

## CHAPTER 5

# Deploying Your Chatbot

In this chapter we'll learn how to deploy our chatbots on the web. There are various ways and channels through which one can deploy or expose their chatbot web application to the outside world. For an example, we can expose our HoroscopeBot with NLU and dialog model on Facebook and Slack as they provide you a user interface already. You may also want to have your own web app that exactly runs on your own server. We will also explore how to deploy a chatbot on our own servers using our own user interface by the end of this chapter.

## First Steps

The first step is to create a copy of your existing chatbot you built in Chapter 4 and make a new copy so that you have a backup with you. Since, we will be doing some changes by adding some new code, let's keep both the projects separate.

So, I created a new folder called “**Chapter V**” and pasted my horoscope\_bot folder in there. So, now all my model files, datasets, and code files are copied, which I can directly use for deployment.

## Rasa's Credential Management

Rasa provides a way to manage all your credentials at one place. You may have one single model, but you may want it to be deployed on various other platforms like Facebook, Slack, Telegram, etc. All of these third-party platforms need some credentials to be used while connecting. These credentials are stored in a YAML file called `credentials.yml`.

Let's create a file named `credentials.yml` file in our project directory horoscope\_bot folder and add our Facebook credentials there. If you don't know how to get that then just create the file for now, and in next section of this chapter you can find the steps to get Facebook credentials.

Contents of `credentials.yml` will look like this:

```
facebook:
  verify: "horoscope-bot"
  secret: "bfe5a34a8903e745e32asd18"
  page-access-token: "HPaCABJJ1JmQ7qDedQKdjEAAb04iJKr7H9nx4rEBAAuFk4Q3g
PQcNTowtD"
```

These credentials are dummy credentials; the length of the token or secret and characters type may differ for your Facebook app.

If you are working on a big project where you are integrating your chatbot on various platforms and you want to make the project more maintainable, then it's best to make use of `credentials.yml`. I highly recommend you maintain a `credentials.yml` if you are a business and trying to build a bot that works on various platforms like Facebook, Slack, Twitter, Telegram, or your own website in the same way. Managing keys and secrets becomes easier in this case.

A good way to manage application-level secret keys is to store the keys as environment variables and write the code to read the values of secret keys or any other sensitive information from the operating system's environment itself. Remember, it's never a good idea to keep any kind of keys inside your code.

You can also create a `dot(.)env` file on your server and read keys from this file, which is not tracked anywhere in your code repository.

For the sake of simplicity, we are going to use the access keys and secret keys in our standalone scripts for deployment. We are going to make it simple to understand so that you are first able to build the bot, then you can try to scale it, and most importantly you can think about security-level issues.

In case you need to deploy your bot on multiple platforms and want to use `credentials.yml` to maintain different credentials, then you can use it by passing an extra argument. For an example to use the above credentials file named `credentials.yml` while running `rasa core` you can use the below command.

```
python -m rasa_core.run -d models/dialogue -u models/nlu/current
--port 5002 --credentials credentials.yml
```

It is good to know for bigger enterprise-level chatbots development, but as discussed, we'll be using the credentials directly in our script in our upcoming examples.

## Deploying the Chatbot on Facebook

In this section we are first going to deploy our chatbot using Heroku in cloud. Heroku is a platform-as-a-service (PaaS) that enables developers to build, run, and operate applications entirely in the cloud. The benefit of Heroku is that we can easily get our app running on https without much pain. We don't need to go and buy SSL certificates while we are learning and testing our chatbots. The reason why https is required is because some platforms like Facebook do not allow developers to use non-https urls as callback URLs.

