes a thing needed opening before closure was found.

—Hugh Howey

Shift

Clojure is the only LISP-based language presented in this book, but it's a language I use on an everyday basis. Clojure, with its share of left and right parentheses, can repel a few, but it surely never gets lost in translation.

Clojure development almost always uses a read-eval-print-loop (REPL) or, in simple terms, a shell that understands Clojure code line by line. Of course, in an editor, you can just execute block of lines of code, and I will show you how to do that in Visual Studio Code again.

Clojure makes it easy to understand the data structure passing via the Telegram server, so we will look at a few of the JSON updates coming from Telegram. To write Telegram bots in the Clojure language, we will be using another Telegram wrapper named Morse, which makes it dead easy to set up your own custom bot, by having a template project ready for you.

Initial Setup and First Clojure Bot

Clojure is running on top of Java Virtual Machine, so if you do not have it installed already, head to <http://jdk.java.net>, and install a version of openjdk suitable for your machine. On Linux machines, almost every single package manager has a version of openjdk. For example, on Manjaro/Arch Linux, you can go with `yaourt -S java-openjdk-ea-bin`.

As far as setups tested for this book, JDK versions 8 to 11-ea made the cut.

Now that you have the Java Compiler, you don't need much more than Leiningen, the de facto build tool for Clojure. Installation instructions are short and available from the web site at <https://leiningen.org/>.

The main task is to install a package or a shell script that downloads and bootstraps Leiningen for you. On Manjaro/Arch, here it is:

```
yaourt -S leiningen
```

If the setup is good to go, you should now have Leiningen responding to you on the command line (Figure 7-1).

```

17 [Running]
Terminal - niko@niko-pc:~
File Edit View Terminal Tabs Help
[niko@niko-pc ~]$ lein
Leiningen is a tool for working with Clojure projects.

Several tasks are available:
change      Rewrite project.clj with f applied to the value at key-or-path.
check       Check syntax and warn on reflection.
classpath   Write the classpath of the current project to output-file.
clean       Removes all files from paths in clean-targets for a project
compile     Compile Clojure source into .class files.
deploy      Deploy jar and pom to remote repository.
deps        Show details about dependencies.
do          Higher-order task to perform other tasks in succession.
help        Display a list of tasks or help for a given task or subtask.
install     Install jar and pom to the local repository; typically ~/.m2.
jar         Package up all the project's files into a jar file.
javac       Compile Java source files.
new         Generate scaffolding for a new project based on a template.
plugin      DEPRECATED. Please use the :user profile instead.
pom         Write a pom.xml file to disk for Maven interoperability.
release     Perform release tasks.
repl        Start a repl session either with the current project or standalone.
retest      Run only the test namespaces which failed last time around.
run         Run the project's -main function.
search      Search Central and Clojars for published artifacts.
show-profiles List all available profiles or display one if given an argument.
test        Run the project's tests.
trampoline  Run a task without nesting the project's JVM inside Leiningen's.
uberjar     Package up the project files and all dependencies into a jar file.
update-in   Perform arbitrary transformations on your project map.
upgrade     Upgrade Leiningen to specified version or latest stable.
vcs         Interact with the version control system.
version     Print version for Leiningen and the current JVM.
with-profile Apply the given task with the profile(s) specified.

Run 'lein help $TASK' for details.

Global Options:
-o          Run a task offline.
-U          Run a task after forcing update of snapshots.
-h, --help  Print this help or help for a specific task.
-v, --version Print Leiningen's version.

These aliases are available:
downgrade, expands to upgrade

```

Figure 7-1. Hello Leiningen

Leiningen has all you need to start coding with Clojure, notably the possibility of offering you an REPL, the shell to execute Clojure code. You start this REPL by using Leiningen with the `repl` subcommand.

```
lein repl
```

Once the REPL is ready, you can start typing code directly at the prompt.

```

[niko@niko-pc ~]$ lein repl
nREPL server started on port 46749 on host 127.0.0.1 -
nrepl://127.0.0.1:46749
REPL-y 0.3.7, nREPL 0.2.12
Clojure 1.8.0

```

OpenJDK 64-Bit Server VM 11+28

```
Docs: (doc function-name-here)
      (find-doc "part-of-name-here")
```

```
Source: (source function-name-here)
```

```
Javadoc: (javadoc java-object-or-class-here)
```

```
Exit: Control+D or (exit) or (quit)
```

```
Results: Stored in vars *1, *2, *3, an exception in *e
```

```
user=> (+ 1 1)
```

```
2
```

```
user=> (println "hello world")
```

```
hello world
```

```
nil
```

```
user=>
```

Apart from writing Clojure code directly, Leiningen can also generate a full project layout, by using Leiningen templates. The third-party library we want to use for communicating with Telegram, Morse, has a template ready to prepare a new bot. This is done using the following subcommand of Leiningen:

```
lein new morse mytelegrambot
```

This will have the effect of downloading all the Clojure dependencies and creates the project structure for the bot.

```
$ lein new morse mytelegrambot
```

```
Retrieving morse/lein-template/0.1.1/lein-template-0.1.1.pom
from clojars
```

```
Retrieving morse/lein-template/0.1.1/lein-template-0.1.1.jar
from clojars
```

```
Generating fresh 'lein new' morse project.
```

Once the project has been created, you can check the existence of the project files, namely:

- `project.clj`: The usual main project metadata. It contains the project name, compilation details, dependencies, etc.
- `core.clj`: The main source file for the project (unless specified otherwise in `project.clj`)
- `core_test.clj`: Where you can write your Clojure tests

The generated project tree structure follows:

```
$ tree
.
├── CHANGELOG.md
├── LICENSE
├── project.clj
├── README.md
├── resources
├── src
│   ├── mytelegrambot
│   │   └── core.clj
└── test
    ├── mytelegrambot
    │   └── core_test.clj
```

5 directories, 6 files

Your Clojure bot is ready to be started, this time, using the `run` subcommand of Leiningen.

This will execute the main function defined in the `core.clj` file, providing you with your Telegram token, which should make things work out of the box.

```
$ export TELEGRAM_TOKEN="585672177:..."  
$ lein run
```

Starting the mytelegrambot

With the basic setup in place, you can start chatting with your Clojure Telegram bot at once (Figure 7-2).

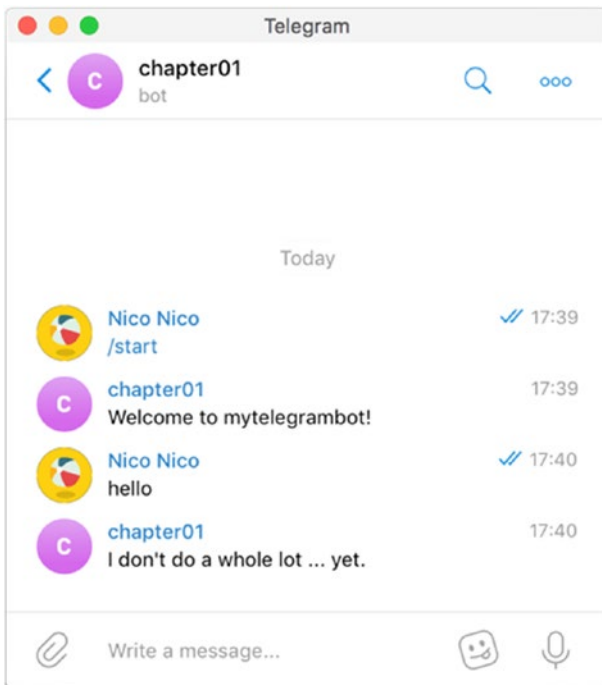


Figure 7-2. *It's been a long time*

In the console where you started the bot, you can also see the following debugging messages being printed on the standard output:

```
Bot joined new chat: {:id 121843071, :first_name Nico, :last_name Nico, :username hellonico, :type private}
```

```
Intercepted message: {:message_id 675, :from {:id 121843071,
:is_bot false, :first_name Nico, :last_name Nico, :username
hellonico, :language_code en-JP}, :chat {:id 121843071,
:first_name Nico, :last_name Nico, :username hellonico, :type
private}, :date 1537519211, :text hello}
```

Visual Studio Code

As a quick workflow to develop your bot, you can either start an REPL with `lein repl`, as we have seen, or use the Visual Studio Code plug-in shown in Figure 7-3.

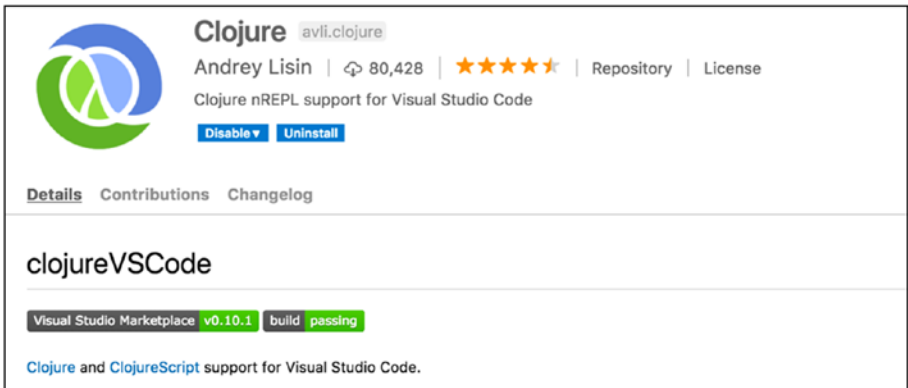


Figure 7-3. *Clojure plug-in for Visual Studio Code*

The next three things to do are

- Look at `project.clj`, the metadata file of the project.
- Look at the Clojure code to handle Telegram requests.
- Look at the messages coming from Telegram.

Look deep into nature, and then you will understand everything better.

—Albert Einstein

We are now going to look at all these, and in order mentioned.

The Project Metadata in project.clj

A good point with Leiningen and Clojure is that the project metadata is in the same language as the code itself, meaning the project configuration is in Clojure. It's actually more or less a big hash map. Consistency is a key (pun intended), and it's reassuring to find the file with that constant structure.

In `project.clj`, you start by defining a Clojure project with `defproject`, passing a project name (here, `mytelegrambot`), a version number, and a map of different things, in which each key is prefixed with `:`.

Among the different things used to define a project, we can find

- *Dependencies*: A list of third-party libraries to import and use in your project
- *Plug-ins*: A list of plug-ins for Leiningen
- `main`: The main file, actually namespace, to compile and or run

Just as with project names, dependencies come with a project name and a version number, and this is actually the same format used. The full `project.clj` file that has been generated is copied here.

```
(defproject mytelegrambot "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
```

```

:license {:name "Eclipse Public License"
          :url "http://www.eclipse.org/legal/epl-v10.html"}

:dependencies [[org.clojure/clojure "1.8.0"]
               [environ             "1.1.0"]
               [morse                "0.2.4"]]

:plugins [[lein-environ "1.1.0"]]

:main ^:skip-aot mytelegrambot.core
:target-path "target/%s"

:profiles {:uberjar {:aot :all}})

```

Now, let's have a look at the code itself, located in `core.clj`.

The Clojure Code in `core.clj`

Let's break this source file in smaller parts. First, Clojure project source files start with a namespace definition, `namedasns`, and then a list of other namespaces to use in the current context.

```

(ns mytelegrambot.core
  (:require [clojure.core.async :refer [<!!>]]
            [clojure.string :as str]
            [environ.core :refer [env]]
            [morse.handlers :as h]
            [morse.polling :as p]
            [morse.api :as t])
  (:gen-class))

```

You will notice namespace for async code, string manipulation, easy retrieval of environment variables, and different namespaces from Morse, the third-party library responsible for doing low-level communication with the Telegram API. Note, too, how each namespace also defines a prefix,

such as `p`, `t`, or `h`, which you can use as shortcuts, instead of using the full namespace name. Finally, `gen-class` is used to tell Clojure code that it has to turn itself into something the Java runtime understands natively, namely, a Java class.

Without knowing too much, you can feel that the following is going to retrieve the token from a Clojure environment variable that we will set in a later section. The library `environ` “keywordizes,” or makes Clojure-friendly, variables from the external shell environment. Here, `TELEGRAM_TOKEN` will be turned into `:telegram-token`.

```
; TODO: fill correct token
(def token (env :telegram-token))
```

I will hold off on the handler definition for a bit, but for now, know that this is where you will write the meat of your bot code.

```
(h/defhandler handler
...
)
```

Finally, `main`, as its name implies, is the main function for the program. It checks for the existence of a Telegram token, then starts to poll for updates, using the function `p/start`, with parameters `telegram token` and the `telegram handler`.

```
(defn -main
  [& args]
  (when (str/blank? token)
    (println "Please provide token in TELEGRAM_TOKEN environment
variable!")
    (System/exit 1))

  (println "Starting the mytelegrambot")
  (<!! (p/start token handler)))
```

I said that we were going to look at the handler code in a bit more detail later, and later is the new now. Reminder: Each command is handled via a `command-fn` directive, and all these are defined within the `defhandler` section of the file.

```
(h/defhandler handler
  ; your handlers here.
)
```

First, we define a simple "start" command for our bot. A command takes a Clojure function as callback, and the parameter from that function is (de-)constructed from the incoming message.

```
(h/command-fn "start"
  (fn [{id :id :as chat} :chat]
    (println "Bot joined new chat: " chat)
    (t/send-text token id "Welcome to mytelegrambot!"))))
```

Once the start command kicks in the chat, we simply send a message back to the chat with the `send-text` function from the `morse.api` namespace prefixed by `t` (as defined in the `require` section of the namespace).

The generated code also defines a `help` command, which is defined exactly in the same way and actually does almost exactly the same thing as the start command, sending back a text message.

```
(h/command-fn "help"
  (fn [{id :id :as chat} :chat]
    (println "Help was requested in " chat)
    (t/send-text token id "Help is on the way"))))
```

Last, we define a generic message handler, and we also get a shortcut directly on the message data structure in the callback function using the keyword `:as`.

```
(h/message-fn
  (fn [{:id :id} :chat :as message]
    (println "Intercepted message: " message)
    (t/send-text token id "I don't do a whole lot ...
yet.")))
```

The Token!

You may remember that there was a `lein-environ` plug-in defined in the `project.clj` file. This plug-in reads environment variables from different places, for example, `project.clj`, and then populates a file named `.lein-env` with all the necessary environment variables.

So, you can add in your `project.clj` (for one project), as follows:

```
(defproject mytelegrambot "0.1.0-SNAPSHOT"
  ...
  :env
  {:telegram-token "585672177:..."}))
```

or in `$HOME/.lein/profiles` (for multiple projects)

```
{:user {:env{:telegram-token "585672177:..."}}}
```

Debugging Telegram Messages

To have a look at messages as they are coming from Telegram, we will format and output them in a log file. To do this, we will create an appending write to the file. In the `core.clj` file, let's declare a write, right below the token definition.

```
(def writer (clojure.java.io/writer "message.log " :append
true))
```

And in the callback function receiving the message, let's output the received message.

Clojure does the format for you, using the `pprint` function from namespace `clojure.pprint`, and you can tell it where to write messages.

```
(clojure.pprint/pprint message writer)
```

On the next message sent to the bot, the `message.log` file is being filled with the incoming messages, as shown in Figure 7-4.

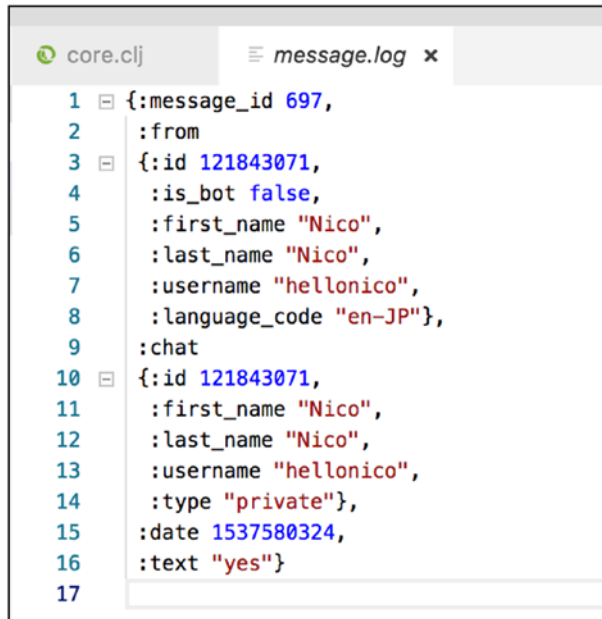


Figure 7-4. *Formatted incoming messages*

Creating a Reverse Bot

Let's create a text-reversing bot now. You will remember that you have access to the Clojure string namespace, with the `str/` prefix. This namespace has many functions, one of them being `reverse` (Figure 7-5).



Figure 7-5. *reverse function, shown alongside all the `str/` functions*

To create a handler for reversing text, you can call this `reverse` function on the received text. The received text itself, if you look at the map from Figure 7-4 again, can be taken from the message with the key `:text`.

Here is the new message handler that reverses text sent to the chat:

```
(h/message-fn
  (fn [{:id :id} :chat :as message]
    (clojure.pprint/pprint message writer)
    (t/send-text token id (str/reverse (:text message)))))
```

And now, this works nicely in the Telegram chat, in which you can try sending any kind of text (Figure 7-6).

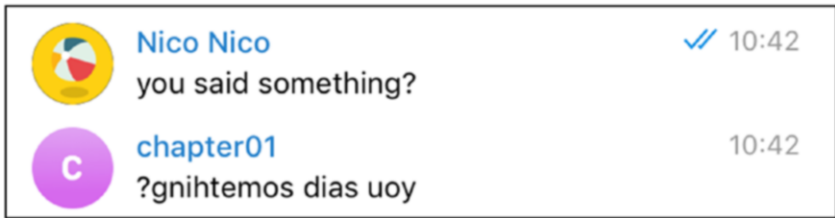


Figure 7-6. *Reversed text*

Inline Handler

You'll probably also remember how to define inline handlers from previous chapters. Following is a way to do it in Clojure with Morse.

This inline handler just logs the message, at first.

```
(h/inline-fn
  (fn [inline]
    (clojure.pprint/pprint inline writer)
    inline))
```

The content of the inline message itself is shown in Figure 7-7.

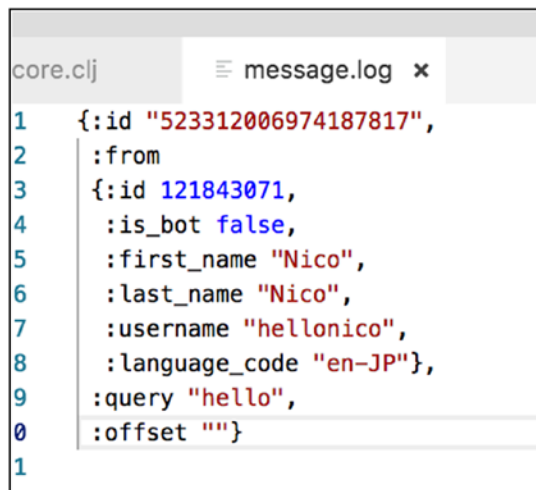


Figure 7-7. *Inline message content*

Note that to answer inline, the Morse library documentation is a bit lax. Looking at the Telegram documentation on inline results makes this slightly more explicit.

<https://core.telegram.org/bots/api#inlinequeryresultgif>

From the official documentation, you will notice that

- You cannot specify any other type than gif.
- `thumb_url` and `gif_url` are both required.

And so, to send inline answers, you can write code similar to the following snippet. Here, we always send the same gif.

```
(t/answer-inline
  token
  (:id inline)
  [{:type "gif"
    :id "gif1"
    :thumb_url "https://bit.ly/2DtXcIi"
    :gif_url "https://bit.ly/2DtXcIi"}])
```

Run the handler to find out which one!

A Simple Weather Bot

Next, we will have a bot to receive messages containing a location, and we will retrieve weather information for that location, by sending a request to OpenWeather (<https://openweathermap.org/>).

Obviously, you must register to receive an API token from the OpenWeather web site, as shown in Figure 7-8.

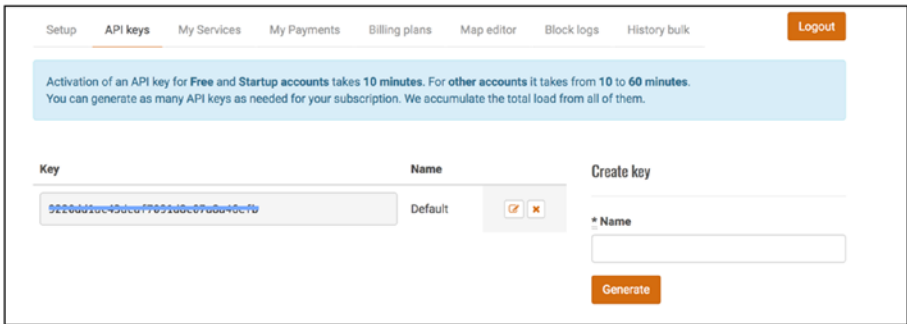


Figure 7-8. OpenWeather API token

Registration is free, and obtaining a token should only take a few minutes.

Once you have a token for OpenWeather, it is just a matter of sending an HTTP query similar to the one following:

<http://api.openweathermap.org/data/2.5/weather?q=<city>&units=metric&APPID=openweather-api-token>

If you try this on Tokyo with curl or httpie, you will get a JSON response similar to the one shown following:

```
{
  "base": "stations",
  "clouds": {
    "all": 75
  },
  "cod": 200,
  "coord": {
    "lat": 35.68,
    "lon": 139.76
  },
  "dt": 1537583700,
  "id": 1850147,
```



```
"main": {
  "humidity": 88,
  "pressure": 1009,
  "temp": 22.18,
  "temp_max": 24,
  "temp_min": 21
},
"name": "Tokyo",
"sys": {
  "country": "JP",
  "id": 7619,
  "message": 0.0056,
  "sunrise": 1537561714,
  "sunset": 1537605508,
  "type": 1
},
"visibility": 10000,
"weather": [
  {
    "description": "light intensity shower rain",
    "icon": "09d",
    "id": 520,
    "main": "Rain"
  }
],
"wind": {
  "deg": 340,
  "speed": 4.1
}
}
```

To do that in Clojure, we are going to

- Execute an HTTP request simply by using Clojure slurp, which retrieves the whole content of either a file or a URL
- Parse the slurped message using Clojure's JSON Cheshire library and its function parse-text, to generate a Clojure data structure
- Convert all this to a string to send back the message to the chat

The function to retrieve the weather is

```
(defn weather[city]
  (let [request
        (str
         "http://api.openweathermap.org/data/2.5/weather?q="
         city
         "&units=metric&APPID="
         openweather-api-token )]
    (:main
     (parse-string (slurp request)
                    (fn [k] (keyword k))))))
```

And the Morse/Telegram handler that can call it, retrieves the name of city from a chat message, and send the request using the weather function defined above.

```
(defhandler handler
  (message-fn
   (fn [{id :id} :chat :as message]
     (let [place (:text message)]
       (try
        (api/send-text token id
```

```
{:parse_mode "Markdown"}
(str "*" place "*" "\n" (weather place)))
(catch Exception e))))))
```

The result is shown in Figure 7-9.