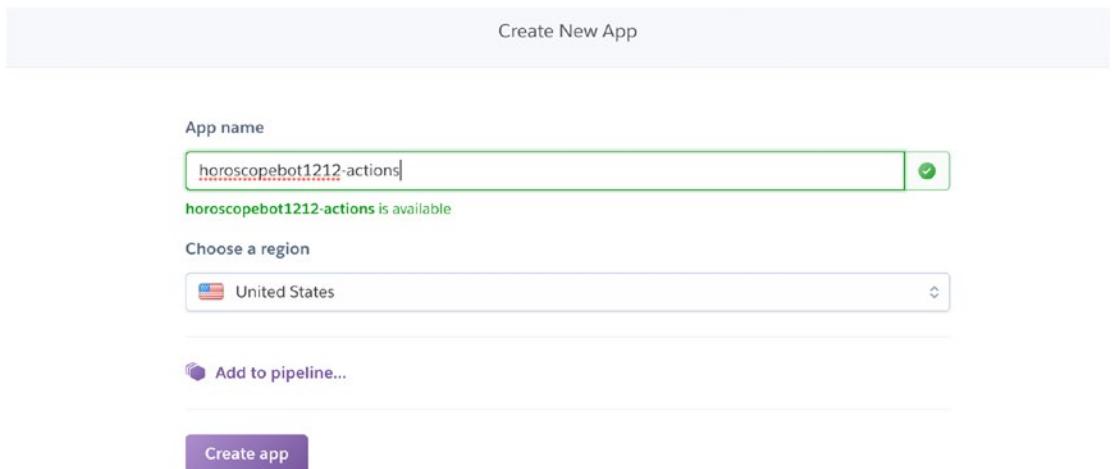
We'll be following a set of steps one-by-one to successfully deploy our chatbot as a web service in cloud. Once we have done that successfully, it will be way easier to integrate it with different platforms like Slack, Telegram, etc. So, let's start.

### Creating an App on Heroku

Let's get started:

Sign up on Heroku, create an app, and name it something-actions, as this is going to be our actions server app. Have a look at the screenshot in Figure 5-1, where you can give a unique name for your actions server, which should be available on Heroku. Once that name seems available, you can click on the Create app button to create the actions server app.

Feel free to name it anything you want if your name is not available, but always try to give meaningful names.



Create New App

App name

horoscopebot1212-actions

horoscopebot1212-actions is available

Choose a region

United States

Add to pipeline...

Create app

**Figure 5-1.** Creating the action server app on Heroku with the name *horoscopebot1212-actions*

## Setting Up Heroku on Your Local System

Install Heroku CLI on your local operating system. Refer to this link: <https://devcenter.heroku.com/articles/heroku-cli>.

If you are on macOS, use the following command:

```
brew install heroku/brew/heroku
```

## Creating and Setting Up an App at Facebook

To be able to deploy our chatbot on Facebook, first we need to have credentials of the Facebook app. In order to get the Facebook credentials, we need to set up a Facebook app and a page, like we did in one of our Chapter 3.

1. Go to <https://developers.facebook.com/> and create an app if you don't have one already. We created one for our OnlineEatsBot; now we'll create one for HoroscopeBot. Enter the details and click on Create App ID. Check Figure 5-2 to see how to enter the display name of your bot and your contact email.

### Create a New App ID

Get started integrating Facebook into your app or website

Display Name

Contact Email

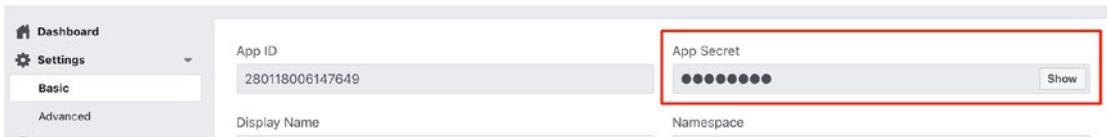
By proceeding, you agree to the [Facebook Platform Policies](#)

Cancel

Create App ID

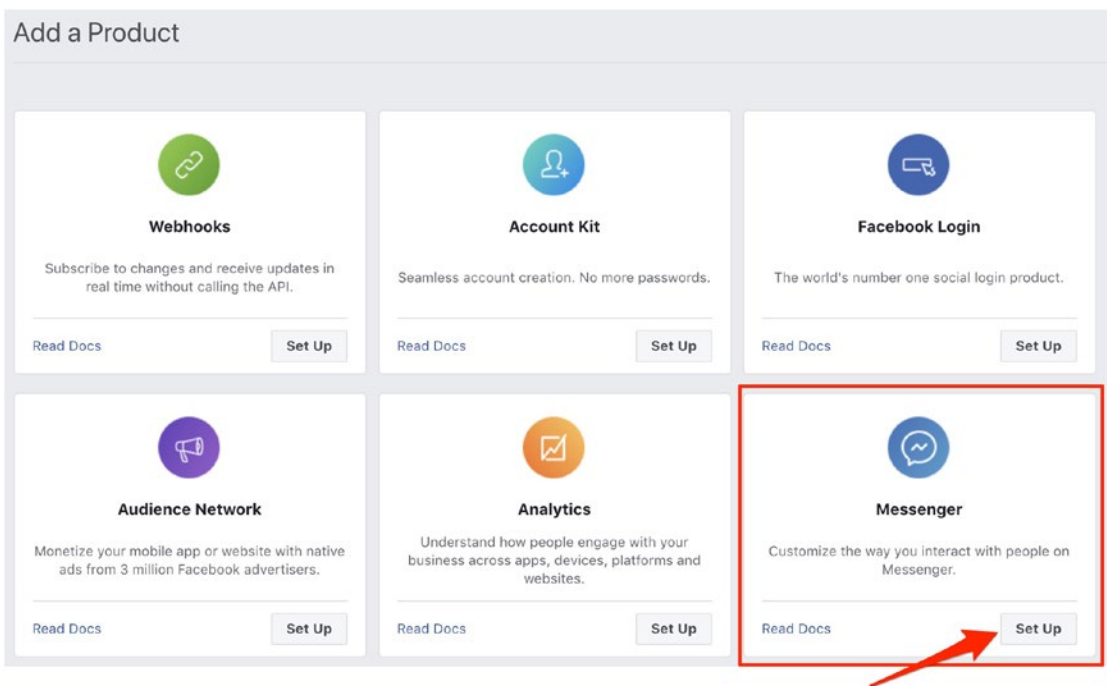
**Figure 5-2.** *Creating app on Facebook for developers*

2. Once your app is created, go to Basic under Settings, and click on the Show button under App Secret. This is your fb\_secret. Refer Figure 5-3 to see where exactly you will get your fb\_secret key.



**Figure 5-3.** Getting App Secret from Facebook's app

3. Go to the dashboard for the app and scroll down to “Add a Product.” Click Add Product and then add Messenger (click on Set Up). Check Figure 5-4.

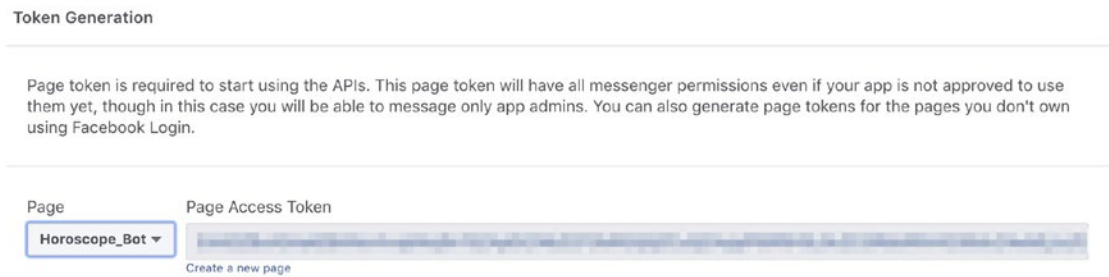


**Figure 5-4.** Adding Messenger as product to Facebook app

4. Under settings for Messenger, when you scroll down to the Token Generation section you will get a link to create a new page for your app. If you don't have a page already, then create it or choose a page from the “Select a page” dropdown. The “Page Access Token” is your fb\_access\_token here. Refer Figure 5-5.

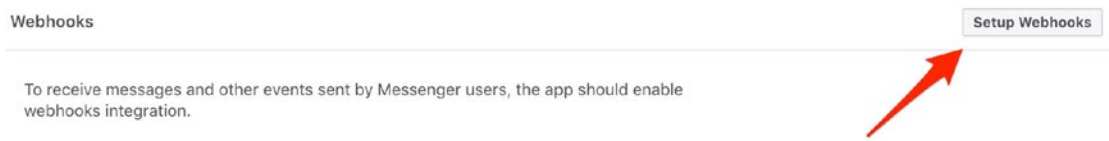
You can go to the following link to create a brand new page for your bot project:

<https://www.facebook.com/pages/creation/>



**Figure 5-5.** *Generating token for Facebook Messenger App*

- 5. Right after the Token Generation section, under Webhooks, click on “Setup Webhooks.” Refer Figure 5-6.



**Figure 5-6.** *Setting up Facebook Webhooks*

- 6. Next, choose a verify token, which we’ll need to use later. The verify token can be any random string. This will be your fb\_verify. Check Figure 5-7 to understand where to add the verification token in facebook app. Now, leave the callback URL section blank as it is. Don’t close the browser; just leave it—we’ll come back here again.