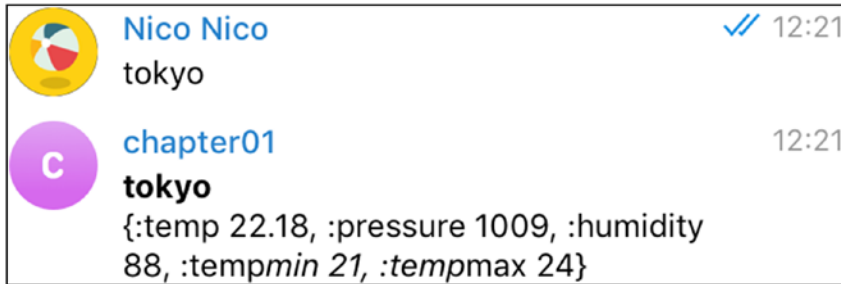


Figure 7-9. Send the name of the city, and receive the temperature

Sweet! Finally, let's move to an origami bot.

OpenCV and Telegram: Origami Bot

Origami is a Clojure wrapper around the OpenCV library. To set up anything on your machine, it usually helps just to download a wrapped opencv delivered through Clojure dependencies.

To do this, we will update slightly the `project.clj`, to retrieve the Origami library and to bootstrap the opencv environment.

```
(defproject origamibot "0.1.0-SNAPSHOT"
  :injections [
    (clojure.lang.RT/loadLibrary org.opencv.core.Core/NATIVE_
      LIBRARY_NAME)
  ]
```

```

:repositories [["vendredi" "https://repository.hellonico.
info/repository/hellonico/"]]
:main origamibot.core
:license {:name "Eclipse Public License"
          :url "http://www.eclipse.org/legal/epl-v10.html"}
:plugins [[lein-envirom "1.1.0"]]
:dependencies [
[envirom "1.1.0"]
[cheshire "5.6.1"]
[origami "0.1.11"]
[hellonico/morse "0.2.4"]
[org.clojure/clojure "1.8.0"]]

```

Now, to retrieve a picture, remember how to access the Telegram static files: from the file id retrieved in the chat message by a full message, when a picture is sent to the chat, as shown below.

```

{:message_id 851,
 :from
 {:id 121843071,
  :is_bot false,
  :first_name "Nico",
  :last_name "Nico",
  :username "hellonico",
  :language_code "en-JP"},
 :chat
 {:id 121843071,
  :first_name "Nico",
  :last_name "Nico",
  :username "hellonico",
  :type "private"},
 :date 1537587499,

```

```

:photo
[{:file_id "AgADBQADTqgxGxauIVWm22ogiY88fiZL1TIABM-
as8GGX14indYDAAEC",
 :file_size 1022,
 :width 90,
 :height 57}
 ; other files
]}

```

To download a file from Telegram, remember the official documentation: <https://core.telegram.org/bots/api#file>.

Also, remember the request to retrieve the file from the file path, https://api.telegram.org/file/bot<token>/<file_path>, where this file path is retrieved after calling `getFile` with the `file_id` contained from a message on a chat.

This is achieved by using the `api/download-file` of Morse (the custom version of the library, actually: `hellonico/morse`), which has been added to the project. `clj`. The custom version diff can be found online, and the reason to use it is to download a file to the local file system. This is just for convenience, and you could, of course, code it yourself, after a few days of practice with Clojure.

All this being in place, let's add the `origami` package to the namespace section as `origami`, the core wrapper for `opencv`.

```

(ns origamibot.core
  (:require
    [opencv3.core :as origami]
    ;...
    [clojure.string :as str])
  (:gen-class))

```

You can then apply any opencv transformation you want. Let's define a function named `apply-cv` that applies a canny effect to the picture, loaded from a file. Note that the transformation is done in place in the file.

```
(defn apply-cv [filename]
  (-> filename
    (origami/imread)
    (origami/cvt-color! origami/COLOR_RGB2GRAY)
    (origami/canny! 300.0 100.0 3 true)
    (origami/bitwise-not!)
    (origami/imwrite filename)))
```

Finally, you call the OpenCV transformation on the file retrieved from the chat, by calling the previously defined `apply-cv` function and the downloaded file.

```
(defhandler handler
  (message-fn
    (fn [{:id :id} :chat :as message])
    (let [fid (-> message :photo last :file_id)
          filename (str fid ".png")]
      (api/download-file token fid)
      (apply-cv filename)
      (api/send-photo token id (clojure.java.io/as-file
        filename))))))
```

The rest of the story is one more picture being uploaded in the chat. You can see it in action in Figure 7-10.

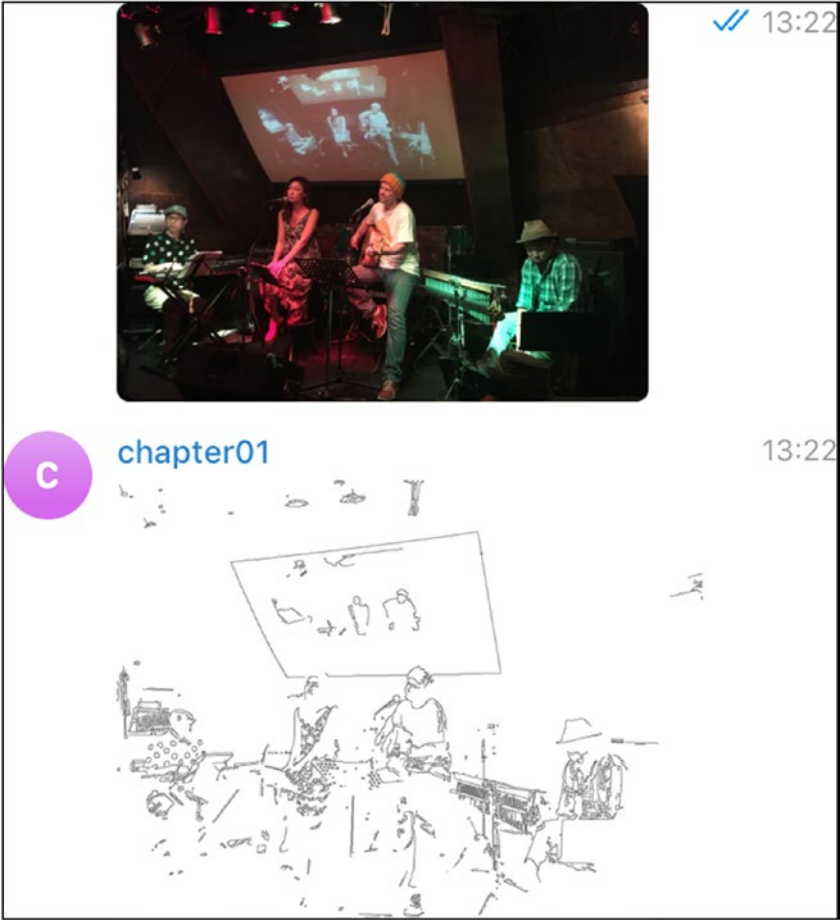


Figure 7-10. Applying the OpenCV transformation directly from the bot handler

CHAPTER 8

Week 8: Java

Coffee is a language in itself.

—Jackie Chan

In this chapter, we are going to tackle a Telegram bot in Java. The API in Java is not as bad as it looks, and debugging Java code in Visual Studio Code also works slightly better than expected.

The project will use the Java de facto build tool, Gradle. The library will be the `java-telegram-bot-api`.

Telegram is now proposing a payment API, so you can start selling stuff directly through Telegram. This is especially effective for selling services.

While the first part of this chapter will revisit the basics, we will then implement a bot that responds to the challenges of using the Telegram Payment API and create an example of the full payment life cycle.

Installation

Apart from Apache Maven, Gradle is the de facto build tool in Java land. It's actually the main build tool for building Android applications.

You can manually download and install Gradle on your machine, by downloading binaries available through the different package managers.

The Gradle web site (<https://gradle.org/install/>) has an extensive section on how to install the software.

sdkman is nice to use these days:

```
sdk install gradle 4.10.2
```

Homebrew is the standard on macOS.

```
brew install gradle
```

Chocolatey is the standard on Windows.

```
choco install gradle
```

Once installed, you can check whether you have the most recent version available, which, for Gradle, at the time of writing, was version 4.10.2

```
$ gradle -v
```

```
Welcome to Gradle 4.10.2!
```

Here are the highlights of this release:

- Incremental Java compilation by default
- Periodic Gradle caches cleanup
- Gradle Kotlin DSL 1.0-RC6
- Nested included builds
- SNAPSHOT plugin versions in the `plugins {}` block

For more details see <https://docs.gradle.org/4.10.2/release-notes.html>

```
-----  
Gradle 4.10.2  
-----
```

```
Build time: 2018-09-19 18:10:15 UTC
```

```
Revision: b4d8d5d170bb4ba516e88d7fe5647e2323d791dd
```

```
Kotlin DSL: 1.0-rc-6
```

```
Kotlin: 1.2.61
```

```

Groovy:      2.4.15
Ant:         Apache Ant(TM) version 1.9.11 compiled on
             March 23 2018
JVM:         1.8.0_171 (Oracle Corporation 25.171-b11)
OS:          Mac OS X 10.13.6 x86_64

```

The Project Structure

A Java project using Gradle is mostly made of the `build.gradle` file, which contains metadata and build information that Gradle can understand and source files located in `src/main/java` (by default).

```

.
├── build.gradle
├── resources
│   ├── cat.jpg
│   └── token
└── src
    ├── main
    │   └── java
    │       ├── com
    │       │   └── hellonico
    │       │       ├── Invoice.java
    │       │       └── MyMain.java

```

6 directories, 5 files

The build.gradle file

Gradle is quite versatile, and you can build pretty much anything with it. In our case, we are going to build a Java project, so we will use the Gradle plug-in for Java, with a few standard settings used for compilation, such as the file encoding and compilation compatibilities.

Dependencies are defined in the dependencies section, each of them identified by the following format:

```
<group>:<name>:<version>.
```

If you do not remember the dependency format, you can search and find any Java dependency on mvnrepository.com, as shown in Figure 8-1.

Home » [com.github.pengrad](#) » [java-telegram-bot-api](#) » 4.1.0

JavaTelegramBotApi » 4.1.0
Java API for Telegram Bot API

License	Apache 2.0
HomePage	https://github.com/pengrad/java-telegram-bot-api/
Date	(Aug 27, 2018)
Files	jar (210 KB) View All
Repositories	Central
Used By	4 artifacts

[Maven](#) [Gradle](#) [SBT](#) [Ivy](#) [Grape](#) [Leiningen](#) [Buildr](#)

```
// https://mvnrepository.com/artifact/com.github.pengrad/java-telegram-bot-api
compile group: 'com.github.pengrad', name: 'java-telegram-bot-api', version: '4.1.0'
```

☒ Include comment with link to declaration

Figure 8-1. Details for *java-telegram-bot-api* from *MvnRepository*

Here is the content of the `build.gradle` file:

```
apply plugin: 'java'

sourceCompatibility = 1.8
targetCompatibility = 1.8
```

```

compileJava {
    options.encoding = "UTF-8"
}

repositories {
    jcenter()
}

dependencies {
    compile 'com.github.pengrad:java-telegram-bot-api:4.1.0'
    compile 'com.sparkjava:spark-core:2.2'
    compile 'org.jsoup:jsoup:1.8.3'
    compile 'io.reactivex:rxjava:1.0.16'
}

apply plugin: 'application'
mainClassName = "com.hellonico.Simple"

```

The last two lines of the application plug-in and the `mainClassName` are not required, but they help, if you want to start your program simply by using

```
gradle run
```

For example, given the following simple Java class and program:

```

package com.hellonico;

public class Simple {
    public static void main(String[] args) {
        System.out.println("hello nico");
    }
}

```

if you execute the run command, the program will execute, and the following output will be shown in the terminal or command prompt:

```
$ gradle run

> Task :run
hello nico

BUILD SUCCESSFUL in 1s
2 actionable tasks: 2 executed
```

This will execute the main method of the `com.hellonico.Simple` class. More on the application plug-in can be found on the Gradle web site (https://docs.gradle.org/current/userguide/application_plugin.html).

Visual Studio Code Setup

Because I have suggested using Gradle as the build tool for the chapter on Java, Visual Studio Code can recognize the build tool and set up the project with a close-to-perfect integration, using Java tooling.

The main plug-in for Java is shown in Figure 8-2 and can be installed through the usual Visual Studio Code marketplace.

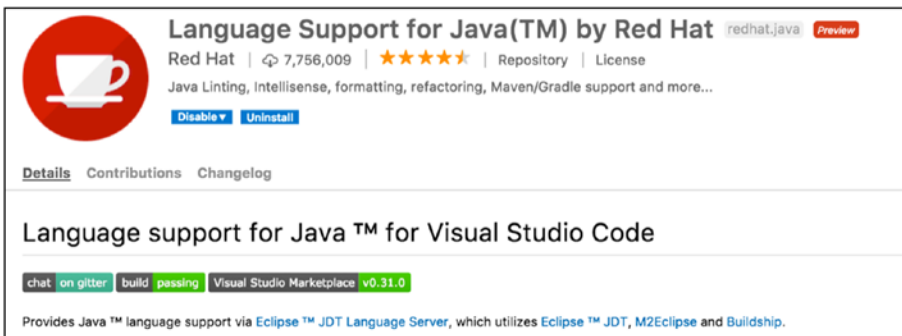


Figure 8-2. Visual Studio Code Java plug-in

The same simple Java class that was written above yields the result shown in Figure 8-3.

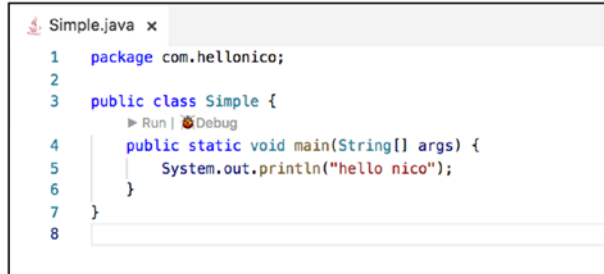


Figure 8-3. *hello nico Java program*

You now see two icons with which to run and debug your code. They will be useful when you write the Telegram bot.

To try it now, click Run, which executes the program and shows the proper output on the Visual Studio Code console (Figure 8-4).



Figure 8-4. *Running Java from Visual Studio Code*

First Java Bot

Our first Java bot will send some text and a photo, just to make sure the full Java setup is working. The bot will be initialized with the token loaded from a resources/token file, in which you should paste the token of your bot.

Along the way, you will probably find that the Java imports are a bit hard to find, but they can be auto-imported using Visual Studio Code Organize Imports, as shown in Figure 8-5.

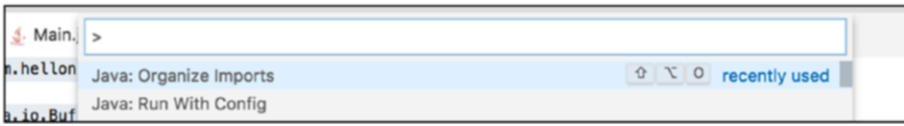


Figure 8-5. *Organize Imports*

Probably the most difficult part of the code for this first bot is the following line:

```
bot.setUpdatesListener(new UpdatesListener() {..})
```

This is where you tell the bot to poll and listen for updates. In its simplest form, the bot code is as follows:

```
package com.hellonico;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.List;
import java.util.stream.Collectors;

import com.pengrad.telegrambot.TelegramBot;
import com.pengrad.telegrambot.UpdatesListener;
import com.pengrad.telegrambot.model.Update;

public class Main {

    public static String streamFile_Buffer(String file) throws
Exception{
        BufferedReader reader =
new BufferedReader(new FileReader(file));
        return reader
.lines()
.collect(Collectors.joining(System.lineSeparator()));
    }
}
```

```

    public static void main(String[] args) throws Exception {
        TelegramBot bot =
new TelegramBot(streamFile_Buffer("resources/token"));
bot.setUpdatesListener(new UpdatesListener() {
@Override
    public int process(List<Update> updates) {
        System.out.println(updates.toString());
        // DO SOMETHING HERE.
        return UpdatesListener.CONFIRMED_UPDATES_ALL;
    }
});
    }
}

```

Send Some Text

Our first interaction will be to send some text to the chat. You can navigate through update messages from Telegram just as with other languages. You'll find the usual messages with the same structure seen to now.

```
[Update{update_id=573518674, message=Message{message_id=956,
from=User{id=121843071, is_bot=false, first_name='Nico', last_
name='Nico', username='hellonico', language_code='en-JP'} ,
date=1537602748, chat=Chat{id=121843071,..
```

From there, we can obtain the chat id

```
int id = updates.get(0).message().chat().id().intValue();
```

and send a message back to the chat room. Note that we are using bot.execute with a request and a callback, so the result is asynchronous.


```

SendMessage requestText =
new SendMessage(id, "*hello from java*").parseMode(ParseMode.
Markdown);
bot.execute(requestText, new Callback<SendMessage,
SendResponse>() {
@Override
public void onResponse(SendMessage request, SendResponse
response) {}
@Override
public void onFailure(SendMessage request, IOException e) {}
});

```

Before starting the bot, you must organize the imports or complete the list manually. The full list is shown following and should be located at the top of the source file.

```

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.List;
import java.util.stream.Collectors;
import com.pengrad.telegrambot.Callback;

import com.pengrad.telegrambot.TelegramBot;
import com.pengrad.telegrambot.UpdatesListener;
import com.pengrad.telegrambot.model.Update;
import com.pengrad.telegrambot.model.request.ParseMode;
import com.pengrad.telegrambot.request.SendMessage;
import com.pengrad.telegrambot.response.SendResponse;

```

Start the bot by clicking Run or Debug, which give some markdown formatted text in the chat (Figure 8-6).

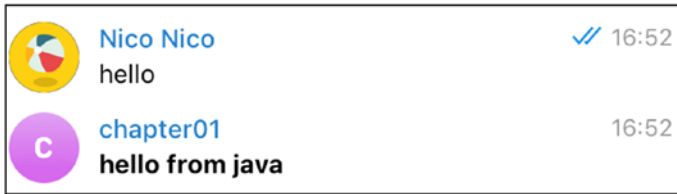


Figure 8-6. *Your Java bot is running!*

Send a Photo

Sending a photo is as easy as sending text, and this time, we will use the synchronous version of sending the request to Telegram. Because the code is not asynchronous anymore, failures are not handled in the `onFailure` callback that was available in the asynchronous version. So, it's a good idea to surround the request by a `try/catch` block.

```
SendPhoto requestPhoto = new SendPhoto(id, new File("resources/
cat.jpg"));
try {
    SendResponse response = bot.execute(requestPhoto);
} catch (Exception e) {
    e.printStackTrace();
}
```

And the cat picture is sent to the chat, as shown in [Figure 8-7](#).



Figure 8-7. *The cat is back!*

Now let's try to generate an invoice and do some payment with a Java bot.

Bot with Invoice Capabilities

Telegram has added a Payment API to accept payment directly from a chat room. This is a truly wonderful setup for sending money, and transactions in the services field will greatly benefit from its expansion.

Asking Permission

Before using these capabilities, you must obtain permission, again from the almighty BotFather, by changing the Payments settings of your bot (see [Figure 8-8](#)).

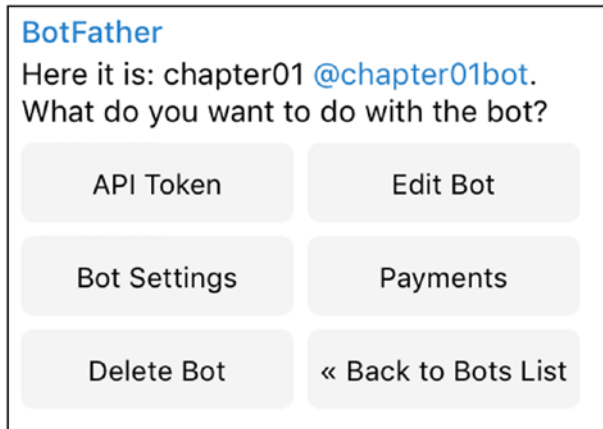


Figure 8-8. *Payment section in the bot settings of BotFather*

After payment settings for the bot have been updated, BotFather will give you a payment token, as shown in Figure 8-9.

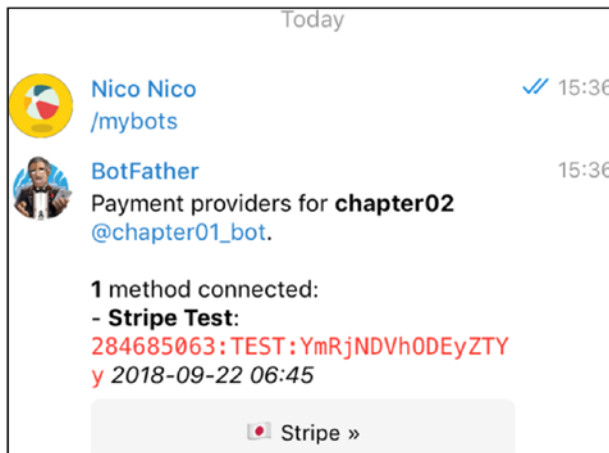


Figure 8-9. *Payment details have been updated*

Note here that another token is given to you. This is the token you will have to use when sending invoice messages.

Along the way, you will have to set up a payment account with your favorite provider. Here, we are using Stripe and testing its capabilities, as shown in Figures 8-10 and 8-11.