**New Page Subscription** ✕

Callback URL

Validation requests and Webhook notifications for this object will be sent to this URL.

Verify Token

some-secret-token

Subscription Fields

<input type="checkbox"/> messages	<input type="checkbox"/> messaging_postbacks	<input type="checkbox"/> messaging_optins
<input type="checkbox"/> message_deliveries	<input type="checkbox"/> message_reads	<input type="checkbox"/> messaging_payments
<input type="checkbox"/> messaging_pre_checkouts	<input type="checkbox"/> messaging_checkout_updates	<input type="checkbox"/> messaging_account_linking
<input type="checkbox"/> messaging_referrals	<input type="checkbox"/> message_echoes	<input type="checkbox"/> messaging_game_plays
<input type="checkbox"/> standby	<input type="checkbox"/> messaging_handovers	<input type="checkbox"/> messaging_policy_enforcement

[Learn more](#)

Cancel Verify and Save

**Figure 5-7.** Adding verify token to Facebook webhook setup

7. Keep `fb_verify`, `fb_secret` and `fb_access_token` handy to connect your bot to Facebook.

## Creating and Deploying Rasa Actions Server App on Heroku

In this step we are going to use our actions Heroku app for our Rasa action's server. We need to have two different applications, as we cannot run two web applications in a single Heroku app. Go to your command line and execute the following set of commands from your project directory as directed.

1. Create a new folder called `actions_app` and get into the directory:
 

```
mkdir actions_app
cd actions_app
```
2. Copy your `actions.py` from main project directory to `actions_app` directory.

3. Create a requirements.txt file with the following contents.  
requirements.txt will tell the Heroku app to install the packages with their versions.

```

rasa-core-sdk==0.11.0
requests==2.18.4

```

4. Create a file named Procfile with the following contents. Procfile is the file for which Heroku understands what to do in order to crank up the applications.

```

web: python -m rasa_core_sdk.endpoint --actions actions
--port $PORT

```

- a) Run the below set of commands:

```

$ heroku login
$ git init
$ heroku git:remote -a <your-heroku-app-name>
$ heroku buildpacks:set heroku/python
$ heroku config:set PORT=5055
$ git add .
$ git commit -am "deploy my bot"
$ git push heroku master

```

After the last command, Heroku will install all our packages needed as per the requirements.txt file. If your app is successfully deployed, you should be getting logs similar to the following:

```

remote:
remote: -----> Discovering process types
remote:      Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:      Done: 48.3M
remote: -----> Launching...
remote:      Released v4
remote:      https://horoscopebot1212-actions.herokuapp.com/ deployed to
remote:      Heroku

```

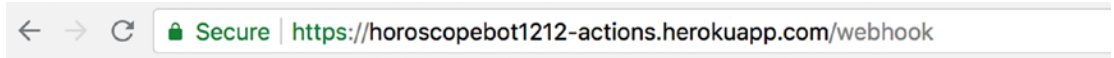


```
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/horoscopebot1212-actions.git
* [new branch]      master -> master
```

At this point, we will just verify whether our app is responding to public requests. In order to do that let's hit the app url appended by "webhook."

App url in my case is <https://horoscopebot1212-actions.herokuapp.com/> so I'll go and check if my action's server is responding.

I go to this url <https://horoscopebot1212-actions.herokuapp.com/webhook>, and as expected, it comes back saying method not allowed, like in Figure 5-8, which is totally fine and means that app is responding correctly as per the user request.



## Method Not Allowed

The method is not allowed for the requested URL.

**Figure 5-8.** *Verifying action server endpoint*

## Creating Rasa Chatbot API App

In this step we will follow some steps and commands similar to what we just did, but this is a new app we'll create that will be our main app for dialog management. So, let's do it. First come back to the main project directory (i.e., in `horoscope_bot`) and create a file name (**Procfile**) and add the following contents to it:

```
web: python -m spacy download en && python facebook.py
```

## Creating a Standalone Script for Facebook Messenger Chatbot

Create a file name `facebook.py` in the same project directory. The contents of the Python file should be as given here:

```
from rasa_core.channels.facebook import FacebookInput
from rasa_core.agent import Agent
from rasa_core.interpreter import RasaNLUIInterpreter
import os
from rasa_core.utils import EndpointConfig

# load your trained agent
interpreter = RasaNLUIInterpreter("models/nlu/default/horoscopebot/")
MODEL_PATH = "models/dialog"
action_endpoint = EndpointConfig(url="https://horoscopebot1212-actions.
herokuapp.com/webhook")

agent = Agent.load(MODEL_PATH, interpreter=interpreter)

input_channel = FacebookInput(
    fb_verify="YOUR_FB_VERIFY_TOKEN",
    # you need tell facebook this token, to confirm your URL
    fb_secret="YOUR_FB_SECRET", # your app secret
    fb_access_token="YOUR_FB_ACCESS_TOKEN"
    # token for the page you subscribed to
)

# set serve_forever=False if you want to keep the server running
s = agent.handle_channels([input_channel], int(os.environ.get('PORT',
5004))), serve_forever=True)
```

Make sure to replace the `fb_verify`, `fb_secret`, and `fb_access_token` variable values in this code with what we kept in Step 3.

Create a new `requirements.txt` file and add all the packages and their versions needed for this project. My `requirements.txt` looks like the following; for your project the requirements may differ, but these requirements should be fine if you are following the same bot example in this chapter.

```

rasa-core==0.11.1
rasa-core-sdk==0.11.0
rasa-nlu==0.13.2
gunicorn==19.9.0
requests==2.18.4
spacy==2.0.11
sklearn-crfsuite==0.3.6

```

to install our packages in the server.

Now let's create a new app again in Heroku like we did earlier. Go to your Heroku dashboard and create a new app as shown in Figure 5-9.

**Figure 5-9.** *Creating dialog management app in Heroku*

Once you have created the app, you can now go to your project root directory and run the following set of commands from the command line in your project folder:

```

$ git init
$ heroku git:remote -a <your-heroku-app-name>
$ heroku buildpacks:set heroku/python
$ heroku config:set PORT=5004
$ git add .
$ git commit -am "deploy my bot"
$ git push heroku master

```

If you get a runtime error after deployment, it may look like below this

ValueError: You may be trying to read with Python 3 a joblib pickle generated with Python 2. This feature is not supported by joblib.

This will mainly happen if you are using Python 2.x version. Heroku, by default, uses Python 3.x version. So, in case you want to use Python 2.x, you need to perform the steps below to resolve the above error. Change Python 3.6 to Python-2.7.15. to do this.

Create a file runtime.txt under root app directory of your project. Open the runtime.txt and add the following line python-2.7.15 and then save it. Heroku will use the aforementioned Python version only to build your project.

Once the successful deployment is done, you will see a url Heroku gives saying the app deployed to <url>.

```
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing requirements with pip
remote:
remote: -----> Discovering process types
remote:      Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:      Done: 254M
remote: -----> Launching...
remote:      Released v17
remote:      https://horoscopebot1212.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/horoscopebot1212.git
   cd3eb1b..c0e081d  master -> master
```

This deployment is going to take a little time, so be patient as much as you can—you are about to see magic. If you didn't get any error messages, then you have successfully deployed your chatbot to Heroku on cloud to make it work with Facebook Messenger. Let's verify if it works.

## Verifying the Deployment of Our Dialog Management App on Heroku

To verify if our dialog management app is successfully deployed on Heroku, we'll be doing the following steps.

1. Take the url given by Heroku and append this endpoint to it: `/webhooks/facebook/webhook?hub.verify_token=YOUR_FB_VERIFY_TOKEN&hub.challenge=successfully_verified`. Make sure to use the correct verify token that you used for webhooks settings in Facebook. The complete url for me looks like the below: [https://horoscopebot1212.herokuapp.com/webhooks/facebook/webhook?hub.verify\\_token=my-secret-verify-token&hub.challenge=success](https://horoscopebot1212.herokuapp.com/webhooks/facebook/webhook?hub.verify_token=my-secret-verify-token&hub.challenge=success).
2. Go to the browser and paste the entire url, and it should return your hub.challenge value back if your hub.verify\_token is correct. Your complete url will look like the following: [https://horoscopebot1212.herokuapp.com/webhooks/facebook/webhook?hub.verify\\_token=YOUR\\_FB\\_VERIFY\\_TOKEN&hub.challenge=successssfully\\_verified](https://horoscopebot1212.herokuapp.com/webhooks/facebook/webhook?hub.verify_token=YOUR_FB_VERIFY_TOKEN&hub.challenge=successssfully_verified). If you get the message successssfully\_verified in the browser then your app is successfully deployed and working.