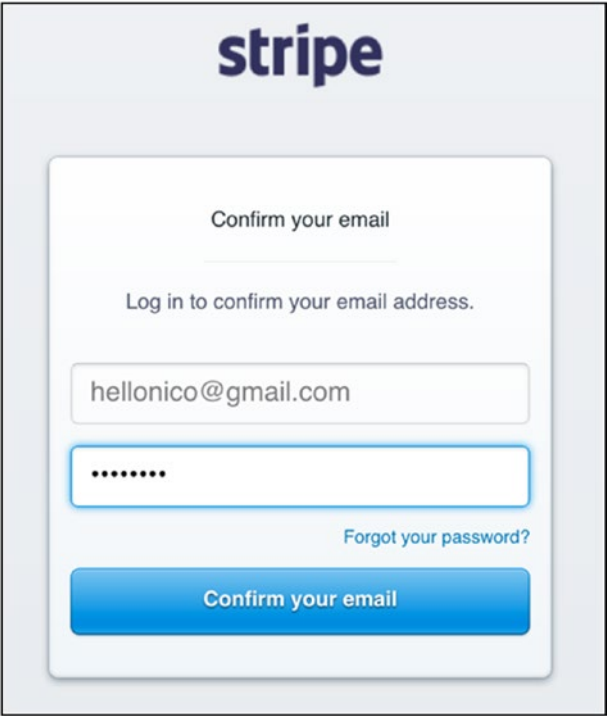


Figure 8-10. Stripe setup

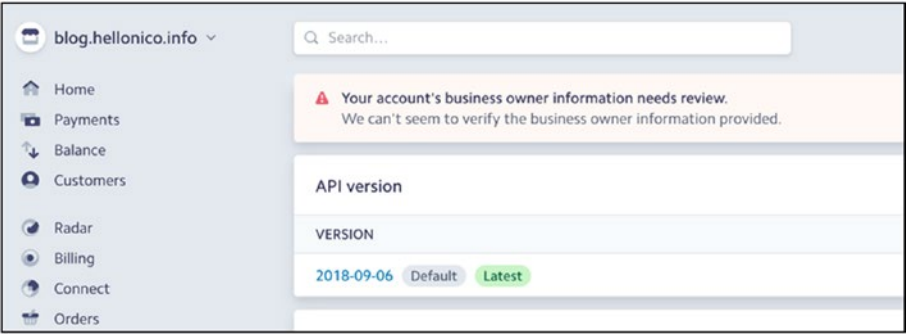


Figure 8-11. Stripe dashboard

If the setup has completed properly, you will get the confirmation from the Stripe Test Bot, as shown in Figure 8-12.

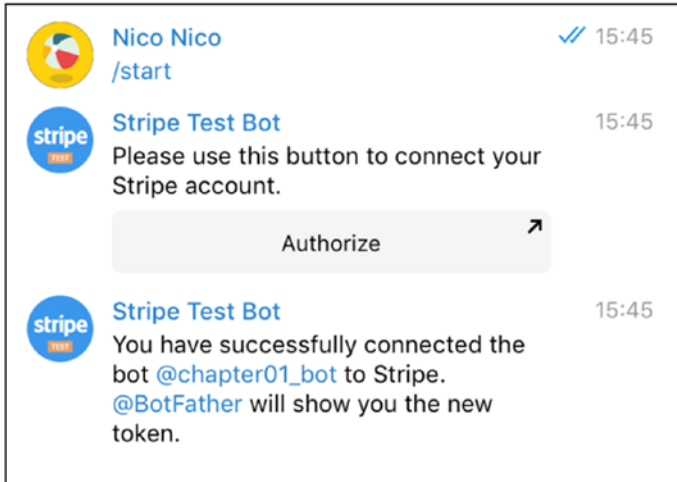


Figure 8-12. Stripe Test Bot connected

Now your bot can send invoice messages and attempt to get money from people talking to it.

Sending an Invoice Message

In the same vein as sending a photo or text, you can send invoices to the chat, using a `SendInvoice` message.

```
SendInvoice sendInvoice

= new SendInvoice(id, "Lemon", "desc", "hello", "2846850
63:TEST:NDBlMjliMGM2YmQ0", "my_start_param", "JPY", new
LabeledPrice("label", 2000))

.needsPhoneNumber(false)
.needsShippingAddress(false)
```

```
.isFlexible(true)
.replyMarkup(new InlineKeyboardMarkup(new InlineKeyboardButton[]
{ new InlineKeyboardButton("just pay").pay(),
new InlineKeyboardButton("google it").url("www.google.com") }));
```

Again, you can execute the query synchronously or asynchronously. It is recommended that you avoid blocking the execution on the main thread.

```
// sync version
SendResponse response = bot.execute(sendInvoice);

// async version
bot.execute(sendInvoice, new Callback<SendInvoice, SendResponse>() {
@Override
public void onResponse(SendInvoice request, SendResponse
response) {}
@Override
public void onFailure(SendInvoice request, IOException e) {}});
```

The payment can only be made using the mobile version of the Telegram app, to which users will also have to provide their shipping details (Figure 8-13).



Figure 8-13. Invoice me lemons

In this step, clicking just pay will get the payment bot to send a request for shipping query. (Actually, only if `.isFlexible (true)` has been set.)

At this stage, we can send a few options for the shipment and send them back to the chat.

```

ShippingQuery shipping= updates.get(0).shippingQuery();
if (shipping!= null) {
    ShippingOption option =
new ShippingOption("fedex", "FedEx", new LabeledPrice("JOY", 2000));
    AnswerShippingQuery query =
new AnswerShippingQuery(shipping.id(), option);
    bot.execute(query);
    return UpdatesListener.CONFIRMED_UPDATES_ALL;
}

```

The shipment options will then show up in the chat, as shown in Figure 8-14.

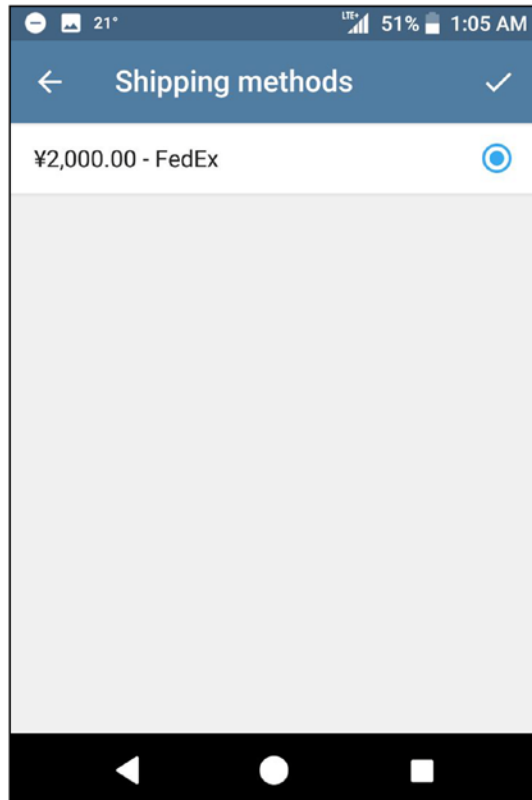


Figure 8-14. Shipping options

Figures 8-15 and 8-16 show the process for filling in the remaining info of the payment process.

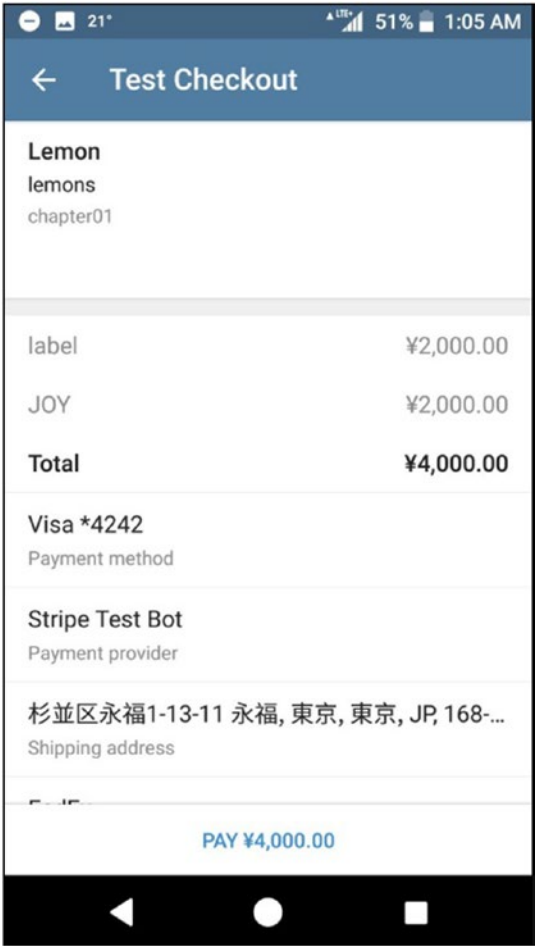


Figure 8-15. Full checkout screen

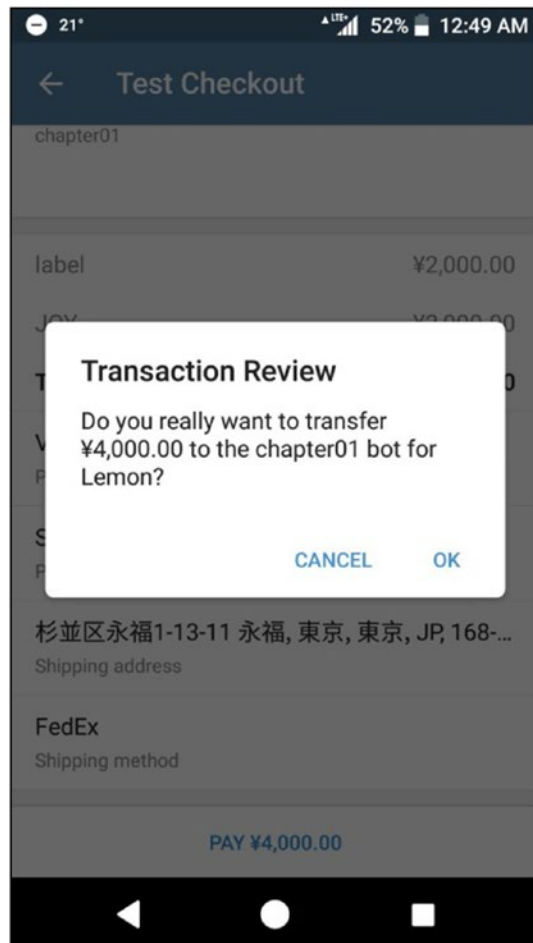


Figure 8-16. *Transaction validation*

Finally, the payment bot will send a request to your bot for pre-checkout, with all the payment details. You have to answer this request from the bot as fast as you can, and actually ten seconds is the max time limit, or the payment will be canceled.

Completion of the transaction is done by sending a `PreCheckoutQuery` with the query id, when a `preCheckoutQuery` element is received.

```

PreCheckoutQuery query = updates.get(0).preCheckoutQuery();
if (query != null) {
    AnswerPreCheckoutQuery apcq = new AnswerPreCheckoutQuery
        (query.id());

    bot.execute(apcq);
    return UpdatesListener.CONFIRMED_UPDATES_ALL;
}

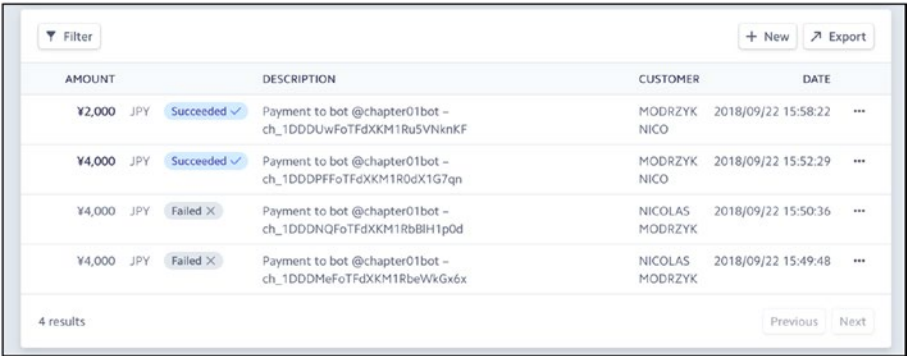
```

Figure 8-17 shows the chat after the payment has been completed.



Figure 8-17. *Payment complete*

On the Stripe dashboard, in the payment section, after enabling test data, you can see the different orders coming through, as shown in Figure 8-18.



Filter		+ New		Export	
AMOUNT			DESCRIPTION	CUSTOMER	DATE
¥2,000	JPY	Succeeded ✓	Payment to bot @chapter01bot - ch_1DDDUwFoTFdXKM1Ru5VNknKF	MODRZYK NICO	2018/09/22 15:58:22 ...
¥4,000	JPY	Succeeded ✓	Payment to bot @chapter01bot - ch_1DDDPFFoTFdXKM1R0dX1G7qn	MODRZYK NICO	2018/09/22 15:52:29 ...
¥4,000	JPY	Failed ✗	Payment to bot @chapter01bot - ch_1DDDNQFoTFdXKM1RbBIH1p0d	NICOLAS MODRZYK	2018/09/22 15:50:36 ...
¥4,000	JPY	Failed ✗	Payment to bot @chapter01bot - ch_1DDDMeFoTFdXKM1RbeWkGx6x	NICOLAS MODRZYK	2018/09/22 15:49:48 ...
4 results					
Previous Next					

Figure 8-18. Stripe test logs

That was quite smooth. That’s it for the full payment process life cycle in Java. Now it’s your turn to start selling lemons...and become rich!

CHAPTER 9

Week 9: Go

I have nine armchairs from which I can be critical.

—Rick Moranis

Go was originally created by people who were not too fond of the C++ programming language. Go is a strongly typed and compiled language, which aims mostly at being both easy to learn and fast to execute. Go is also one of the few languages to have a fantastic logo!

The logo, shown in Figure 9-1, was designed by [Renée French](#), and I understand that I can use it here, as she is credited.

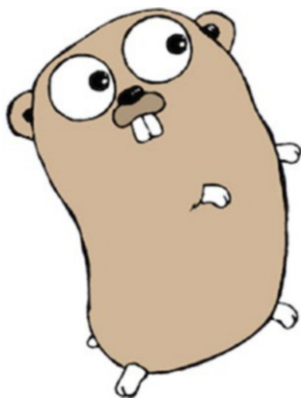


Figure 9-1. *Gopher, Renée French’s logo for the Go language*

In this chapter, I will review how to install the Go binary, followed by first steps and basic Go samples, before moving on to writing a Telegram bot and, finally, writing a command-line binary to send different Telegram objects via the API.

Installation of Go

To download Go, you can use the prepackaged version from the Go web site, located at <https://golang.org/dl/>. Most platforms have an option available for download, as shown in Figure 9-2.



Figure 9-2. Go packages for your preferred platform

Your favorite package manager should also have the Go package available.

```
# On Linux/Manjaro
-S go
# on macOS
brew install go
# on Windows
choco install golang
```

Once installed, you should check with the version subcommand whether you have a relatively current version.

```
# Current version
$ go version
go version go1.11 linux/amd64
```

It is usually recommended that you create a GOPATH variable, which is used so that your Go packages can be downloaded and stored in a known place.

```
export GOPATH=$HOME/go
```

Visual Studio Code has a plug-in for Go, and to install it, it is recommended that you follow this recipe. There are multiple plug-ins for Go, but the one from Microsoft is very solid and is shown in Figure 9-3.

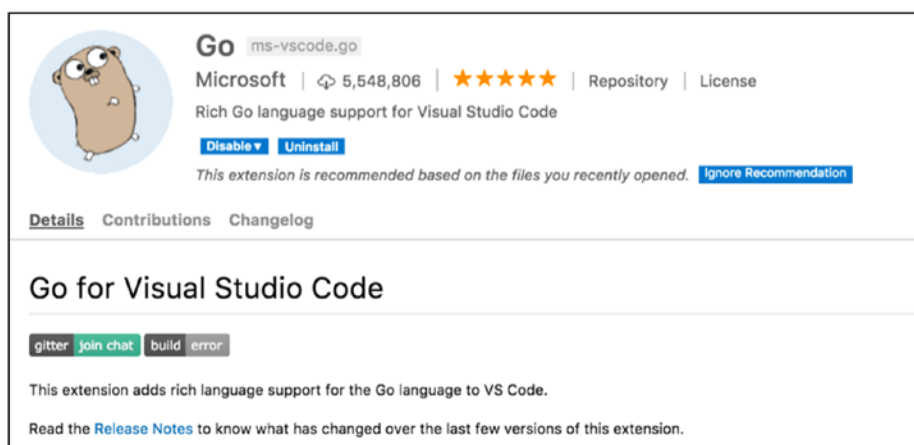


Figure 9-3. *The Go plug-in for Visual Studio Code*

The Go-related tasks.json for the Visual Studio Code build tasks, and its Command+Shift+B shortcut, required to execute code from within Visual Studio Code, is written following, for convenience.

```
{
  "version": "2.0.0",
  "tasks":
```

```
[
  {
    "label": "letsgo",
    "command": "go",
    "args": [
      "run",
      "${file}"
    ],
    "options": {
      "cwd": "${workspaceRoot}"
    },
    "group": {
      "kind": "build",
      "isDefault": true
    }
  }
]
```

From the `tasks.json` file, you'll notice that the command to build a Go program is `go run<filename>`.

Now, let's move on to the first Go program.

Let's Go

The basic structure of a program in Go is separated into three main blocks.

- The package definition, done with `package`
- The imports, all defined in one block
- The main function that gets executed when running the `go run` command

We'll start our Go adventures with a program that will read text from a file. This technique will be reused for reading a token for our bot, later in this chapter.

The project structure for a Go project relies on a folder with only one main function and a selection of Go source files, each ending with the `.go` extension. Our first setup will have a `reading.go` file to write the go source code and a text file, `file.txt`, to read sample text from.

```
$ tree
```

```

.
├── file.txt
└── reading.go
```

```
0 directories, 2 files
```

Now on to the Go code itself. As presented in the preceding bulleted list, the source file has three sections, starting with the package definition. You'll also notice that if you forget to write the package definition, the Visual Studio Code plug-in will automatically add it for you on saving.

There are two imports we are going to use:

- `io/ioutil`, to read the file content
- `fmt`, which is quite classic, to print on the standard output

With the help of auto-completion, it's easy to browse your way through the different packages in Go, as shown in Figure 9-4.

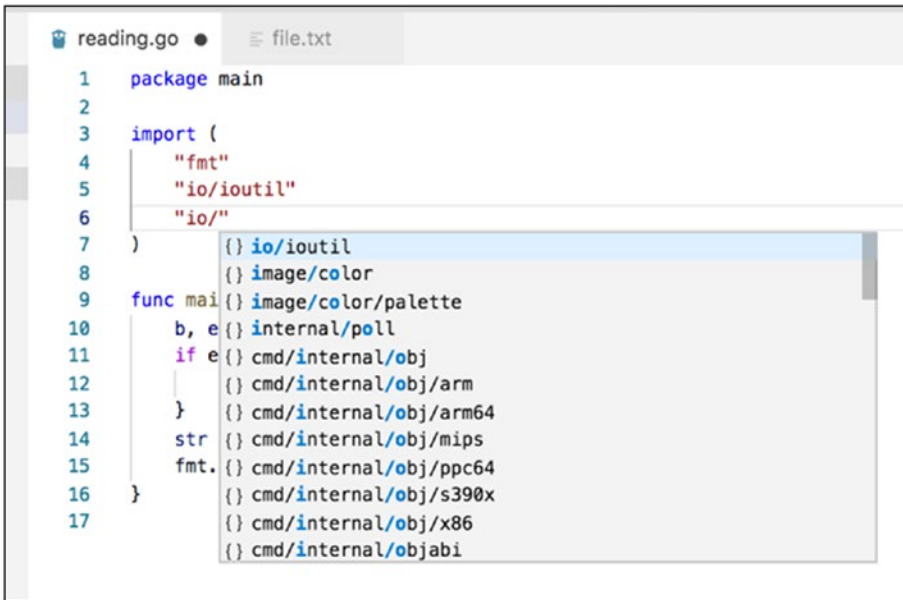


Figure 9-4. Importing packages through completion in Visual Studio Code

Before going on and explaining the main function, let's copy and paste the following code in the `reading.go` file and execute the code first.

```

package main

import (
    "fmt"
    "io/ioutil"
)

func main() {
    b, err := ioutil.ReadFile("file.txt")
    if err != nil {
        fmt.Println(err)
    }
}
  
```

```

    str := string(b)
    fmt.Println(str)
}

```

Now open the folder containing the `reading.go` and `file.txt` files and use the Visual Studio Code build command defined during the installation process, as shown in Figure 9-5.



Figure 9-5. Our first Go program reads content from a file

So, the file does not contain a token yet, but it does contain some text that was read by the program from the main function. What was the function doing, by the way?

First, we used `ReadFile` from the `ioutilgo` package, to open and read the full contents of file to an array of bytes.

```
b, err := ioutil.ReadFile("file.txt")
```

`ReadFile` actually returns two values, one for the content of the file in bytes, and one in case an error has occurred while reading those bytes from the file. As you can see, Go makes it easy to assign the returned results to multiple variables.

Now, let's see if an error occurred, by checking whether `err` is `nil`. If it's not, let's use `fmt` and its `Print` function, to display the content of the `err` variable itself in the standard output.

```
if err != nil {
    fmt.Print(err)
}
```

If the code has executed till this `if` statement, we're in pretty good shape. Let's convert the byte content returned by `ReadFile` to a string and display the content of the string, again using `fmt`'s `Println`.

```
str := string(b)
fmt.Println(str)
```

Great. We're all set up, and our first Go program executed properly.

Note here that you can generate a binary file from all the source files contained in this folder, using the `build` subcommand of Go.

`Gobuild` comes in two forms, one for which you specify the file name and one for which you don't. It is usually better to separate projects into different folders and use the version of the `build` command without the parameter.

```
go build
```

After you execute the command, a new file, called `go1`, will be generated in the project's folder, as shown in [Figure 9-6](#).

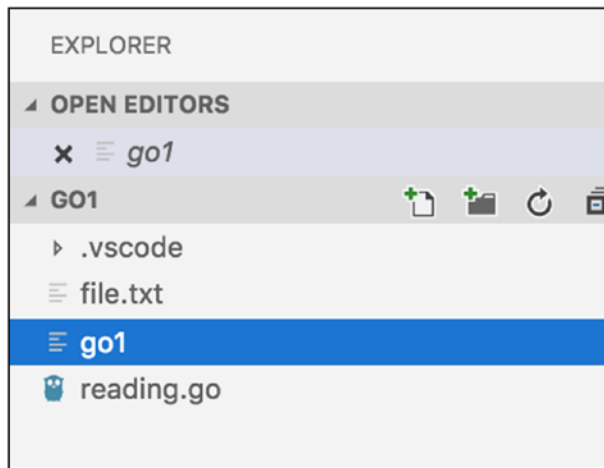


Figure 9-6. The executable binary *go1* generated by the *build* command

Also note that the file name of the generated binary is by default the name of the folder containing the code, not the Go source file containing the main function.