## Integrating Webhook With Facebook

Now let's go back to our Facebook app configuration. We'll go to the point where we left off in Step 3 and add our callback URL. Make sure to check the **messages** in Subscription Fields. Check Figure 5-10 for reference.

New Page Subscription

×

Callback URL

https://horoscopebot1212.herokuapp.com/webhooks/facebook/webhook

Verify Token

Subscription Fields

☒ messages

☐ message\_deliveries

☐ messaging\_pre\_checkouts

☐ messaging\_referrals

☐ standby

☐ messaging\_postbacks

☐ message\_reads

☐ messaging\_checkout\_updates

☐ message\_echoes

☐ messaging\_handovers

☐ messaging\_optins

☐ messaging\_payments

☐ messaging\_account\_linking

☐ messaging\_game\_plays

☐ messaging\_policy\_enforcement

[Learn more](#)

Cancel

Verify and Save

Figure 5-10. Facebook Messenger webhooks configuration

Click on “Verify and Save.” Facebook will match the verify token using the above url, meaning the server, or say our app will only respond to requests that have the correct verify token. Once the verify token matches, the webhook subscription will get activated for our app.

Next, select a page to which you can subscribe your webhook to the page events, under the Webhooks section on the page. Click on subscribe (see Figure 5-11).

Webhooks

Edit events

To receive messages and other events sent by Messenger users, the app should enable webhooks integration.

Selected events: messages

Complete

Select a page to subscribe your webhook to the page events

The app is not subscribed to any pages

Horoscope\_Bot

Subscribe

Figure 5-11. Subscribe webhook to Facebook page events

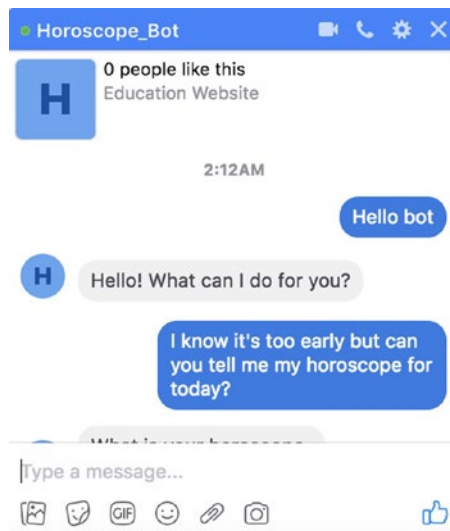
All done! Time to test our Horoscope Bot on Facebook.

## Post-Deployment Verification: Facebook Chatbot

In normal software development scenarios, people build software, test it, and then deploy and do PDV (post-deployment verification). We'll also be doing something similar, and we'll do a PDV for our chatbot after our successful deployment on Facebook Messenger. This is important because, as you learned, the chatbot has a part where it needs to connect to the actions server to respond to some intent requests of the user. PDV is like a sanity testing to see that the health of the app is good overall. If you are building a bot that uses 10 to 15 different vendors' APIs, then it's a must to check all scenarios where your bot hits the action server and uses the API to return data back to the user.

So, go to your messenger app or Facebook in your computer browser and search for your bot to start talking.

Figures 5-12.1 through 5-12.3 show what my Horoscope Bot does and tells me.