Let's quickly confirm that the binary works as expected.

```
$ ./go1
```

I contain a token

If you move one folder up, in the folder in which the `file.txt` is not present, you can also confirm that the program crashes and displays an error message, the one from the `if` block checking for `err`.

```
$ ./go1/go1
```

open file.txt: no such file or directory

Sweet! Let's move to `Fib` now!

Let's Fib

For this Fibonacci Go program, we will be using the recursive route. This implementation brings four new procedures, compared to the first simple program we just executed.

First, we'll define a new function separated from `main`, named `Fibonacci`, which will simply call itself recursively with different parameters, `n-1` and `n-2`. Then, we will implement the `main` function that will retrieve the first parameter that is being sent to the program. We'll then convert its value from a string to an integer, using a function named `Atoi` from the `strconv` package. Last, we'll call the `Fibonacci` function with the integer parameter and print the result of the call to the `Fibonacci` function in the standard output.

The Go code snippet is as follows:

```
package main

import (
    "fmt"
    "os"
    "strconv"
)

// Fibonacci computes fibonacci by recursion
func Fibonacci(n int) int {
    if n <= 1 {
        return n
    }
    return Fibonacci(n-1) + Fibonacci(n-2)
}

func main() {
    i, _ := strconv.Atoi(os.Args[1])
    fmt.Println(Fibonacci(i))
}
```

The code is relatively easy to follow at this stage, the main point being the conversion of the parameter to an integer. `os.Args` retrieves the array of parameters, which starts at 0 with the command path itself. Note that if you try to print the parameter at index 0 with

```
fmt.Println(os.Args[0])
```

you'll get the automatically generated path from Visual Studio Code.

```
/var/folders/8g/42979vpd0m1_ly722rgl3x780000gp/T/go-  
build290797788/b001/exe/fib
```

Looking a bit more closely at the main function, you'll see that the conversion done with `Atoi` also returns two parameters and that you can also ignore the case in which there is an error, with the symbol `_`. If you try to type in a variable name but do not use it, the Go compiler will complain, as shown in Figure 9-7.

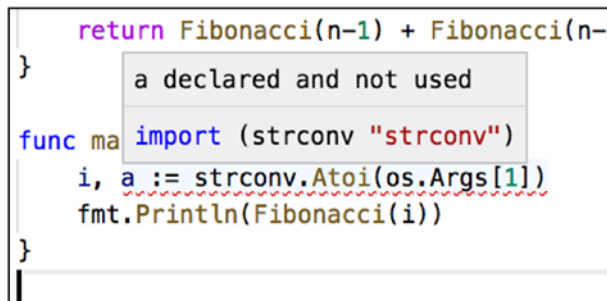


Figure 9-7. In Go, you cannot declare a variable and not use it

In Visual Studio Code, if you want to build and avoid hard-coding the parameters in the code itself, you can update the `tasks.json` a little, as shown in Figure 9-8.

```

{
  "label": "letsgo",
  "command": "go",
  "args": [
    "run",
    "${file}",
    "10"
  ],
  "options": {
    "cwd": "${workspaceRoot}"
  },
  "group": {
    "kind": "build",
    "isDefault": true
  }
}

```

Figure 9-8. Add 10 as a parameter to the program, to execute from within the Visual Studio Code

You can, of course, use `go build` to compile and run from the command line as well.

```

$ go build
$ ls
fib.go go2
$ ./go2 10
55
$ ./go2 100
... wait for ever

```

Recursive Fibonacci does not seem to be so performant. But I'll let you implement a faster version on your own and move on to the Telegram bot.

First Bot in Go

The Go language has one of the best libraries to interact with the Telegram Bot API. Its name is `telebot`, and you can find it on GitHub at <https://github.com/tucnak/telebot#overview>.

You don't really have to download it, because the Go command line can do that for you with the `get` subcommand. To install `telebot` on your local machine, use the following command:

```
go get -u gopkg.in/tucnak/telebot.v2
```

`-u` tells the `go get` command to connect to the network and look for updates if needed.

The code to follow is building on the two first examples of this chapter. In this first bot, we'll

- Require the `telebot` library
- Read the token from a file
- Use this token to initialize a polling bot
- Add a basic handler for the bot command

Note how you can give a prefix to the packages imported. The command below shows both the package name to be imported and the prefix used for all the functions, here, `telegram`.

```
telegram "gopkg.in/tucnak/telebot.v2"
```

You'll also note that the auto-completion from Visual Studio Code gives you a clean and convenient visual access to the functions exposed by the `telebot` library, as shown in Figure 9-9.

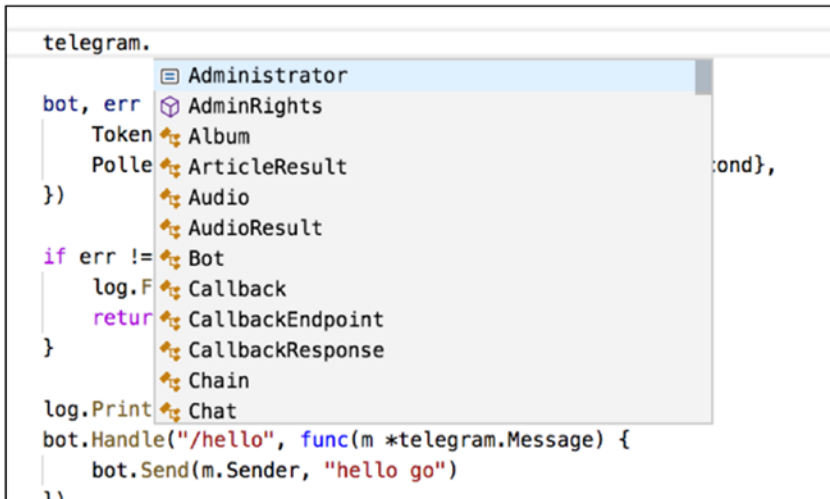


Figure 9-9. Familiar APIs and constructs

In the upcoming code listing, there are mainly two new constructs that you have not really seen before. First is the fact that you can assign a reference to a custom data type, or struct, instead of referencing the struct itself, using the `&` sign (ampersand).

```
Poller: &telegram.LongPoller{Timeout: 10 * time.Second}
```

You use a pointer instead of a struct literal in mostly two situations:

- When the struct is big and you pass it around
- When the struct is meant to be shared, that is, all modifications affect the struct, instead of affecting the copy

The second new piece of code relates to the definition of a callback using an anonymous function. In this case, you get a pointer to a Telegram message object, instead of passing a copy of the message.

```
bot.Handle("/hello", func(m *telegram.Message) {
    // implement logic here
})
```

Now that the hard parts are over, there it is: our first bot in Go. It will answer “hello go” whenever a /hello command is sent to the chat.

```
package main

import (
    "io/ioutil"
    "log"
    "time"

    telegram "gopkg.in/tucnak/telebot.v2"
)

func main() {
    token, _ := ioutil.ReadFile("token")

    bot, err := telegram.NewBot(telegram.Settings{
        Token: string(token),
        Poller: &telegram.LongPoller{Timeout: 10 * time.Second},
    })

    if err != nil {
        log.Fatal(err)
        return
    }

    log.Println("Starting GO bot")
    bot.Handle("/hello", func(m *telegram.Message) {
        bot.Send(m.Sender, "hello go")
    })

    bot.Start()
}
```

If you execute from Visual Studio Code (Command+Shift+B) or from the command line (after gobuild), you will get a message that the bot is starting, as shown in Figure 9-10, and the bot answers properly, as shown in Figure 9-11.

```
> Executing task: go run /Users/niko/Dropbox/BOOKS2/APRESS/09-go/firstbot/main/firstbo.go <
2018/09/23 13:03:10 Starting GO bot
```

Figure 9-10. Starting the Go bot



Figure 9-11. hello go

The rest of the telebot documentation is well-organized, and you can refer to it for more details. For convenience, the main handlers that can be used to capture different types of messages are shown following:

```
b.Handle(tb.OnText, func(m *tb.Message) {
    // all the text messages that weren't
    // captured by existing handlers
})

b.Handle(tb.OnPhoto, func(m *tb.Message) {
    // photos only
})

b.Handle(tb.OnChannelPost, func (m *tb.Message) {
    // channel posts only
})
```

```
b.Handle(tb.Query, func (q *tb.Query) {
    // incoming inline queries
})
```

Just Sending Pictures

The second example will not create a polling bot but actually send a picture to a given user, using parameters passed from the command line. As you remember, in downloading files from the Telegram API, there is a bit of magic between filepath and fileid, etc. telebot has a very clean implementation to handle these files and media files properly, without doing the job twice.

Within the telebot documentation, examples show how to read from disk, or read document from a URL, filling in photo or video Go structs, as required.

```
p := &tb.Photo{File: tb.FromDisk("chicken.jpg")}
v := &tb.Video{File: tb.FromURL("http://video.mp4")}
```

Once the media object is created, you can send it through the `SendAlbum` function. Following is an example of sending a group a photo and a video via the chat.

```
msgs, err := b.SendAlbum(user, tb.Album{p, v})
```

We'll make use of these functions to create an object and send it to the chat.

When calling our program, we'll require two parameters. The id of the user will be the first parameter for which we'll use the already seen `os.Args` and convert the id to an integer, per the `User` struct.

```
idd, _ := strconv.Atoi(os.Args[1])
```

We then create a user using the `User` struct and only fill the `ID` field of the user.

```
user := telegram.User{ID: idd}
```

Then, to load the photo, we'll use the `telegram.Photo` struct, with the second parameter of the program, again retrieved via `os`.

```
p := &telegram.Photo{File: telegram.FromDisk(os.Args[2])}
```

Finally, as was shown in the documentation, the `SendAlbum` function makes use of both the user and the photo prepared structs to send the media object to the Telegram bot.

Here is the full listing for the picture-sending program.

```
package main

import (
    "io/ioutil"
    "os"
    "strconv"
    telegram "gopkg.in/tucnak/telebot.v2"
)

func main() {
    token, _ := ioutil.ReadFile("token")
    bot, _ := telegram.NewBot(telegram.Settings{Token:
        string(token)})
    idd, _ := strconv.Atoi(os.Args[1])
    user := telegram.User{ID: idd}
    p := &telegram.Photo{File: telegram.FromDisk(os.Args[2])}
    bot.SendAlbum(&user, telegram.Album{p})
}
```

After pre-compiling the code within Visual Studio Code, you can build the program using the go binary on the command line.

```
go build
```

Finally, execute the newly created binary to send arbitrary photos to the chat rooms.

```
./thirdbot <user_id><picture_filename>
```

And see in Figure 9-12 how the picture appears, without the user having sent a message previously.

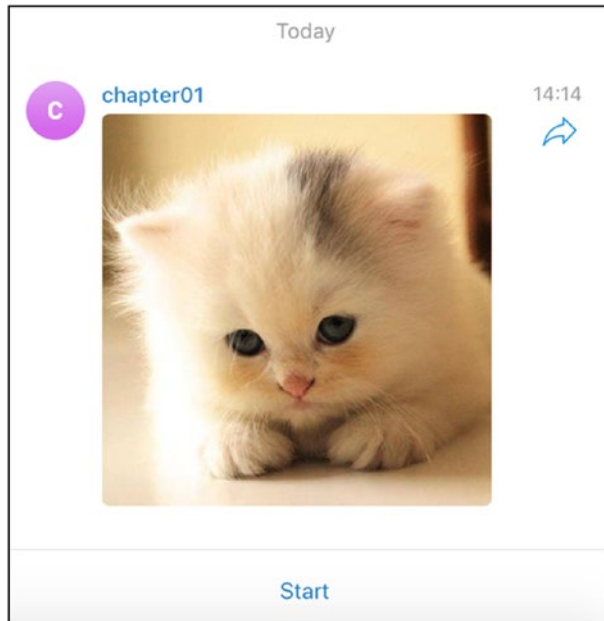


Figure 9-12. *Picture sent from the bot*

CHAPTER 10

Week 10: Elixir

*Clarity in my cup. Transparency of my soul. Lucidity of myself.
Elixir of the ages. Tea makes us all sages.*

—Dharlene Marie Fahl

Elixir (<https://elixir-lang.org/>) is a programming language that has a syntax very similar to Ruby but runs on the highly distributed Erlang VM.

The Erlang VM was developed by Ericsson and has been around for ages, since 1986 to be exact, and over time has proven its resilience across heterogeneous and highly concurrent environments. Usually, Erlang is used for programs that require the following:

- Distribution
- Fault-tolerance
- (Soft) real-time capabilities
- Highly available nonstop applications
- Hot swapping

Two main features fall under Erlang:

- The Erlang runtime, or the virtual machine (VM)
- The Erlang programming language

Because I do not want this chapter to be only a series of bullet points, I'm going to focus on the Erlang runtime—the virtual machine, not the language. To code on the Erlang VM, we will use prefaced Elixir, a language that has a lower learning curve but exposes strong functional programming concepts as well (see Figure 10-1 for a nice logo).



Figure 10-1. Elixir logo

Just as with Clojure, Elixir comes with an environment that is well prepared to being edited as a read-eval-print-loop (REPL) or line by line, so you will encounter a bit of both in this chapter. And, as written in the forum,

There's no time like the present to jump into Elixir—the functional language that's taking the programming world by storm.

Let's move on to installing Elixir.

Installation

The Elixir installation page can be found at <https://elixir-lang.org/install.html>. The link describes every possible way of installing the tools for Elixir, from macOS, Linux, and Windows all the way to Raspberry Pi and Docker.

```
#macOS with homebrew  
brew install elixir
```

```
# manjaro with pacman
pacman -S elixir

# chocolatey
cinst elixir

...
```

Once you are set up, you should have two main commands ready for you: `iex` and `mix`.

```
$ mix --version
Erlang/OTP 21 [erts-10.0.7] [source] [64-bit] [smp:4:4]
[ds:4:4:10] [async-threads:1] [hipe] [dtrace]

Mix 1.7.3 (compiled with Erlang/OTP 21)

$ iex --version
Erlang/OTP 21 [erts-10.0.7] [source] [64-bit] [smp:4:4]
[ds:4:4:10] [async-threads:1] [hipe] [dtrace]

IEx 1.7.3 (compiled with Erlang/OTP 21)
```

`iex` is the interpreter, or REPL, for Elixir, with which you can test and write your code directly at a prompt. `mix` is the Elixir project management and build tool.

Using iex

Most of `iex` usage is for executing commands, one by one. Here are a few examples of Elixir generating output text on the standard output, using the `I0` module and reading data from file using the `File` module.

```
iex(1)> I0.puts "hello"
hello
:ok
```

```
iex(2)> File.read! "secret.key"
** (File.Error) could not read file "secret.key": no such file
or directory
      (elixir) lib/file.ex:319: File.read!/1

iex(2)> File.read! "mybot/secret.key"
"585672177:.."
```

Once you have finished your iex session, you can end it with Ctrl+C, followed by abort, to terminate the VM.

```
iex(7)>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

Using mix

mix is the Elixir project management tool. You can create a new project with it using the new subcommand.

For example, for the following Telegram project, you can generate your new project with:

```
$ mix new telegrambot
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/telegrambot.ex
* creating test
* creating test/test_helper.exs
* creating test/telegrambot_test.exs
```

Your Mix project was created successfully.

You can use "mix" to compile it, test it, and more:

```
cd telegrambot  
mix test
```

While you're at it, you may have heard that Elixir has a very good web framework named Phoenix.

<https://phoenixframework.org/>

You can generate a ready-to-code project template for Phoenix, using the `phoenix.new` subcommand for `mix`.

For example:

```
mix phoenix.new hellophoenix
```

This will generate the project structure for a Phoenix application. This is beyond the scope of this chapter, but you should definitely have a look if you are curious to see a modern web framework.

Running iex with mix

Why would you want to run `iex` with `mix`? Because you can have all the project management done by `mix` within the REPL, presented by `iex`, which means that all your dependencies, configuration, and project code will be recognized, loaded, and available. The way to do that is by running the following command on the command line:

```
iex -S mix
```

The `-S` flag here tells `iex` to load `mix`, and `mix` will load itself and the project defined by the files of the generated `mix`-based project.

So, in the telegrambot folder that was created a few minutes ago...

```
$ iex -S mix
Erlang/OTP 21 [erts-10.0.7] [source] [64-bit] [smp:4:4]
[ds:4:4:10] [async-threads:1] [hipec] [dtrace]
```

Interactive Elixir (1.7.3) - press Ctrl+C to exit (type h()
ENTER for help)

And then you can try to execute code from the files in your project.

```
iex(1)> Telegrambot.hello
:world
```

But, by the way, what are all those project files made of?

Project Structure of a mix Project

mix generated a bunch of files for you, and the directory structure of the telegrambot folder is shown following.

```
.
├── README.md
├── config
│   └── config.exs
├── lib
│   └── telegrambot.ex
├── mix.exs
└── test
    ├── telegrambot_test.exs
    └── test_helper.exs
```

For our purposes, I will focus on describing the following Elixir files:

- `config.exs`: Your project config, defining keys, and values
- `telegrambot.ex`: Your own custom code and the entry point
- `mix.exs`: The mix project configuration

Note how `telegrambot.ex` is the only file to have the `.ex` extension, to recognize it as an entry point of the program.

config.exs

This is one of the places in which you can place the bot token.

```
use Mix.Config

# You can configure your application as:
#
#   config :telegrambot, key: :value
s#
# and access this configuration in your application as:
#
#   Application.get_env(:telegrambot, :key)
```

So, in the coming recipes, you can define your token like this:

```
config :telegrambot, token: "secret_token"
```

And in an `iex` session (start with `iex -S mix`), you can retrieve the token directly with

```
iex(1)> Application.get_env(:telegrambot, :token)
"secret_token"
```

There's not much else to see in the config file, so on to the project metadata.

mix.exs

A shorter version of the file is shown here for convenience.

```
defmodule Telegrambot.MixProject do
  use Mix.Project

  def project do
    [
      app: :telegrambot,
      version: "0.1.0",
      elixir: "~> 1.7",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end
  ...

  # Run "mix help deps" to learn about dependencies.
  defp deps do
    [
      # {:dep_from_hexpm, "~> 0.3.0"},
      # {:dep_from_git, git: "https://github.com/elixir-lang/
      my_dep.git", tag: "0.1.0"},
    ]
  end
end
```

Each `def` in this `mix.exs` file is an Elixir function. The `project` function returns a map of metadata, including app, project version, and the required minimal Elixir version.

`defp` is used to define a private Elixir function, and `dep()` is called only from the `project` function, so as to separate dependencies in a section on its own. We will see dependencies in a few lines.

`:app` is the name of the file to load from the `lib` folder. Although not used in this chapter, you could switch entry points, depending on environment variables.

You'll notice here again the way that the Elixir project uses Elixir as the configuration language. It's always nice to have consistency all around.

Dependencies

As you have seen just now, dependencies are defined and listed in the `deps` block of the `mix.exs` file. You will usually find your dependencies on hexdocs.

<https://hexdocs.pm/timex/getting-started.html>

Say, for example, that you want to add the `timex` library to your project, this is the way to import and use it in your project.

Why `timex`? Because if you search for a time library on hexdocs .pm, you'll find that `timex` is the first one to pop up. It also has the most downloads, as shown in Figure 10-2.

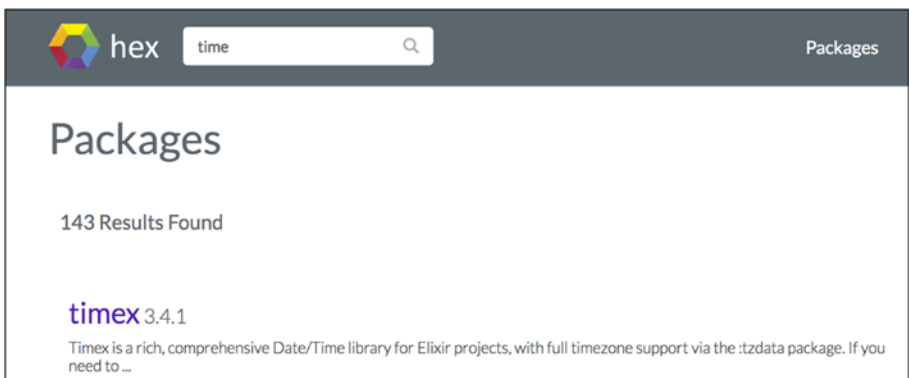


Figure 10-2. *Never-ending search for time*

Navigating to the `timex` page, you can see the way to add this to your `mix.exs` file on the right-hand side (Figure 10-3).

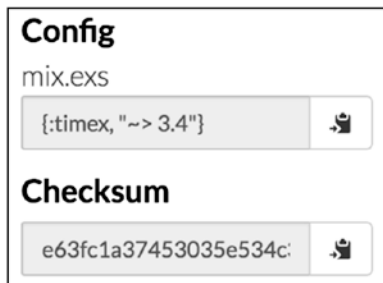


Figure 10-3. *timex* coordinates

You can now copy the config for `mix` into `mix.exs`, where the `dep` section of the `mix.exs` file should now look like the following:

```
defp deps do
  [
    {:timex, "~> 3.0"}
  ]
end
```

Once you have done this, you can ask `mix` to retrieve, download, and prepare the third-party libraries for you, using `mixdeps.get`.

Oh, by the way, here's quick reminder of the `mix` list of commands involved in dependencies.

```
$ mix help | grep deps
mix deps                # Lists dependencies and their status
mix deps.clean          # Deletes the given dependencies' files
mix deps.compile        # Compiles dependencies
mix deps.get            # Gets all out of date dependencies
mix deps.tree           # Prints the dependency tree
mix deps.unlock         # Unlocks the given dependencies
```

```

mix deps.update      # Updates the given dependencies
mix hex.audit        # Shows retired Hex deps for the current
                    # project
mix hex.outdated     # Shows outdated Hex deps for the current
                    # project

```

And so, once you have run `deps.get`, you can get to know whether the dependency is properly recognized by your project. See `timex` in the dependency tree following.

```

$ mix deps.tree
telegrambot
├── timex ~> 3.0 (Hex package)
│   ├── combine ~> 0.10 (Hex package)
│   ├── gettext ~> 0.10 (Hex package)
│   └── tzdata ~> 0.1.8 or ~> 0.5 (Hex package)
│       ├── hackney ~> 1.0 (Hex package)
│       │   ├── certifi 2.4.2 (Hex package)
│       │   │   └── parse_trans ~>3.3 (Hex package)
│       │   ├── idna 6.0.0 (Hex package)
│       │   │   └── unicode_util_compat 0.4.1 (Hex package)
│       │   ├── metrics 1.0.1 (Hex package)
│       │   ├── mimerl 1.0.2 (Hex package)
│       │   └── ssl_verify_fun 1.1.4 (Hex package)

```

You haven't been writing code for so long, you must be getting a bit excited by now. Let's start an elixir REPL

```
iex -S mix
```

and see if `timex` is properly loaded. now is the `timex` function to retrieve the current time in the current time zone, but before using a separate module, you must use it, as shown following:

```
$ iex -S mix
iex(1)> use Timex
Timex.Timezone

iex(2)> Timex.now
#DateTime<2018-09-24 02:31:42.450412Z>
```

Now, let's have a look at where the coding goes.

telegrambox.ex

Finally, in the list of files for the project, you have `telegrambot.ex`, in which you put all your custom code. As in Ruby, and as in the `mix.exs` file, a lot is done with the `def` module and `def`. Each `def` block defines a function. The default file generated by `mixnew` is shown as follows, where `Telegrambot` has one function.

```
defmodule Telegrambot do

  def hello do
    :world
  end

end
```

In an `iex` session, you would call that function as follows:

```
iex(1)> Telegrambot.hello
:world
```

Adding to what we have learned looking at the Timex dependency, we can use Timex and a new def block to give the time.

```
defmodule Telegrambot do
  use Timex

  def hello do
    :world
  end

  def timexnow do
    IO.puts Timex.now
  end
end
```

And at a new iex session,

```
iex(2)> Telegrambot.timexnow
2018-09-24 02:38:15.954175Z
:ok
```

(Back to) Dependencies

Yes, we're back! You quickly saw how to add dependencies to your project, but Elixir/mix has this wonderful way of adding dependencies directly from Git projects as well (and other source control repositories, by the way).

Searching for a library on `hex.pm` is easy enough (Figure 10-4).

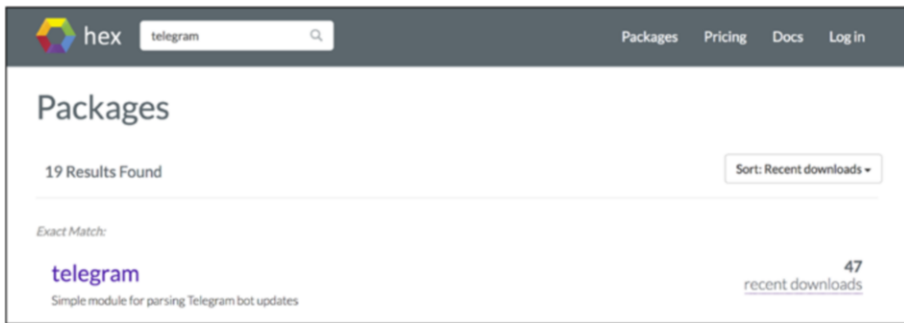


Figure 10-4. Finding a library with Hex

And, yes, the library we want to use for the Telegram bot is the one that was found, so go to

<https://github.com/visciang/telegram>

As recommended on the library web site, you can add the library directly to the `mix.exs` file, using the Git repository address and a tag, as follows:

```
{:telegram, git: "https://github.com/visciang/telegram.git",
tag: "0.5.0"}
```

By now, your `deps` block should look like this:

```
defp deps do
  [
    {:timex, "~> 3.0"},
    {:telegram, git: "https://github.com/visciang/telegram.
git", tag: "0.5.0"}
  ]
end
```

Retrieving the dep is again done with `mixdeps.get`, which will transparently check out the Telegram project locally for you. The following `mixdeps.tree` command confirms that the library is present, and you can also see the Git repository of Telegram showing up in the output of the command.

```
$ mix deps.tree
telegrambot
├── timex ~> 3.0 (Hex package)
│   ├── combine ~> 0.10 (Hex package)
│   ├── gettext ~> 0.10 (Hex package)
│   └── tzdata ~> 0.1.8 or ~> 0.5 (Hex package)
│       └── hackney ~> 1.0 (Hex package)
│           ├── certifi 2.4.2 (Hex package)
│           │   └── parse_trans ~>3.3 (Hex package)
│           ├── idna 6.0.0 (Hex package)
│           │   └── unicode_util_compat 0.4.1 (Hex package)
│           ├── metrics 1.0.1 (Hex package)
│           ├── mimerl 1.0.2 (Hex package)
│           └── ssl_verify_fun 1.1.4 (Hex package)
└── telegram (https://github.com/visciang/telegram.git)
    ├── tesla ~> 1.0 (Hex package)
    │   ├── hackney ~> 1.6 (Hex package)
    │   ├── jason >= 1.0.0 (Hex package)
    │   └── mime ~> 1.0 (Hex package)
    ├── hackney ~> 1.9 (Hex package)
    └── jason ~> 1.0 (Hex package)
```

Telegram also seems to sometimes require that the dependencies be recompiled beforehand, and this is done with `mix deps.compile` (and, if required, `domix deps.clean`, also beforehand).

```
$ mix deps.compile  
==> Compiling parse_trans  
==> Compiling mimerl  
==> Compiling metrics  
==> Compiling unicode_util_compat  
==> Compiling idna  
==> Compiling ssl_verify_fun  
==> Compiling certifi  
==> Compiling hackney
```

Now, finally, on to some more fun with mix and Telegram.

Get Something

In this section, we'll have a look at sending a few requests to the Telegram API directly. For authentication, all those calls will be using the bot token.

GetMe

Let's try to simply send a request to the Telegram API with our local bot. You'll remember the following GetMe method:

<https://core.telegram.org/bots/api#getme>.

We will send the request from an iex/mix session.

```
$ iex -S mix
```

First, we load the token from the configuration file, so make sure you have your token inserted properly in config/config.exs at this stage.

```
iex(1)> token = Application.get_env(:telegrambot, :token)  
"585672177:..."
```

Then we can call the Telegram API, with the token, and the `getMe` request.

```
iex(2)> Telegram.Api.request(token, "getMe")
{:ok,
 %{"first_name" => "chapter01",
  "id" => 585672177,
  "is_bot" => true,
  "username" => "chapter01bot"
 }}
```

Request has gone through, and we can use the usual Telegram User object, with the bot name and the bot id.

GetChat

The `getChat` Telegram function documentation is located at <https://core.telegram.org/bots/api#getchat>. Unlike the function `getMe`, which can be called without a parameter, `getChat` requires a `chat_id`. Parameters to be sent along the request with the Telegram library are just appended to the request call.

Let's see that in action in the same `iex` session.

```
iex(8)> Telegram.Api.request(token, "getChat", chat_id: 121843071)
{:ok,
 %{"first_name" => "Nico",
  "id" => 121843071,
  "last_name" => "Nico",
  "photo" => %{"big_file_id" =>
    "AQADBQADQakxG38tQwcACAox1TIABLM4sAnsmf3pPM4AAgI",
```



```

    "small_file_id" =>
"AQADBQADQakxG38tQwcACAox1TIABCSohuBN4Zh10s4AAgI"
  },
  "type" => "private",
  "username" => "hellonico"
}}

```

As usual, the response status and the structure are printed in the output of the session.

GetFile

You may have noticed a file id in the user profile while calling `getChat` in the preceding snippet. Let's try to retrieve that file, with `getFile`.

The `getFile` function details are located in the bot API at

<https://core.telegram.org/bots/api#getfile>

And so, in the same iex session again, we use the following:

```

Telegram.Api.request(token, "getFile", file_id:
"AQADBQADQakxG38tQwcACAox1TIABLM4sAnsmf3pPM4AAgI")

{:ok,
 %{
   "file_id" =>
"AQADBQADQakxG38tQwcACAox1TIABLM4sAnsmf3pPM4AAgI",
   "file_path" => "profile_photos/file_10.jpg",
   "file_size" => 35814
 }}

```

Ah, right...the Telegram API always returns a file path for download from its web site.

Using Elixir's System

To download the preceding file path, remember the trick used before, avoiding the need for an additional third-party library. We'll just use `curl`, that should already be installed on the local machine, at this stage.

In Elixir, calling a system command is done with `System.cmd`. The HTTP URL to download a file from a `file_path` is made using the following rule:

https://api.telegram.org/file/bot<token>/<file_path>

with the `file_path` of the form: `profile_photos/file_10.jpg`. That gives the full listing.

```
{:ok, res} = Telegram.Api.request(token, "getChat", chat_id:
121843071)
fileId = res["photo"]["big_file_id"]
{:ok, res2} = Telegram.Api.request(token, "getFile", file_id:
"#{fileId}")
System.cmd("curl",
["-O", "https://api.telegram.org/file/bot#{token}/#{res2["file_
path"]}"])
```

And the author's profile picture is shown (Figure 10-5).

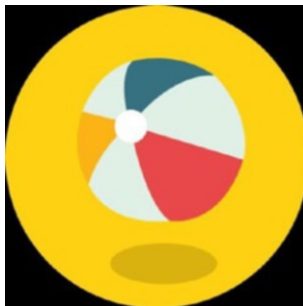


Figure 10-5. Beach ball, not the OS X one

SendPhoto

Sometimes it's not what you can get but what you can give that matters. Following the same pattern, you can send a picture, using `sendPhoto` with the exact same construct.

```
token = Application.get_env(:telegrambot, :token)
chat_id = 121843071
photo = "cat.jpg"
Telegram.Api.request(token, "sendPhoto", chat_id: chat_id,
photo: {:file, photo})
```

And provided you have copied the `cat.jpg` in your project folder, the usual cat photo is showing in the Telegram chat (Figure 10-6).



Figure 10-6. *This is not Marcel, but it is a cat*

You've probably realized that all this can run directly from Visual Studio Code, using the task defined for running `.exs` file at the beginning of this chapter. Try it out (Figure 10-7).

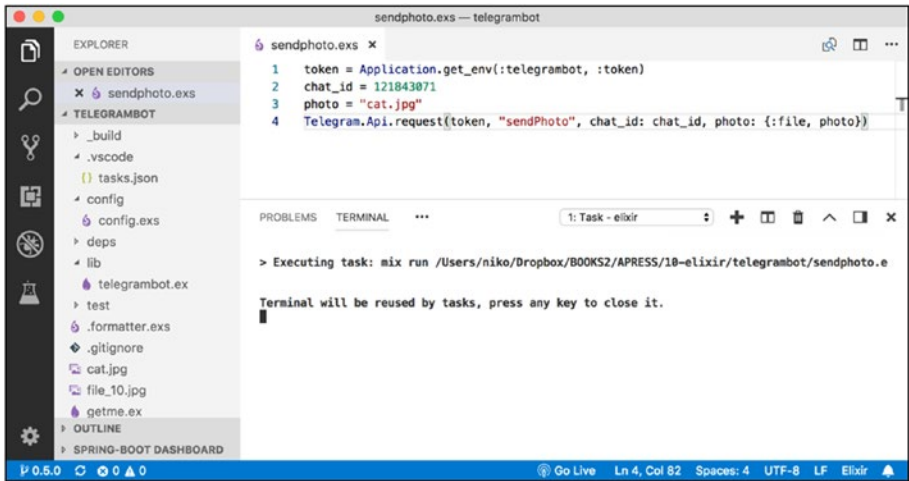


Figure 10-7. Straight from your editor...

Telegram Bot

I've exceeded the optimum number of pages for this chapter, so I'll just offer a quick look at how to create and run a bot and how to implement a few commands for this new bot.

Bot1: Anything Goes

This first bot will send the full data structure of the received update to the chat. See how the token is passed to the `Telegram.Bot` library, after calling `use`?

```
defmodule Bot1 do
  use Telegram.Bot,
    token: Application.get_env(:telegrambot, :token),
    username: "chapter01bot",
    purge: true
```

```

message do
  request(
    "sendMessage",
    chat_id: update["chat"]["id"],
    text: "Hey! You sent me a message: #{inspect(update)}"
  )
end

{:ok, _} = Bot1.start_link()
Process.sleep(:infinity)

```

The last two lines start the bot in a separate thread and tell the main thread to sleep forever. See, too, how the request to `sendMessage` is a repeat of what was done in the previous section.

Bot2: Fibonacci

Our second bot will implement a command that can compute Fibonacci for you and send the result back to the chat. Commands with the Telegram library are simply defined using `command`. The command name is the first parameter of `command`, followed by possible arguments.

The command block itself has access to the update object coming from Telegram. Even though you don't see it so much, Elixir has this functional program gene under the hood, and most of the calls are done using `apply`.

This shows particularly where you have to use `Enum.at` to get the index of a specific element of a list, here, the first one, so 0.

```

command "fib", args do
  {intVal, ""} = Integer.parse(Enum.at(args,0))

  request("sendMessage", chat_id: update["chat"]["id"],
    text: "Fib[#{intVal}] = #{Fib.fib(intVal)}")
end

```

Here, `Fib` is defined in a separate `Fib` module, as shown in the full listing, to which a simple recursive Fibonacci implementation has been added.

```
defmodule Fib do
  def fib(0) do 0 end
  def fib(1) do 1 end
  def fib(n) do fib(n-1) + fib(n-2) end
end

defmodule Bot2 do

  use Telegram.Bot,
    token: Application.get_env(:telegrambot, :token),
    username: "chapter01bot",
    purge: true

  command "fib", args do
    {intVal, ""} = Integer.parse(Enum.at(args,0))
    request("sendMessage", chat_id: update["chat"]["id"],
      text: "Fib[#{intVal}] = #{Fib.fib(intVal)}")
  end

  any do
    IO.puts "not found"
  end
end

{:ok, _} = Bot2.start_link()
Process.sleep(:infinity)
```

Also, avoid the use of any function at the bottom of this second bot, for a catch-them-all message.

At last, you can define all other supported messages of the Telegram API in your new bots, and the list of possible blocks that can go in your bot are relisted here for convenience, straight from the Telegram library.

```
edited_message do
  # handler code
end

channel_post do
  # handler code
end

edited_channel_post do
  # handler code
end

inline_query _query do
  # handler code
end

chosen_inline_result _query do
  # handler code
end

callback_query do
  # handler code
end

shipping_query do
  # handler code
end

pre_checkout_query do
  # handler code
end
```

In addition, code in the `api.ex` file has a few more code construct samples related to keyboard and inline queries.

<https://github.com/visciang/telegram/blob/master/lib/api.ex>

CHAPTER 11

Week 11: Node.js

“The strength of JavaScript is that you can do anything. The weakness is that you will.”

—Reg Braithwaite

For the sake of completion, this book is also presenting Node.js and Telegraf, the library to deal with Telegram from Node.js. With Telegraf, Node.js may have the privilege of being one of the easiest libraries with which to interact with Telegram.

In this chapter, I will focus mostly on running Telegram Bot on `runkit.com`, a cloud service that allows you to have an instance of Node.js running in the cloud, without installing anything locally, thus reducing the time required to write and deploy Telegram bots to a matter of minutes.

Running in the cloud also allows us to have the necessary setup webhook for Telegrams. Basically, webhooks are a replacement for the Telegram bot polling method you have seen so far with a URL that the Telegram API will call on a new message, thus avoiding unnecessary polling traffic. To present a minimal running server that can answer the POST request from Telegram, we will use the Koa library, a replacement for ExpressJS, to set up the integration with Telegraf.

Excited? Let's hit the Node.js road.

Meet RunKit

In this section, you will see how to create a RunKit account, run a first Node.js program, and then write and publish a simple Koa-based service. RunKit starts by being a Node.js playground in your browser. You can access the RunKit the home page at <https://runkit.com> (Figure 11-1).

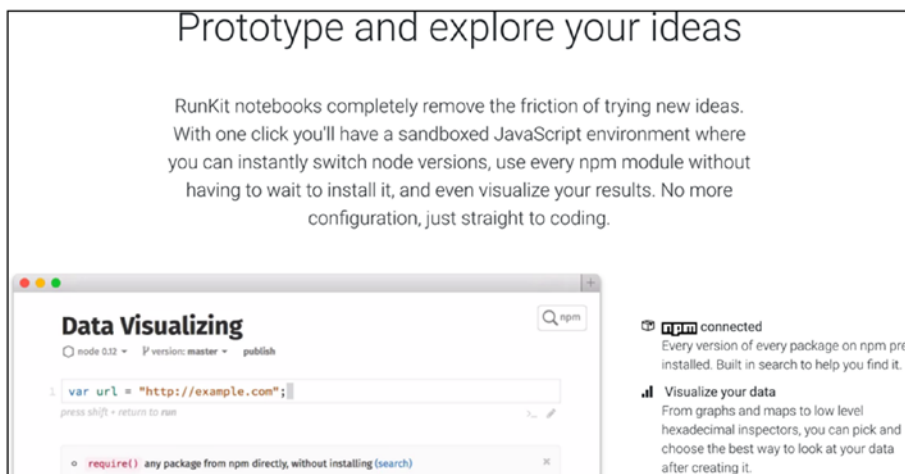



Figure 11-1. RunKit home page

RunKit is used to write test code, fetch and display data, share code, and run server code directly in the cloud.

Creating an Account

To create an account, you can sign up on runkit.com, using your GitHub account, or a standard username password combination (Figure 11-2).



Sign in with Github

or

Email address

Username

Password

Register

or, skip this and fill it out later

Already have an account? [Log in](#)

Figure 11-2. *Creating your RunKit account*

Once you are logged in, you can see a simple interface, with a menu on the left-hand side, mostly to create new playgrounds, and the list of playgrounds you have (will) created in the center of the page (Figure 11-3).

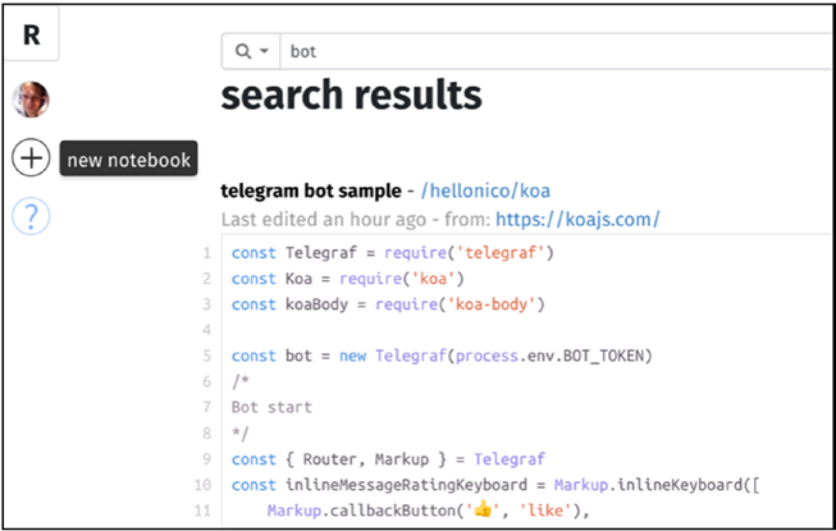


Figure 11-3. RunKit personal page

If you press the + button on the left-hand side, you are directed to an empty playground page (Figure 11-4).

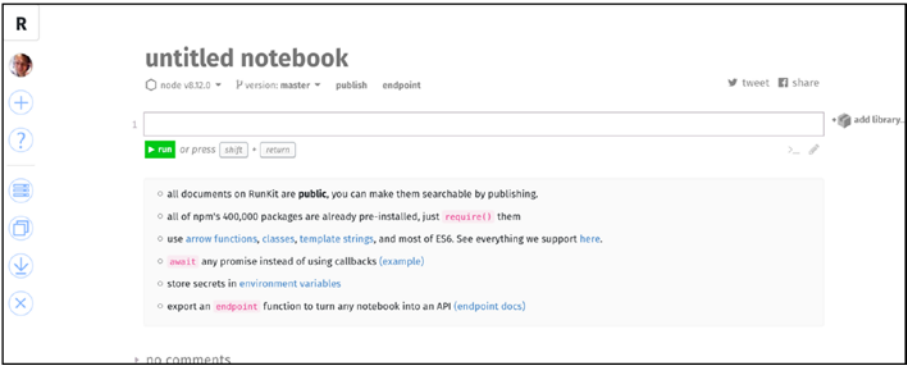


Figure 11-4. RunKit empty playground page

First Code on RunKit

From there, it won't be a surprise that you can just type in whatever JavaScript code you want to write. So, let's start by saying hello, as shown in Figure 11-5.



Figure 11-5. Say “hello” with RunKit

Executing is done either by clicking the run button or, as indicated, with shift+return. Each resulting line of the execution is shown following. Say you want to compute quickly some Fibonacci numbers...You could either implement this yourself, with a possible implementation shown following.

```
function myfibonacci(num) {
  if (num <= 1) return 1;

  return myfibonacci(num - 1) + myfibonacci(num - 2);
}
console.log(myfibonacci(39));
```

Or you could also require a Node.js library, available through RunKit (and which you can check at <https://npm.runkit.com/>).

So, for example, you could simply require a library, for example, `fibonacci-fast`, to compute those numbers directly from the playground.