**Figure 5-12.1.** *Horoscope\_Bot Facebook*

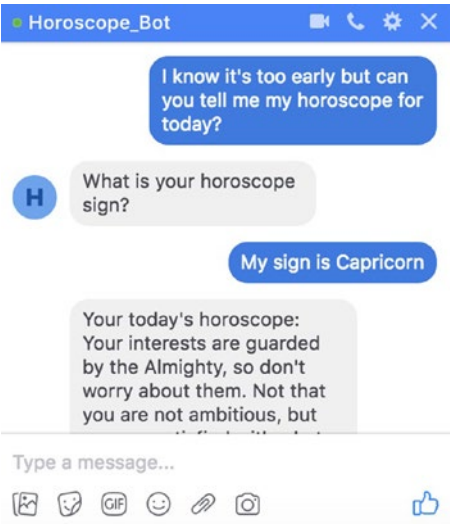


Figure 5-12.2. Horoscope\_Bot Facebook

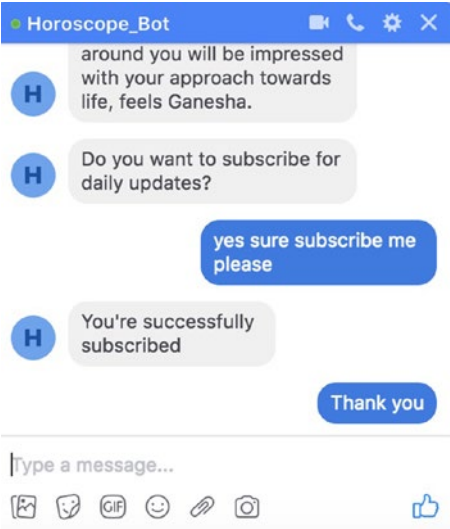


Figure 5-12.3. Horoscope\_Bot Facebook

Voila! Our first in-house-built chatbot app is deployed on the web and can be accessed via the Facebook Messenger platform. So, go ahead and share it with your family, friends, colleagues, and the entire world.



## Deploying the Chatbot on Slack

In this section we'll be deploying our chatbot to Slack. Slack is a team collaboration tool widely popular among developers and corporations. If you are not a social media person, then you might need Slack's help to talk to your chatbot using an interface. So, let's dive into building our first in-house Slack chatbot.

In order to deploy our Horoscope Chatbot to slack, we'll be writing a standalone script just like we did in the case of Facebook.

## Creating a Standalone Script for Slack Chatbot

Create a new file called `slack.py` in your project's directory. Contents of the file `slack.py` will look like the following:

```
from rasa_core.channels.slack import SlackInput
from rasa_core.agent import Agent
from rasa_core.interpreter import RasaNLUInterpreter
import os
from rasa_core.utils import EndpointConfig

# load your trained agent
interpreter = RasaNLUInterpreter("models/nlu/default/horoscopebot/")
MODEL_PATH = "models/dialogue"
action_endpoint = EndpointConfig(url="https://horoscopebot1212-actions.
herokuapp.com/webhook")

agent = Agent.load(MODEL_PATH, interpreter=interpreter, action_
endpoint=action_endpoint)

input_channel = SlackInput(
    slack_token="YOUR_SLACK_TOKEN",
    # this is the `bot_user_oauth_access_token`
    slack_channel="YOUR_SLACK_CHANNEL"
    # the name of your channel to which the bot posts (optional)
)

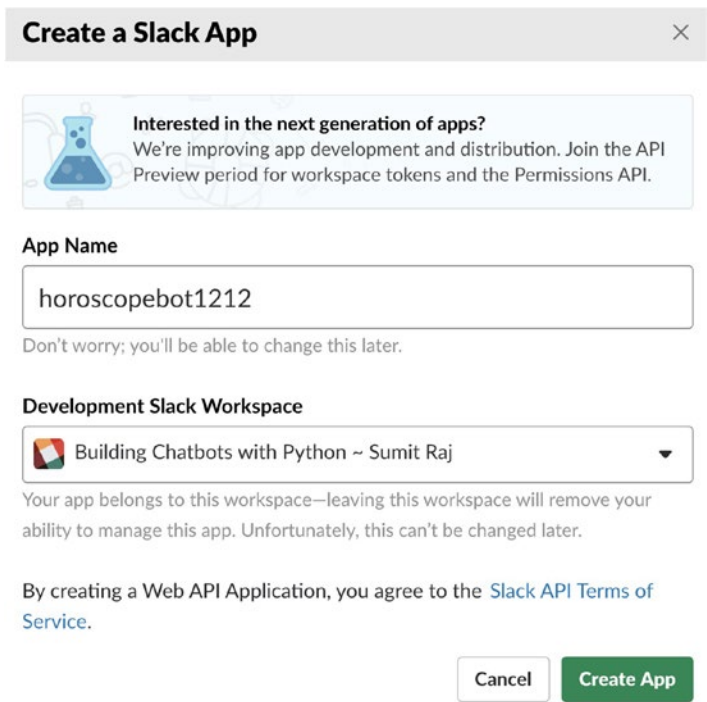
# set serve_forever=False if you want to keep the server running
s = agent.handle_channels([input_channel], int(os.environ.get('PORT',
5004))), serve_forever=True)
```

The primary difference between `facebook.py` and `slack.py` is the `input_channel` object that we create. Rasa provides various in-built channels like Facebook, Slack, Mattermost, Telegram, Twilio, RocketChat, and Microsoft Bot Framework, which we can use directly to deploy the same bot on various channels easily.