```
var fibonacci = require('fibonacci-fast');
var result = fibonacci.get(39).number.toString();
console.log(result);
```

The playground will make that library available to your notebook, without you having to install or download anything. And with or without a library, it works all the same, as shown in Figure 11-6.

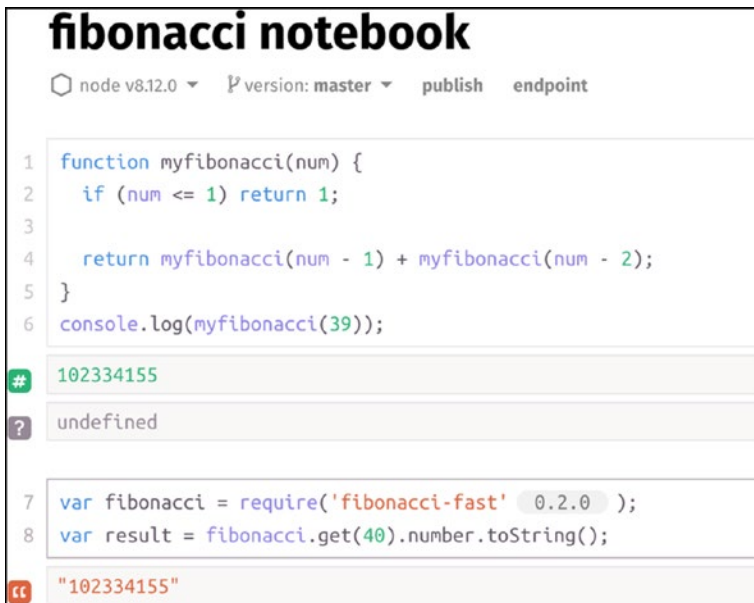


Figure 11-6. *Write it or require it*

Note how the version that was imported in the notebook is displayed right next to the `require` statement.

A Certain Je Ne Sais Koa

Koa is the next-generation web framework for Node.js. It is built with what was learned during the Express JS years and is a step easier to learn and maintain. Koa's home page can be found at <https://koa.js.com/#application>.

With what you've seen of the RunKit playground, you may already feel that you can require Koa libraries and start a server.

Following is a short snippet taken from the Koa samples:

```
const Koa = require('koa');
const app = new Koa();

app.use(async ctx => {
  ctx.body = 'Hello World';
});

app.listen(3000);
```

There are four main steps to creating a simple Koa application.

- Require koa
- Create a new Koa app
- Create an asynchronous handler
- Get the app to listen (on a port)

If you execute the code from the preceding snippet in a new notebook, you can instantiate a server directly from the browser (Figure 11-7).

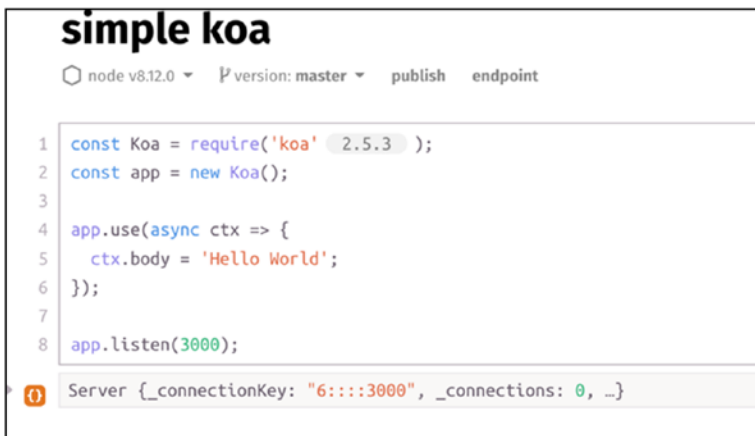


Figure 11-7. Running a server on RunKit

And, yes, this server is already listening to requests. So, if you click the end point click available in the notebook, you will open a page, with a temporary URL assigned to your Koa application, with the expected Hello World message showing up here again (Figure 11-8).

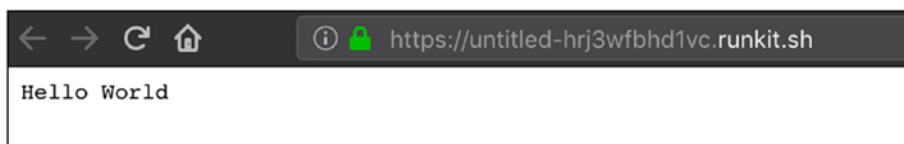


Figure 11-8. *Koa running in the cloud*

Publishing Some Koa

Publishing to a temporary URL is fine, but to work with Telegram Bot, you must have a somewhat permanent and public URL to work with. This can be achieved in RunKit by using the Publish link. Clicking the link will show a pop-up dialog asking you for some semantic (!) version of your application (Figure 11-9).

Ready to publish a release?

Choose a version number and release notes as needed

Notebook Name
hellonico simple-koa

Users can find this notebook at runkit.com/hellonico/simple-koa

Version
1.0.0

Releases use [Semantic Versioning](#)

Release Notes

cancel Publish

Figure 11-9. Ready to publish, Koa?

And now the Koa application is available at an easy-to-remember location, in the following form:

<https://runkit.io/hellonico/simple-koa/branches/master>

You can access it reliably from your browser. Note that you will actually get redirected to a URL mapped internally by RunKit (see Figure 11-10).

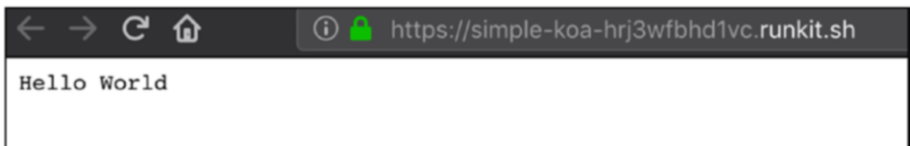


Figure 11-10. Published!

Also, note that the URL is via the HTTPS protocol, which is quite important, because Telegram does not accept pure HTTP end points.

From here, you should practice some more with Koa and RunKit, and once you are ready, let’s see how to write a first Telegram bot using RunKit, Koa, and that mysterious Telegram webhook.

Telegram Bot with Webhooks

So, first, webhooks...What are they? Webhooks work in the opposite direction as polling. Where polling is your client bot periodically asking the Telegram server if any new message has arrived, with webhooks, you ask the Telegram server to send you an update message whenever a new message that your bot should know about has been sent to a chat room.

The flow is summarized in Figure 11-11.

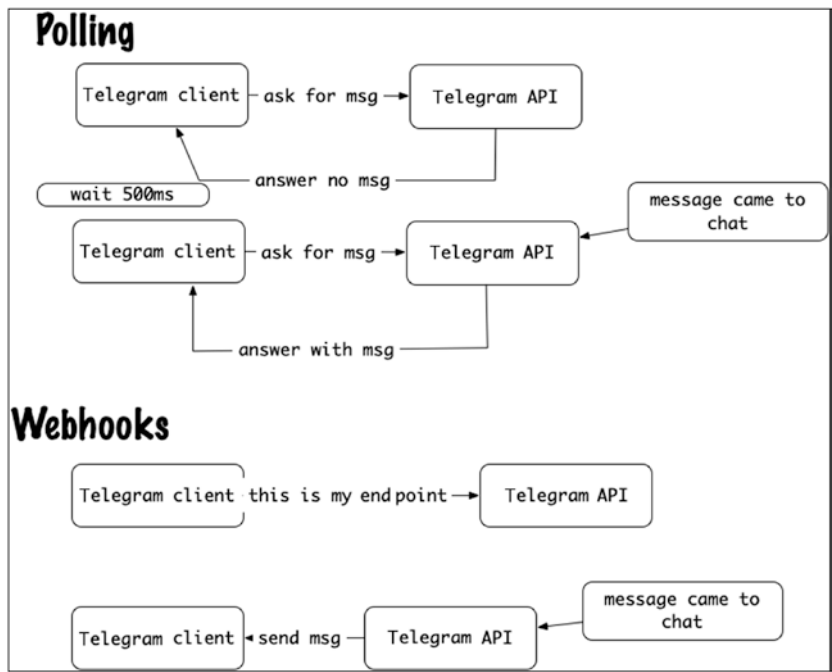


Figure 11-11. Polling vs. webhooks

Knowing this, we just need to know if there is a Telegram Bot API method to tell about your application end point—and there is one here:

<https://core.telegram.org/bots/api#setwebhook>

In the following snippet, you can see how the `setWebhook` function is used to publicize the URL to the bot API.

```
const Telegraf = require('telegraf')
const Koa = require('koa')
const koaBody = require('koa-body')

const bot = new Telegraf(process.env.BOT_TOKEN)
bot.on('text', ({ reply }) => reply('The time here is ::'+ new
Date()))

bot.telegram.setWebhook('https://runkit.io/hellonico/koa-bot/
branches/master')

const app = new Koa()

app.use(koaBody())
app.use((ctx, next) => ctx.method === 'POST' || ctx.url === '/'
secret-path'
  ? bot.handleUpdate(ctx.request.body, ctx.response)
  : next())
)
app.listen(3000)
```

It will show in Koa as in Figure 11-12.

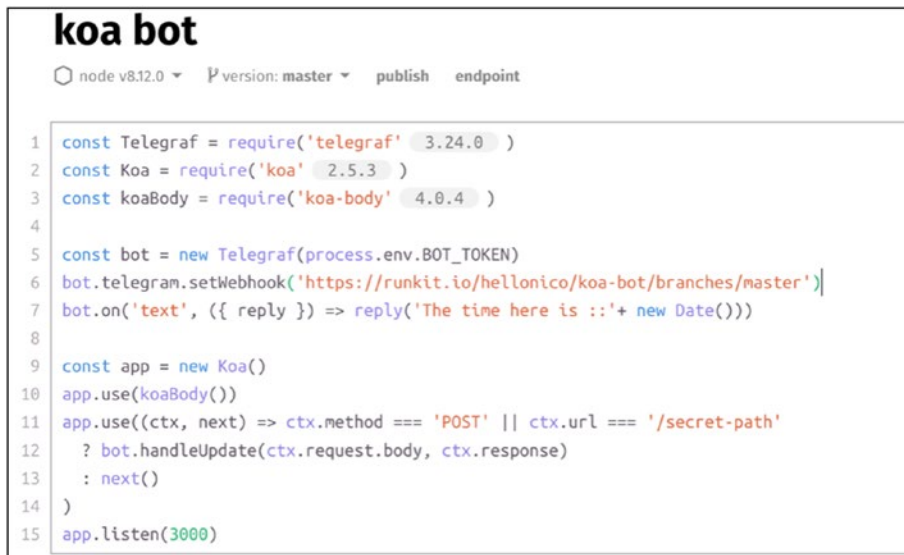


Figure 11-12. First bot on RunKit

Now, I can explain the two missing pieces of the code. The first is where we glue the HTTP POST request to the Telegraf bot, using `handleUpdate`, which does two things:

- Maps `ctx.request.body` to the input request for the bot
- Maps `ctx.response` to the output response from the bot

```
app.use(koaBody())
app.use((ctx, next) => ctx.method === 'POST' || ctx.url
=== '/secret-path'
  ? bot.handleUpdate(ctx.request.body, ctx.response)
  : next())
)
```

The second missing piece is similar to other APIs presented in this book.

```
bot.on('text', ({ reply }) => reply('The time here is ::'+ new Date()))
```

Whenever some text arrives to the bot, we have a callback, with a destructured reply object (prepared by Telegraf), which we can use to send messages back to the chat. Before running this bot, you will have noticed that the code requires a token retrieved from the environment that started the Node.js process.

```
const bot = new Telegraf(process.env.BOT_TOKEN)
```

RunKit allows you to do that, by setting environment variables to your notebooks(yes!).

To access the settings page, navigate to your own RunKit settings page (Figure 11-13).

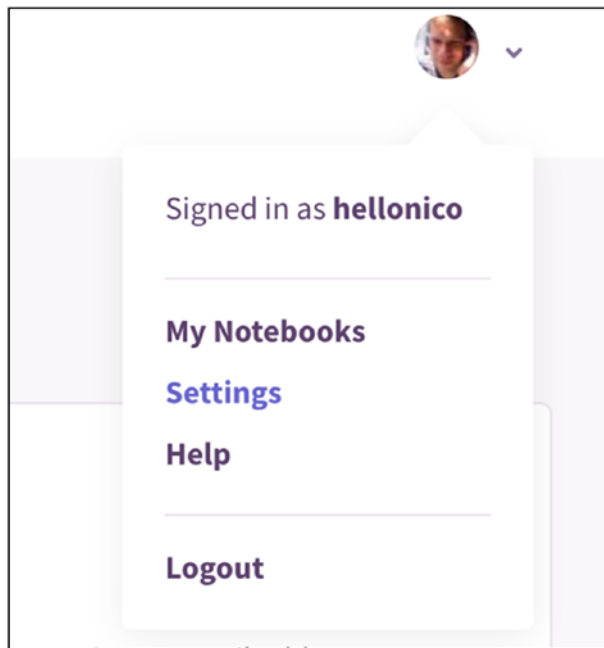


Figure 11-13. Settings page

On that page, there is a section with environment variables, with which you can set the needed `BOT_TOKEN` variable to instantiate the Telegraf bot (Figure 11-14).

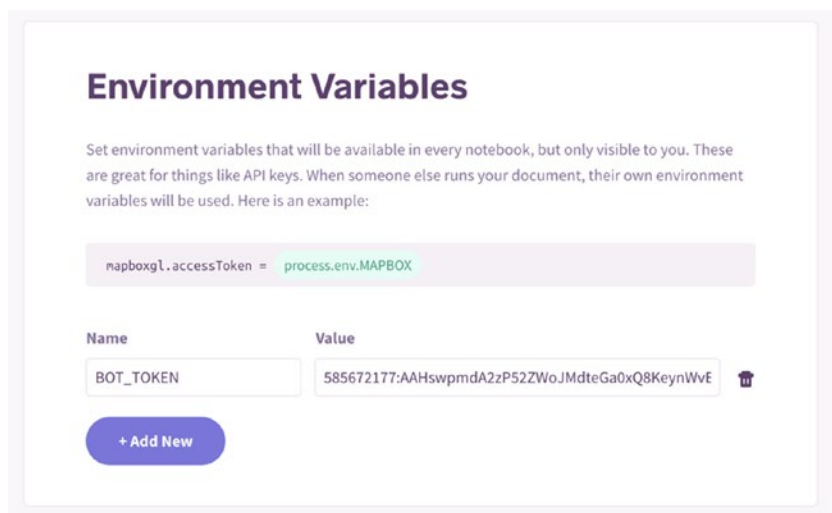


Figure 11-14. Setting the `BOT_TOKEN` variable with your own token

Now, you can start the RunKit/Koa/Telegram bot together and start chatting with your bot as usual (Figure 11-15).

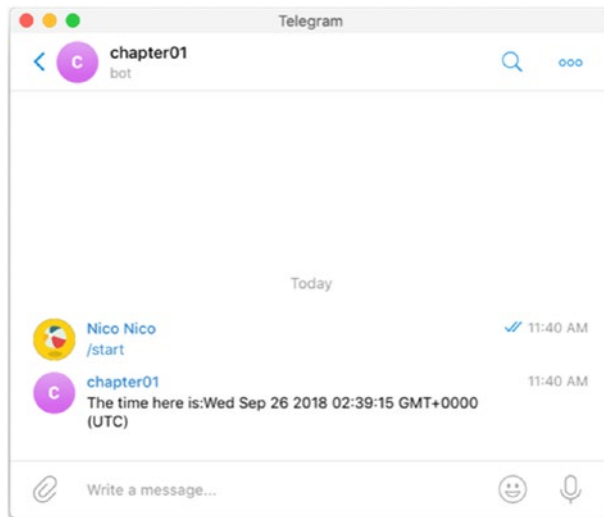


Figure 11-15. *Telegraf, with Koa on RunKit bot*

More on the Telegraf Library

I have not yet reviewed the Telegraf library in great detail. As a reminder, its GitHub URL is the one following:

<https://github.com/telegraf/telegraf/>

And you will find extensive examples on games and inline keyboards in its examples folder.

<https://github.com/telegraf/telegraf/tree/develop/docs/examples>

Image-to-Chat Example

The image command example is an instant self-gratification win. You can try this directly in the RunKit bot you have just defined.

```
const bot = new Telegraf(process.env.BOT_TOKEN)
bot.command('image',
```

```
(ctx) =>
ctx.replyWithPhoto({ url: 'https://picsum.
photos/200/300/?random' })))
```

Once the command is deployed, you can send the `/image` command to the bot and get a random picture from the Picsum web site. Note, too, how `replyWithPhoto` is used with the URL parameter, to send a picture to the chat (Figure 11-16).

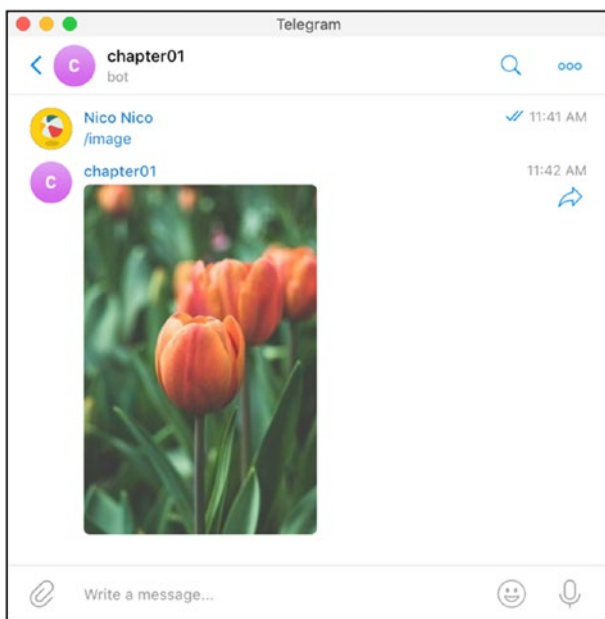


Figure 11-16. *Random photo*

RegExp, Inline Keyboards, and Embedded Emojis

Telegraf has this cool embedded feature that allows you to match messages on RegExp and perform actions depending on those matches. This is done using a regexp within either a `.hears` (for messages coming to a chat group) or `.on` (message directly sent to the bot) callback.

In the following, “like” is searched in group chat messages, and an inline keyboard shows up.

```
const { Markup } = Telegraf
const inlineMessageRatingKeyboard = Markup.inlineKeyboard([
  Markup.callbackButton('👍', 'like'),
  Markup.callbackButton('👎', 'dislike')
]).extra()
bot.hears(/like (.+)/, (ctx) => ctx.telegram.sendMessage(
  ctx.from.id,
  'Like?',
  inlineMessageRatingKeyboard)
)
```

Actions for callbacks themselves can be defined after. Here, we are editing the last message, the one with the inline keyboard in place, using the API function `editMessageText`.

```
bot.action('like', (ctx) => ctx.editMessageText('●● Awesome! ●●'))
bot.action('dislike', (ctx) => ctx.editMessageText('okey'))
```

When the code is run, the bot will notify you whenever it finds a message starting with “like,” as seen in Figure 11-17.

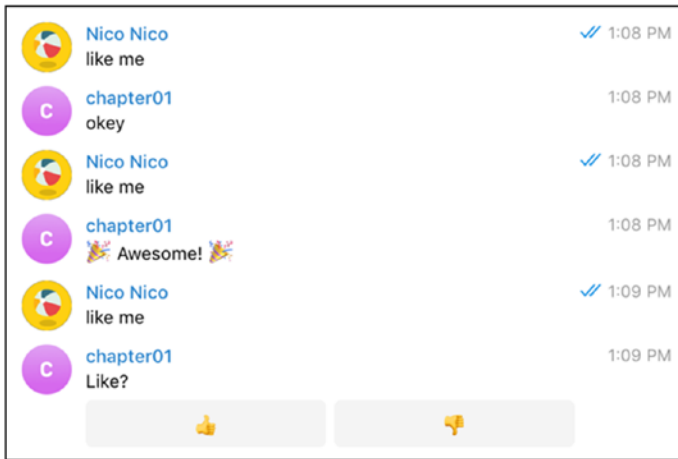


Figure 11-17. *Like me or not?*

Note that you can also use the matched pattern, by using `match` in the callback.

```
bot.hears(/reverse (.+)/,
  ({ match, reply }) => reply(match[1].split("").reverse().
    join(")))
```

Finally, let's have a look on how to run this bot locally or our own server.

Running Node.js Locally

After running all the examples from within RunKit, you may indeed want to host this somewhere other than RunKit, for example, on your own machine. For this, we will have to install Node.js.

Setting Up Node.js

Installing Node.js is a no-brainer—from either the home page, which has the necessary packages,

<https://nodejs.org/en/>

or from your usual package manager.

Also, whereas working with NPM is good enough, these days, the yarn build tool for Node.js is all the rage, and you can find/install it from <https://yarnpkg.com/en/docs/install#mac-stable>.

Once you have it installed, you can start with a few version checks.

```
$ node -v
v10.4.0
$ yarn -v
1.7.0
$ npm -v
6.2.0
```

When the tooling is set up, you can download the notebook you were working on in RunKit, using the download this notebook button (Figure 11-18).

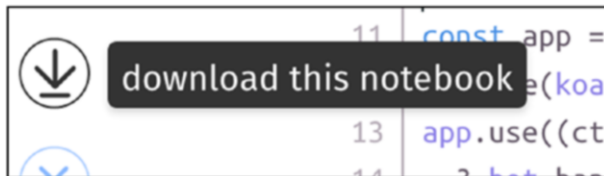


Figure 11-18. Downloading the RunKit notebook to your machine

Once you extract the content of the archive, you will see something similar to Figure 11-19.

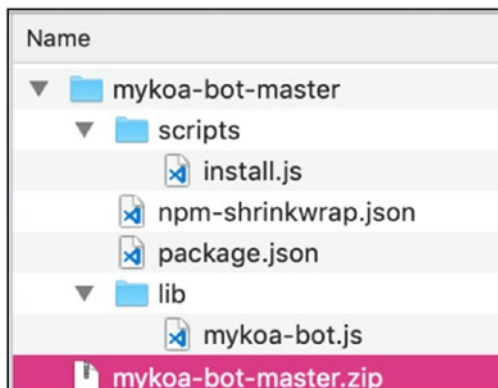


Figure 11-19. Content of *mykoa-bot* zip file from RunKit

Using Local Tunnel

To run this bot, you could use polling, but let's try something a bit more challenging. Whenever you start your Koa application, your application will begin to bind the listening host to localhost and (usually, by default) on port 3000.

Wouldn't it be great if something made this wonderful Koa site immediately available to the world? That's where LocalTunnel comes in. LocalTunnel creates a URL for you, and redirects all requests sent to it to your locally listening listener.

The installation with NPM is as follows:

```
npm install -g localtunnel
```

Then start it with

```
lt --port 3000
```

It will give you a temporary à la Heroku URL, such as the one following:

<https://short-cougar-89.localtunnel.me>

Then you can tell the Koa app to register this as a webhook.

```
const Telegraf = require('telegraf')
const Koa = require('koa')
const koaBody = require('koa-body')

const bot = new Telegraf(process.env.BOT_TOKEN)
bot.command('image', (ctx) => ctx.replyWithPhoto({ url:
  'https://picsum.photos/200/300/?random' }))

bot.telegram.setWebhook('https://short-cougar-89.localtunnel.me')

const app = new Koa()
app.use(koaBody())
app.use((ctx, next) => ctx.method === 'POST' || ctx.url === '/
secret-path'
  ? bot.handleUpdate(ctx.request.body, ctx.response)
  : next()
)
app.listen(3000)
```

In Figure 11-20, we’re back to random images again!

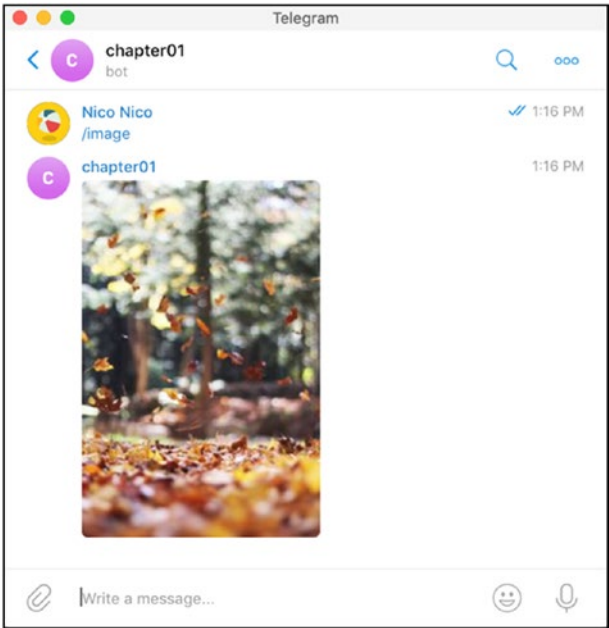


Figure 11-20. Random images from the Telegram bot with the webhook setup, running via localtunnel

CHAPTER 12

Week 12: Python

Always look on the bright side of life.

—Monty Python

Python is en route to becoming the most widely used programming language of the 21st century. Hobbyists love it, analysts love it, and even kids starting to code use Python these days.

With its simple syntax and zillions of available libraries, you can't go wrong with Python. Recently, many of these libraries have refocused on machine learning, AI, data science...so much so that Python, the language, does not require much of an introduction. That said, here's one (Figure 12-1).



Figure 12-1. *Hello, Python*

While I also like Python, a lot of the code is easy to write but harder to maintain. So, I usually switch to another language, but it's hard to match the number of libraries available with Python—OpenCV, TensorFlow...all come with a first-class Python wrapper.

Installation

3.6. What is it? 3.6 is the version of Python you want to install. Yes, the latest version is 3.7, but the TensorFlow library did not support version 3.7 at the time of writing, so you are better off using 3.6. If you really do not care about the TensorFlow samples, but I hope you do, you can stick to 3.7.

Python itself sometimes comes preinstalled and is already available. If not, or if the version does not match, or if you're a Windows lover, you can download the installer from the following Python download page:

<https://www.python.org/downloads/windows/>

The Python package manager, pip, will have to be installed as well. You can perform the installation for that by following the steps here.

<https://pip.pypa.io/en/stable/installing/>

Or, in short,

```
# download the get-pip.py file( manually if you don't have curl)
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
# execute it with the python executable
python get-pip.py
```

An alternative but recommended method is to install mode for Python and use a tool named pyenv. pyenv is a small tool that allows you to switch from one Python tree to another, meaning you can switch from one version of Python and its libraries to another.

<https://github.com/pyenv/pyenv#installation>

With pyenv installed, you can now switch from one version of Python to another in a convenient manner. To check the versions installed, use `pyenv versions`.

```
$ pyenv versions
system
* 3.6.0 (set by /Users/niko/.pyenv/version)
  3.7.0
  anaconda3-5.0.1
```

As shown in the preceding and following code snippets, we're supporting Python version 3.6 and pip version 18.0. Yes, 3.6, because, again, the TensorFlow packages were not available for Python 3.7 at the time of writing.

```
$ pip --version
pip 18.0 from /Users/niko/.pyenv/versions/3.7.0/lib/python3.7/
site-packages/pip (python 3.7)
```

But python2 is also perfectly usable, but not much tested in the scope of this book.

In a new folder for the forthcoming Python script, we can set up the `tasks.json` usable by Visual Studio Code.

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "echo",
      "type": "shell",
      "command": "python",
      "args": [
        "${file}"
      ],
    },
  ],
}
```



```

        "group": {
            "kind": "build",
            "isDefault": true
        }
    }
]
}

```

The plug-in to work with Python in Visual Studio Code is shown in Figure 12-2.

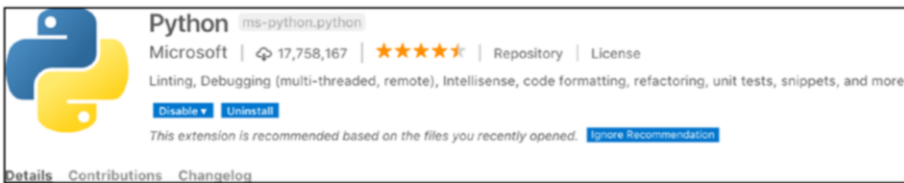


Figure 12-2. *Python plug-in for Visual Studio Code*

That's it for the setup. Let's move on to our first Python program.

A Few Python Programs

You've probably guessed it already, so there's no need to resist too much. Let's work on Fibonacci numbers in Python. What's great is that we can go through the examples one by one and learn (new) things about the language.

Fibonacci 1

This first implementation goes the recursive way. We define a function `fib`, with `def`, that calls itself.

Note that the indentation of any Python program is important. So, the `if`, `elif`, and `else` are all one tab in from the previous indentation, and the `return` statements are an additional one tab in from these. If you don't respect the indentation, the linter or the syntax checker of Visual Studio Code will show an error (Figure 12-3).

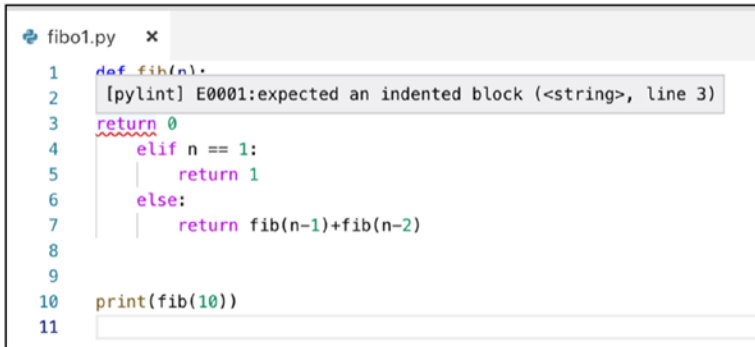


Figure 12-3. *Python is very strict with indentation*

With this in mind, here comes the first listing. The function is defined first, then the print statement, exactly two lines (!), after the function definition.

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1)+fib(n-2)

print(fib(10))
```

If you run this file with the usual Command/Ctrl+Shift+B build task of Visual Studio Code, you will see the output of the program in the console again (Figure 12-4).

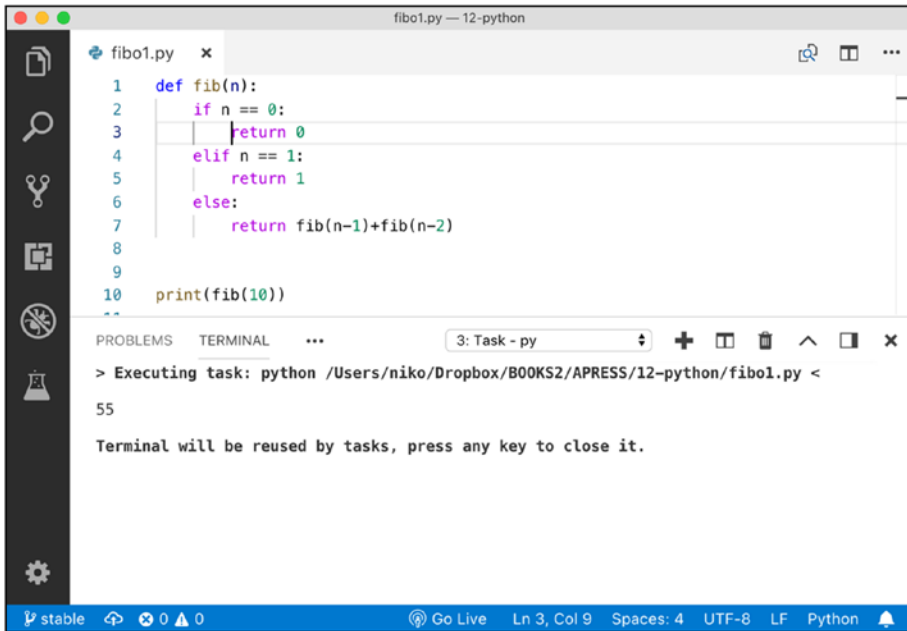


Figure 12-4. Executing the Python code in Visual Studio Code

Let's move on to the second implementation.

Fibonacci 2

This second implementation works with an internal, and simple, cache to compute the numbers. This time, we start by adding some documentation on the fastFib function, right after the def line.

Then we add a new parameter, memo, which will act as a cache. We add a default value to memo, the start of the Fibonacci sequence, in which fib[0]=1 and fib[1]=1.

Next, we just pass the memo around when calling fastFib recursively.

```

def fastFib(n, memo={0:1, 1:1}):
    """memoized recursive function, returns a Fibonacci number"""
    print('>', n, memo)

```

```

    if not n in memo:
        memo[n] = fastFib(n-1, memo) + fastFib(n-2, memo)
    return memo[n]

print(fastFib(5))

```

Because we have the print statements, when you execute the program, you can see the cache being filled along the way.

```

> 5 {0: 1, 1: 1}
> 4 {0: 1, 1: 1}
> 3 {0: 1, 1: 1}
> 2 {0: 1, 1: 1}
> 1 {0: 1, 1: 1}
> 0 {0: 1, 1: 1}
> 1 {0: 1, 1: 1, 2: 2}
> 2 {0: 1, 1: 1, 2: 2, 3: 3}
> 3 {0: 1, 1: 1, 2: 2, 3: 3, 4: 5}
8

```

Fibonacci 3

This next implementation is fun enough to write in here. We also use a cache—this time, global—to avoid passing it around, and we just append element to the ever-growing array. This also shows how to use a for loop with a range and, finally, how to join all the values of the array using join.

```

x = [1, 1] # this is an array

for i in range(2, 40):
    x.append(x[-1] + x[-2])

print(', '.join(str(y) for y in x))

```

Fibonacci 4

This new implementation presents tuples, with the `zip` function. The `zip` function returns a list of tuples `zip(fn1(), fn2())` that finishes whenever one of `fn1s` or `fn2s` stops returning values. `range` returns a list of numbers from 0 to 9, included. This is not so important, but the `print` statement in the loop formats the variables to keep the indentation with padding.

The difficult part yields, pun intended, the usage of the keyword `yields`. While `fib()` looks like a function, it is actually a generator.

What is a generator? `range(n)` is a generator. It returns values from 0 to `n`. A generator works like looping over a list.

To implement a never-ending list, a generator of constant values, you could use the following definition, in which you can picture constant as `(1,1,1,1 ...)`.

```
def constant():
    a = 1
    while True:
        yield a
```

`range()` itself could be simply reimplemented, as in the following snippet, in which we start with the value 0 and return to a list of `n` elements, the list of incremented `a`'s.

```
def myrange(n):
    a = 0
    while a < n:
        a=a+1
        yield a
```

Thus, `yield` is like returning a list of elements.

So, here comes the implementation of `fib` using `yield`. Note that the list is never-ending, so the loop is never-ending.

```
def fib():
    a, b = 0, 1
    while True:
        a, b = b, a + b
        yield a

for index, fibonacci_number in zip(range(20), fib()):
    print('{i:3}: {f:3}'.format(i=index, f=fibonacci_number))
```

The result of the execution is as follows:

```
0:   1
1:   1
2:   2
...
16: 1597
17: 2584
18: 4181
19: 6765
```

Fibonacci 5

The last Fibonacci example in this chapter uses an implementation based on squared roots. `sqrt` is the function used to compute square roots, and it is located in the `Math` package. So, we're using `import` to make it available to our program.

```
from math import sqrt

def F(n):
    return ((1+sqrt(5))**n-(1-sqrt(5))**n)/(2**n*sqrt(5))

print(F(100))
```

Alright, enough Fibonacci. We're done with the basics, so let's write some Telegram code with Python.

First Telegram with Python

To start playing with Telegram, we will use `telepot`, a Python wrapper for the Telegram API, which is located at <https://github.com/nickoala/telepot>.

To install a library for Python, we use `pip`, the Python package installer. To install any package, you use

```
pip install <packagename>
```

So, to install `telepot`, we will run the `pipinstall` with the `telepot` package name.

```
$ pip install telepot
```

```
Collecting telepot
```

```
...
```

```
Installing collected packages: telepot
```

You've now successfully installed `telepot-12.7`. You have seen it before, in Visual Studio Code, to pass a token to the process started by the build task. You can define the environment variable for the token in the integrated terminal of Visual Studio Code.

This can be done by setting up the `terminal.integrated.env`. [youros] key and adding the following into your user settings:

```
"terminal.integrated.env.osx": {
  "BOT_TOKEN":
"682216086:AAGeNyQ4jf9sAKuOvWJzKs45i4ui1VgWulk"
}
```

More on this and the integrated terminal settings can be found in the following Visual Studio Code documentation:

<https://code.visualstudio.com/docs/editor/integrated-terminal>

The first example will use `telepot` to retrieve information on the bot, using the `getMe` method call of Telegram. Again, we will run everything from the Visual Studio Code, as shown in Figure 12-5.

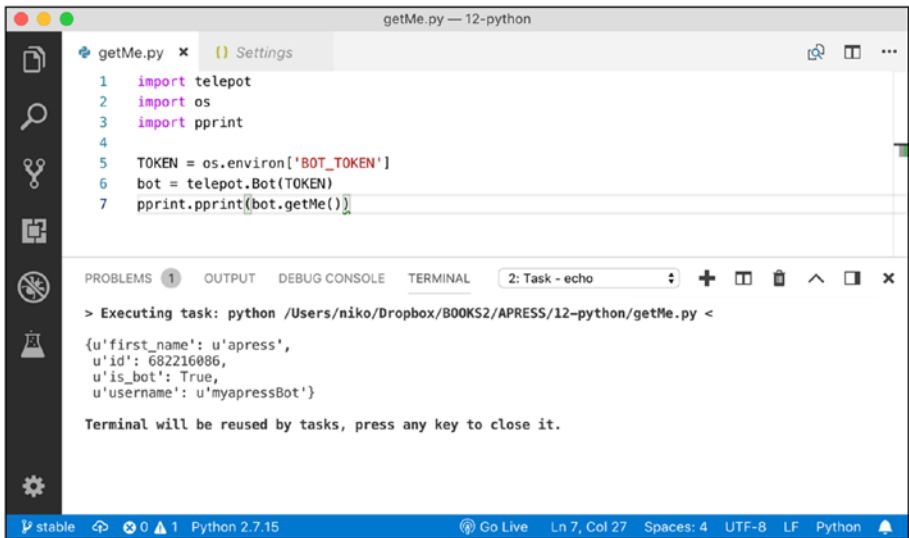


Figure 12-5. *Python bot from Visual Studio Code*

Alright, so to call the Telegram API, we need to

- Retrieve the token to use for the bot authentication from an environment variable
- Create an instance of the Telegram wrapper `telebot`, using that token
- Call `getMe`, to use the newly instantiated bot using the library

Note that we actually do not need to set up pooling or webhooks for this first program. Environment variables in Python can be retrieved using the `os` module. In the `os` module, there is a global array named `environ` that has all the variables passed to the process. We just saw how to define the `BOT_TOKEN` variable, so no surprise here.

A bot object can then be instantiated using `telepot.Bot` and the token. Finally, we can call any method of the API we want here, and because we are retrieving a Python data structure as return value, we use `pprint` from the `pprint` module, to display that as formatted data on the output of the console. This gives the first snippet, as follows:

```
import telepot
import os
import pprint

TOKEN = os.environ['BOT_TOKEN']
bot = telepot.Bot(TOKEN)
pprint.pprint(bot.getMe())
# empty line
```

And as you have seen in the screenshot, after executing the program, you'll get the info data related to the bot queried.

```
{u'first_name': u'apress',
 u'id': 682216086,
 u'is_bot': True,
 u'username': u'myapressBot'}
```

We're getting there, aren't we?

First Bot: Send a Random Photo

Taking on the image example from Chapter 11, we are now going to write some code to send a picture back to the chat, using the `sendPhoto` function from Python's `telepot`.

The bot object is instantiated in the same way, but now we will set it to listen to the messages using the `handle` callback function. At this point in the book, the `handle` function has few secrets. We retrieve the `chat_id` from the `update` and `msg`, going through a map object or, in Python terms, a dictionary.

```

import os
import random
import telepot
from telepot.loop import MessageLoop

def handle(msg):
    chat_id = msg['chat']['id']
    command = msg['text']

    if command == '/image':
        bot.sendPhoto(chat_id, 'https://picsum.
            photos/200/300/?random')

bot = telepot.Bot(os.environ["BOT_TOKEN"])
MessageLoop(bot, handle).run_forever()

```

Start the bot by calling `execute`, and now we can send the `/image` message to retrieve a random image, as was done with Node.js (Figure 12-6).

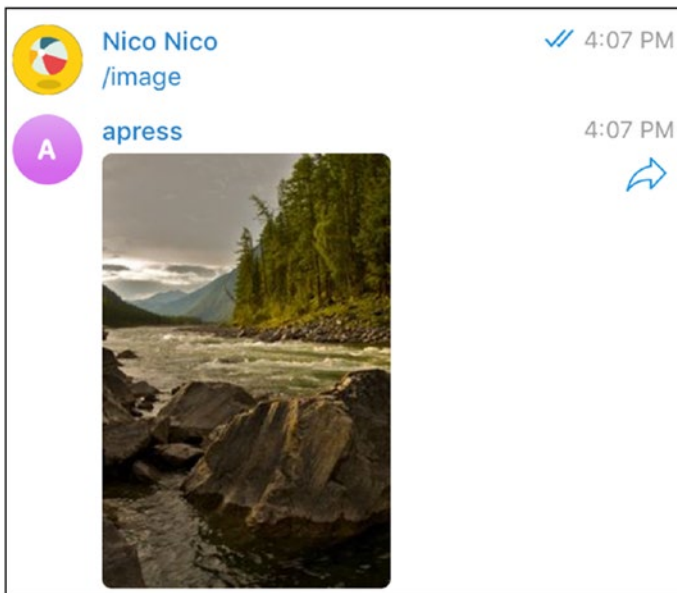


Figure 12-6. Sending a message and getting a photo from the Python `bot`. check.

First OpenCV Bot: Changing the Color Space of a Picture

OpenCV is one of the treats of coding with Python. Everything is ready for instant consumption, and in the 21st century, what could be better than instant consumption?

So, now, we will convert a picture from standard color to a black-and-white version. This is quite easy to do with `opencv`.

We'll start by installing the ready-to-use `opencv` and then move on to the code. The recommended means to install the library with Python is, of course, `pip`.

```
pip install opencv-python
```

The code itself will

- Create a temporary file, using the function `NamedTemporaryFile` from the `tempfile` module
- Read the `file_id` from the photo object contained in the Telegram update
- Download the file to a temporary file
- Open the file using `opencv`'s `imread` function
- Convert the color of the file using the `cvtColor` function
- Write the file to disk again, using `opencv`'s `imwrite` function
- Send the photo, using `telepot`'s `sendPhoto` function again, but this time on a file

Note, along the way, how we rename the default module from `cv2` to `cv`, to make it easier to understand that we are using version 3.

```
import os
import telepot
from telepot.loop import MessageLoop
import pprint
import cv2 as cv
import tempfile

def handle(msg):
    if msg["photo"]:
        chat_id = msg['chat']['id']
        f = tempfile.NamedTemporaryFile(delete=True).name+".png"
        photo = msg['photo'][-1]["file_id"]
        path = bot.getFile(photo)["file_path"]
        bot.sendMessage(chat_id, "Retrieving %s" % path)
        bot.download_file(photo, f)
        p = cv.imread(f)
        gray = cv.cvtColor(p, cv.COLOR_BGR2GRAY)
        cv.imwrite(f, gray)
        bot.sendPhoto(chat_id, open(f, 'rb'))
    else:
        print("no photo")

bot = telepot.Bot(os.environ["BOT_TOKEN"])
MessageLoop(bot, handle).run_forever()
```

The looping part is the same as in the previous example, and so now, if you start the bot and send a picture, you'll get something similar to Figure 12-7.

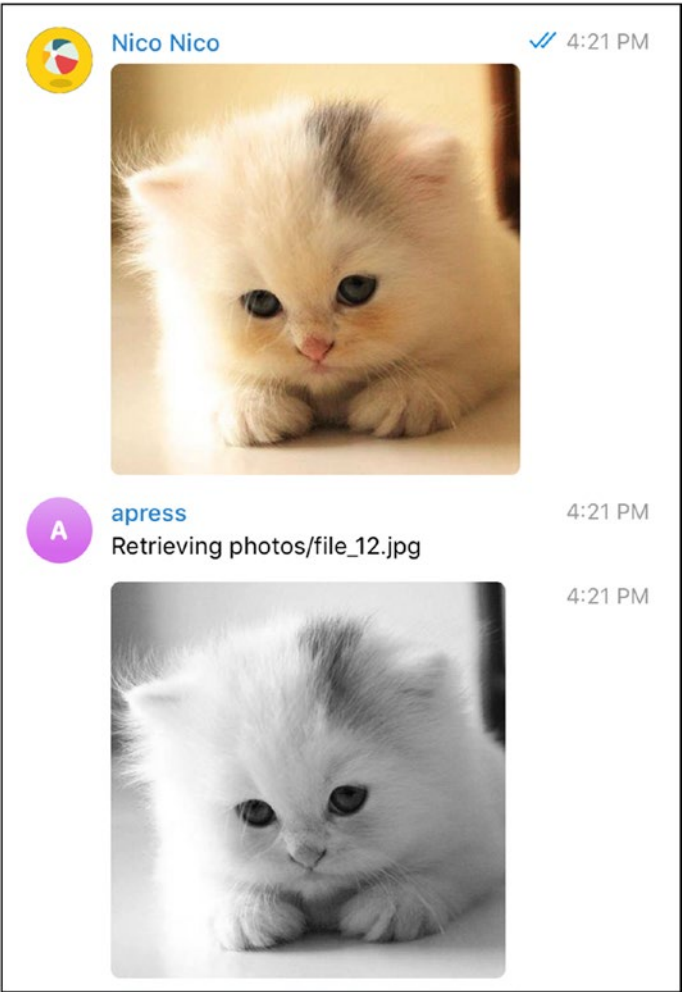


Figure 12-7. *Colored cat turned gray*

Second OpenCV Bot: Count Faces

Because we have opencv installed and ready to use, we can’t really get away with not doing an example that picks up and counts the number of faces in a picture, the most recognizable of all opencv examples. The flow is quite similar to that in the previous example of changing colors with opencv. This

time, we will use what opencv calls a classifier, and use it on a grayscale version of the sent picture. Then we will draw rectangles inside the picture, by looping over the faces found, using the opencv's rectangle function.

The detect function will do most of the opencv work, by detecting, counting, and drawing the faces. Note how the function returns two values simultaneously again, and how we are assigning them to two values in the main handle function.

```
import os
import telepot
from telepot.loop import MessageLoop
import pprint
import cv2 as cv
import tempfile

classifier = cv.CascadeClassifier(cv.data.harcascades +
    "haarcascade_frontalface_default.xml")

def detect(image):
    gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    faces = classifier.detectMultiScale(gray, 1.3, 5)
    count = 0
    for (x, y, w, h) in faces:
        count = count+1
        cv.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 5)
    return image, count

def handle(msg):
    if msg["photo"]:
        chat_id = msg['chat']['id']
        f = tempfile.NamedTemporaryFile(delete=True).name+".png"
        photo = msg['photo'][-1]["file_id"]
        path = bot.getFile(photo)["file_path"]
        bot.download_file(photo, f)
```

```
p = cv.imread(f)
hsv, l = detect(p)
cv.imwrite(f, hsv)
bot.sendMessage(chat_id, "found %i faces" % l)
bot.sendPhoto(chat_id, open(f, 'rb'))
else:
    print("no photo")

bot = telepot.Bot(os.environ["BOT_TOKEN"])
MessageLoop(bot, handle).run_forever()
```

I've tried this bot twice, and while not perfect, it can find quite a few party people in pictures, as shown in Figures 12-8 and 12-9.

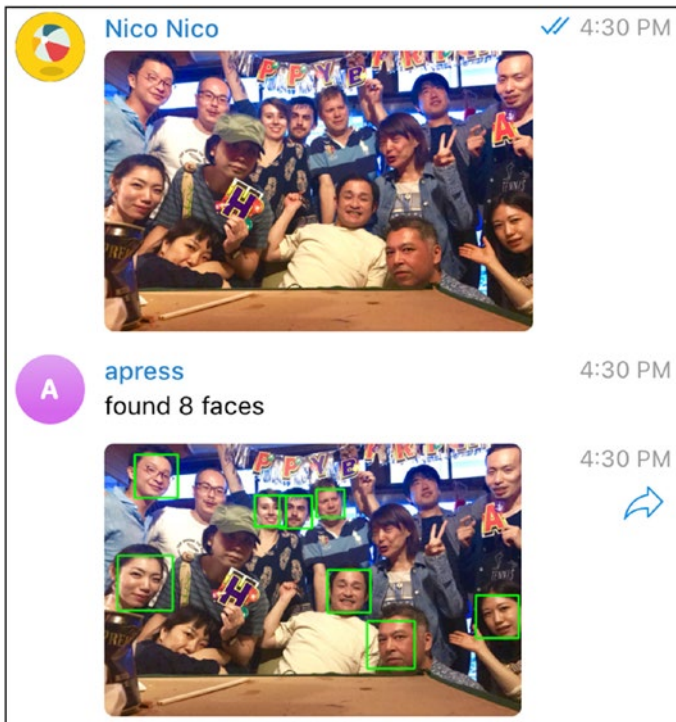


Figure 12-8. Eight party faces!

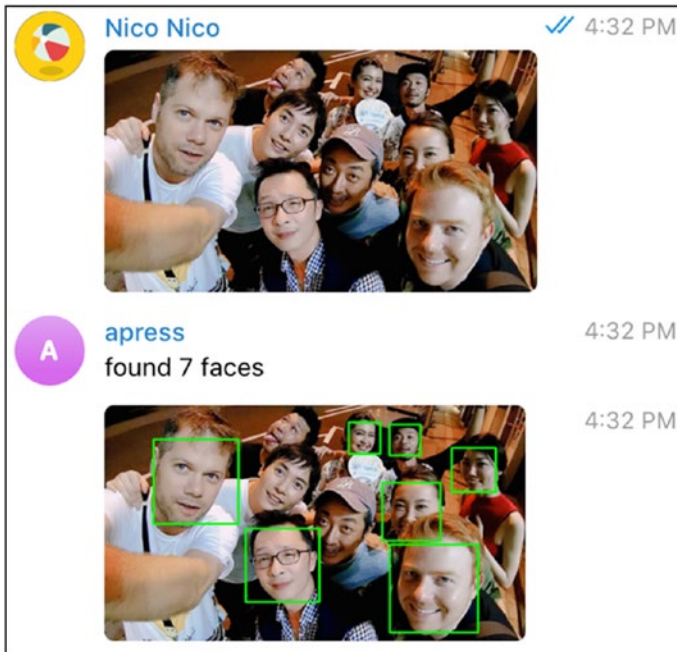


Figure 12-9. *Seven party faces!*

TensorFlow to Close the Show

You have probably heard about TensorFlow, one of the most famous machine learning frameworks. Creating your own models and understanding all the mathematics behind the library is completely beyond the scope of this book. But because TensorFlow comes as a Python first, we are going to see a short example of a Telegram bot that uses TensorFlow to identify the content of pictures.

With TensorFlow, you can train models, using a list of layers (steps), for something that looks like a memory, a network. You tell this memory to recognize that this X in input gives this Y in output, or X,Y in input gives Z in output, and, eventually, that a list of pixels is a cat or a dog.

You train this memory with a large data set of inputs and outputs, in this case, images, and then you can reuse this network in the wild. There are already pretrained models for us, so the following bot will use an existing set of pictures to train a model and reuse that TensorFlow model.

TensorFlow itself is again installed using pip.

```
pip install tensorflow
```

The model and the image classification code is located at

<https://github.com/tensorflow/models>

And, more specifically,

<https://github.com/tensorflow/models/tree/master/tutorials/image/imagenet>

To use this, we are going to first copy the `classify_image` code and apply a few modifications to the code, mostly to

- Call it as a library instead of the command line
- Return the prediction—actually, the top best prediction

The diff is provided in the example and as follows:

```
48c48,54
< FLAGS = None
---
> class ObjectDict(dict):
>     def __init__(self, *args, **kws):
>         super(ObjectDict, self).__init__(*args, **kws)
>         self.__dict__ = self
>
> FLAGS = ObjectDict({'model_dir': "/tmp/imagenet", 'num_top_
  predictions': 1})
>
```

```

164,167c171
<     for node_id in top_k:
<         human_string = node_lookup.id_to_string(node_id)
<         score = predictions[node_id]
<         print('%s (score = %.5f)' % (human_string, score))
---
>     return node_lookup.id_to_string(top_k[0])

```

The bot is following the opencv examples, in which we get a photo file and call the TensorFlow model on it, via the `classify_image.run_inference_on_image` function on the picture.

```

import os
import telepot
from telepot.loop import MessageLoop
import tempfile
import classify_image

def handle(msg):
    if msg["photo"]:
        chat_id = msg['chat']['id']
        f = tempfile.NamedTemporaryFile(delete=True).name+".png"
        photo = msg['photo'][-1]["file_id"]
        bot.download_file(photo, f)
        prediction = classify_image.run_inference_on_image(f)
        bot.sendMessage(chat_id, "I think this image is a %s" %
            prediction)
    else:
        print("no photo")

classify_image.maybe_download_and_extract()
bot = telepot.Bot(os.environ["BOT_TOKEN"])
MessageLoop(bot, handle).run_forever()

```

On running this bot, we can again send pictures and see what it is guessing. A cat is recognized (Figure 12-10).

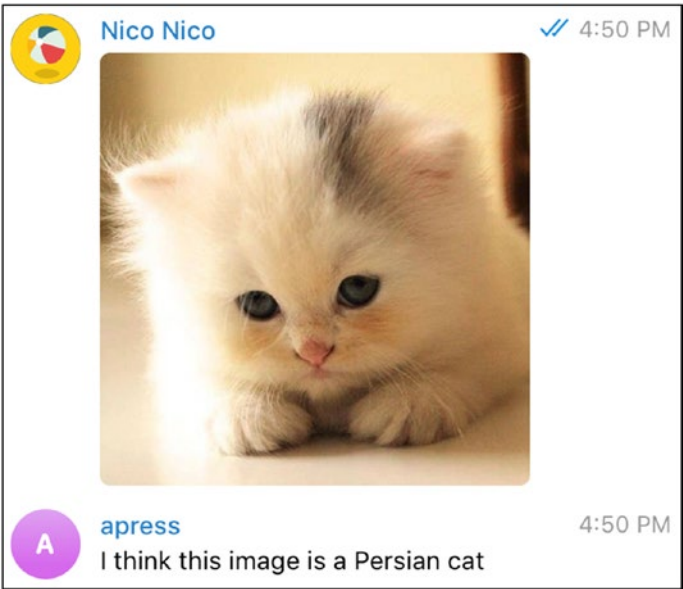


Figure 12-10. *The classification is even more detailed than “cat”; it’s “Persian cat”!*

Figures 12-11, 12-12, and 12-13 show pretty good image classification, which you could put to use!

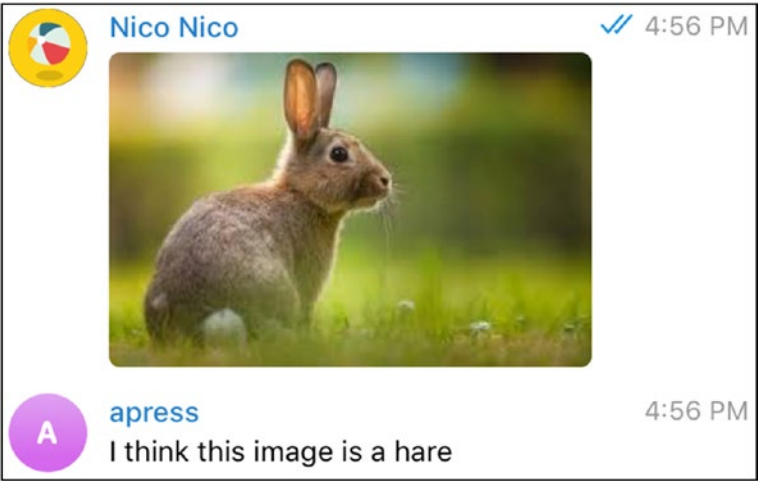


Figure 12-11. Rabbit?



Figure 12-12. A beagle! How did it guess that right?

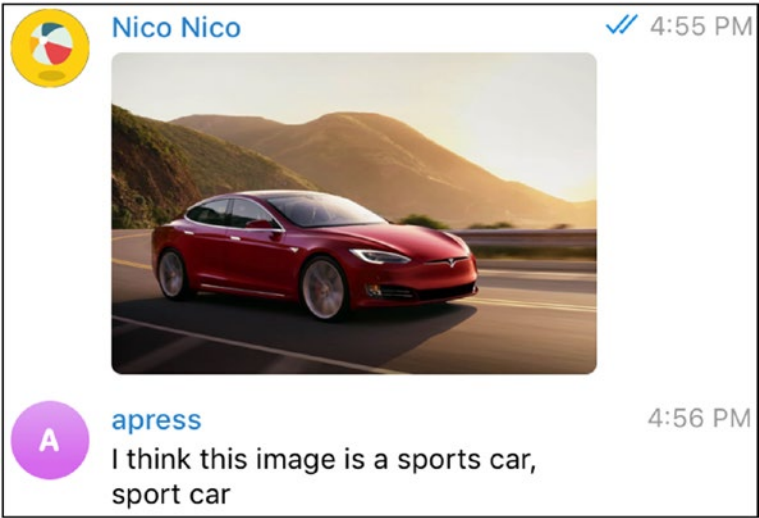


Figure 12-13. *Tesla!*

Now it’s time to hit the road with that Tesla and lead the new bot army that you have created. The fun is all yours from now on.

Index

A, B

BotFather

- edit bots, [3-4](#)
- finding, [2](#)
- new bot, [5](#)
- profile picture, [1-2](#)
- search box, [2](#)
- start charting, [3](#)
- token, [5-6](#)

C

C++

- cmake file
 - commands, [115](#)
 - compilation, [115-116](#)
 - download cat
 - image, [118](#)
 - folder structure, [113](#)
 - main.cpp, [114](#), [117-118](#)
 - plug-in, Visual Studio
 - Code, [116-117](#)
 - running, [118](#)
 - tgbot-cpp, [114](#)
 - URL with image, [118](#)
- description, [111](#)
- echo bot, [119-123](#)
- inline keyboard, [123-125](#)
- OpenCV bot (*see* OpenCV bot)

photo bot, [126-127](#)

tgbot-cpp, [111-112](#)

Clojure

- core.clj, [139](#), [143-146](#)
- debugging messages,
 - [140](#), [146-147](#)
- development, [135](#)
- Hello Leiningen, [136-137](#)
- inline handler, [149-150](#)
- installation instructions, [136](#)
- openjdk, [136](#)
- OpenWeather, [150-151](#), [153-154](#)
- Origami bot, [154-158](#)
- project.clj, [142-143](#)
- project files, [139](#)
- project structure, [138-139](#)
- REPL, [135](#), [137](#)
- reverse bot, [148-149](#)
- Telegram bot, [140](#)
- third-party library, [138](#)
- token, [146](#)
- Visual Studio Code, [141](#)

Crystal

- advantages, [37](#)
- benchmark code, [41](#)
- BotFather commands, [37-38](#)
- command bot
 - /animals command, [54-55](#)
 - hello command, [53-54](#)

INDEX

Crystal (*cont.*)

- send types of messages, [55](#)
- telegram_bot, [55](#)
- definition, [37](#)
- EchoBot, [50–52](#)
- embedded development
 - environment, [40](#)
- Fibonacci, [40](#)
- installation, [38–39](#)
- play command, [39](#)
- project creation
 - build command, [48](#)
 - init subcommand, [46](#)
 - mybot.cr file, [47](#)
 - printing version, [47](#)
 - shard.yml file, [48–49](#)
- Visual Studio Code
 - access settings, [45](#)
 - edit, json file, [45](#)
 - Gerardo Ortega, [43](#)
 - GitHub query, [46](#)
 - HTTP client and JSON
 - parser, [42](#)
 - Market place, [42](#)
 - requesting.cr, [41](#)
 - tasks.json file, [43–44](#)

dmd command, [87–88](#)

download page, [86–87](#)

dub

- dub.json file, [101](#)
- dub list, [102](#)
- dub run, [103](#)
- dubsearch, [104](#)
- installing, custom version, [102](#)
- logging, [105](#)
- source/app.d, [103](#)
- tasks.json file, [104](#)
- telegra, [100](#)

echo text messages, [105–107](#)

features, [85–86](#)

Fibonacci, [98–100](#)

functions, [109](#)

handleUpdate function, [107–108](#)

hello.d file, [87–88](#)

import statements, [87](#)

message, image, document, and
location, [108](#)

plug-in, [88–89](#)

sorting arrays, [95–97](#)

source files, [88, 90](#)

tasks.json file, [89–90](#)

Telegram bots, [100](#)

vibe.d, [86](#)

D

D language

- concurrency
 - shared state, [94–95](#)
 - std.concurrency, [91–92](#)
 - threadState, [92–93](#)

E

Elixir

- Erlang, [201](#)
- Fibonacci, [222–224](#)
- file_path, [219](#)

- getChat, [217–218](#)
- getFile, [218](#)
- getMe, [216–217](#)
- installation
 - iex, [203–204](#)
 - mix, [204–205](#)
 - running iex with
 - mix, [205–206](#)
 - tools, [202](#)
- logo, [202](#)
- mix project
 - config.exs, [207](#)
 - dependencies, [209–213](#), [215–216](#)
 - hex.pm, [213–214](#)
 - mix.exs, [208–209](#)
 - telegrambox.ex, [212–213](#)
- sendPhoto, [220–221](#)
- system command, [219](#)
- Telegram.Bot library, [221–222](#)

F

- Fibonacci
 - Elixir, [222–224](#)
 - D language, [98–100](#)
 - Go program, [190–192](#)
 - Python
 - def, [250](#)
 - fastFib, [252–253](#)
 - for loop, [253](#)
 - indentation, [251](#)
 - print statement, [251](#)
 - range(), [254](#)

- sqrt, [255](#)
- Visual Studio Code, [251–252](#)
- yield, [254](#)
- zip, [254](#)
- Rust, [64–65](#)

G, H, I

- Go
 - build command, [187–189](#)
 - description, [181](#)
 - download, [182](#)
 - Fibonacci, [190–192](#)
 - file.txt, [185](#)
 - first bot
 - custom data type/struct, [194](#)
 - familiar APIs and
 - constructs, [193–194](#)
 - /hello command, [195](#)
 - hello go, [196](#)
 - main handlers, [196](#)
 - starting, [196](#)
 - telebot, [193](#)
 - telegram, [193](#)
 - go1, [188–189](#)
 - if statement, [188](#)
 - logo, [181](#)
 - packages, [182–183](#)
 - platforms, [182](#)
 - plug-in, Visual Studio
 - Code, [183–184](#)
 - program structure, [184–185](#)
 - ReadFile, [187, 188](#)
 - reading.go file, [185–186](#)

Go (*cont.*)

- sending pictures, [197–199](#)
- Visual Studio Code plug-in, [185–186](#)

J

Java bot

- code, [166–167](#)
- installation
 - build.gradle file, [161–164](#)
 - Gradle web site, [159](#)
 - project structure, [161](#)
 - version 4.10.2, [160](#)
 - Visual Studio Code, [164–165](#)

Organize Imports, [165–166](#)

permissions

- payment settings, [170–171](#)
- Stripe dashboard, [172](#)
- Stripe setup, [172](#)
- Stripe Test Bot, [173](#)

resources/token file, [165](#)

sending messages, [167–169](#)

sending photo, [169–170](#)

SendInvoice message, [173](#)

- payment complete, [178](#)
- payment process, [176](#)
- query synchronously/
asynchronously, [174](#)
- shipment options, [175](#)
- shipping details, [174](#)
- shipping query, [174](#)
- Stripe test logs, [179](#)
- transaction validation, [177](#)

K, L, M

Koa application

- creating, [231](#)
- home page, [230](#)
- publishing, [232–234](#)
- running, [231–232](#)
- samples, [231](#)

N

Nim

- asyncdispatch module, [31](#)
- cats and dogs, [34–36](#)
- first chatbot, “hello”, [31–32](#)
- goodmorning.nim file, [23](#)
- hello nim world, [20–22](#)
- HTTP client
 - failed execution, [24](#)
 - Python-like syntax, [23](#)
 - running, [24](#)
 - SSL support, [25](#)
- installation, [17](#)
- packages with Nimble, [29–31](#)
- plug-in, Visual Studio Code, [18–19](#)
- replying message, [33–34](#)
- .vscode/tasks.json file
 - arbitrary external
command, [26](#)
 - build task, [27](#)
 - content of, [28](#)
 - creating, [25](#)
 - default build task, [25](#)
 - echo, [27](#)

- getsome, [29](#)
 - Hello, [26-27](#)
- Node.js, RunKit, [225](#)
 - creating account, [226-228](#)
 - Fibonacci notebook, [229-230](#)
 - home page, [226](#)
 - Koa (*see* Koa application)
 - localtunnel, [244-246](#)
 - saying hello, [229](#)
 - setting up, [243-244](#)
 - Telegraf library
 - GitHub URL, [239](#)
 - image-to-chat, [239-240](#)
 - RegExp, inline keyboards, and embedded emojis, [240-242](#)
 - webhooks, [225](#)
 - BOT_TOKEN variable, [238](#)
 - first bot, [236](#)
 - HTTP POST request, [236](#)
 - settings page, [237](#)
 - setWebhook function, [235](#)
 - start chatting, [238](#)
 - Telegraf, [237](#)
 - vs.* polling, [234](#)

O

- OpenCV bot
 - api.telegramorg/file/bot, [131](#)
 - applyOpenCV, [132-133](#)
 - cat.jpg, [130-131](#)
 - CMakeLists.txt file, [128-129, 131](#)
 - colored and gray cats, [133-134](#)

- compilation, [133](#)
 - handling, chat message, [133](#)
 - imread and imwrite, [130](#)
 - installation, [113](#)
 - Mat object, [129](#)
 - Telegram architecture, [131](#)
- OpenWeather
 - API token, [150-151](#)
 - HTTP query, [151](#)
 - morse/telegram handler, [153](#)
 - registration, [151](#)
 - result, [154](#)
 - retrieve function, [153](#)
 - Tokyo with curl/httpie, [151-152](#)
- Origami bot
 - api/download-file, [156](#)
 - apply-cv function, [157](#)
 - file path, [156](#)
 - OpenCV transformation, [157-158](#)
 - project.clj, [154](#)
 - Telegram static files, [155](#)

P, Q

- Photo bot, [126-127](#)
- Python
 - Fibonacci (*see* Fibonacci)
 - installation
 - download page, [248](#)
 - package manager, pip, [248](#)
 - pyenv, [249](#)
 - tasks.json, [249](#)
 - visual studio code plug-in, [250](#)

INDEX

Python (*cont.*)

- libraries, [248](#)

OpenCV

- changing colors,
picture, [260–262](#)
- count faces, [262–265](#)

- sendPhoto, [258–259](#)

telepot

- environment variables, [257](#)
- executing program, [258](#)
- getMe method, [257](#)
- pipinstall, [256](#)
- pprint module, [258](#)
- user settings, [256](#)
- Visual Studio Code, [256–257](#)

TensorFlow model

- beagle, [269](#)
- classify_image code, [266](#)
- diff, [266](#)
- image classification code, [266](#)
- installation, [266](#)
- Persian cat, [268](#)
- photo file, [267](#)
- rabbit, [269](#)
- Tesla, [270](#)

R, S

- Read-eval-print-loop (REPL), [135](#),
[137](#), [202](#), [205](#), [211](#)

Ruby

- BotFather (*see* BotFather)
- download and install, [7](#)

- ruby and gem versions, [7–8](#)

Telegram bot

- conversation, [11](#)
- execution, [11](#)
- fields, [14](#)
- first message, [12](#)
- first reply, [15](#)
- gems, [8–10](#)
- libraries, [8](#)
- Linux/OS X, [11](#)
- message format, [14](#)
- received message, [13](#)
- ruby command, [13](#)
- to_yaml, [12](#)
- writing code, [10](#)

- text editor, [8](#)

Ruby Version Manager (RVM), [58](#)

Rust

- advantages, [57](#)
- cargo
 - binary application, [67](#)
 - binary targets, [72–73](#)
 - bin flag, [71](#)
 - Cargo.toml, [67](#)
 - chrono, [68](#)
 - dates and time zones, [67](#)
 - main.rs file, [70–71](#)
 - project structure, [67](#)
 - running code, [69–70](#)
 - subcommands, [66](#)
- chain messages, [82–83](#)
- compilation, [84](#)

Fibonacci, [64–65](#)

Hello Rust, [62–64](#)

installation

 build tools, Visual

 Studio, [60–61](#)

 C++ tooling, [59–60](#)

 links, [58](#)

 macOS/Linux, [58–59](#)

 rustupshow command, [61](#)

 Windows, [58–59](#)

text_reply, [74–77](#)

Tokyo, live location, [77–81](#)

T, U, V

tgbot-cpp, [111–112](#)

W, X, Y, Z

Windows Subsystem for Linux
(WSL), [38](#)