As you can see, we need to add a `slack_token` and `slack_channel` to be added into our script. As we had to create a Facebook application on Facebook’s developer platform, similarly we’ll have to create an app on Slack as well.

Let’s do this step by step:

1. Got to this url <https://api.slack.com/slack-apps> and click on the button “Create App.” Refer Figure 5-13.



**Create a Slack App**

Interested in the next generation of apps?  
We're improving app development and distribution. Join the API Preview period for workspace tokens and the Permissions API.

**App Name**

horoscopebot1212

Don't worry; you'll be able to change this later.

**Development Slack Workspace**

Building Chatbots with Python ~ Sumit Raj

Your app belongs to this workspace—leaving this workspace will remove your ability to manage this app. Unfortunately, this can't be changed later.

By creating a Web API Application, you agree to the [Slack API Terms of Service](#).

Cancel Create App

**Figure 5-13.** *Creating an application in Slack*

2. The next step is to create a Bot User. To create a bot user, click on **Bots** under “Add features and functionality.” In the new page you will get an option to “Add a Bot User.”. Check Figure 5-14 to see how to add details and add a bot user.

## Bot User

You can bundle a bot user with your app to interact with users in a more conversational manner. Learn more about [how bot users work](#).

### Display name

Names must be shorter than 80 characters, and can't use punctuation (other than apostrophes and periods).

### Default username

If this username isn't available on any workspace that tries to install it, we will slightly change it to make it work. Usernames must be all lowercase. They cannot be longer than 21 characters and can only contain letters, numbers, periods, hyphens, and underscores.

### Always Show My Bot as Online

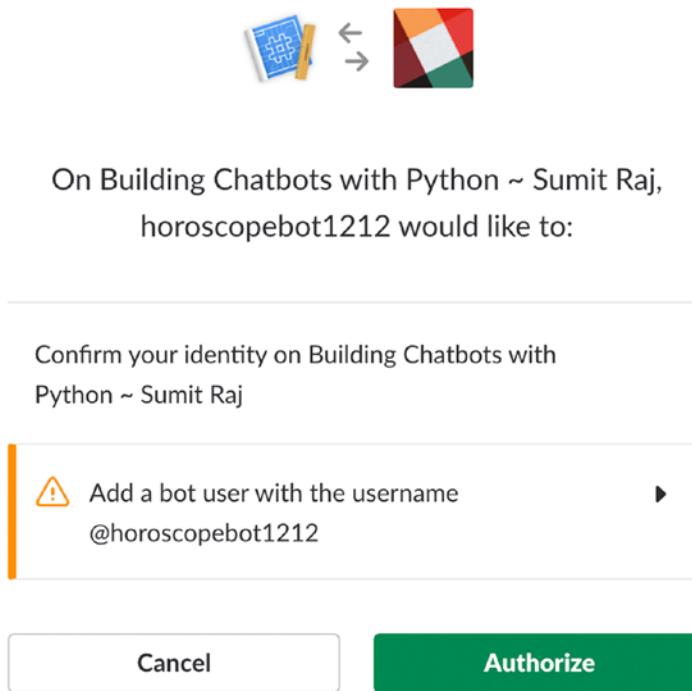
When this is off, Slack automatically displays whether your bot is online based on usage of the RTM API.

On ☒

Add Bot User

**Figure 5-14.** Giving a name to your bot on Slack

3. Fill in the details as per the chatbot you are building. Display name can be anything that you like; default username has to be unique; you can let it be as it is. Toggling the last option to always show my bot as online is to always show the bot to be available as a user. This is what chatbots are meant for—humans can't be available 24/7, but chatbots can, so we turn on this feature. Make sure you click on save changes.
4. Go back to the “Basic Information” tab. Click “Install your app to your workspace.” The app will ask to confirm the identity. Please authorize it like you do for any other app. Check Figure 5-15 which shows how the authorization would look like.



**Figure 5-15.** *Authorizing your Slack app*

You will find the Bots and Permissions tab under “Add features and functionality” with a green checkmark, which means our bot and app are well-integrated. This is a sign that we are doing good so far.

5. Go to your OAuth & Permissions section, and copy **Bot User OAuth Access Token**.
6. Paste the copied token into our Python script `slack.py`. Give a channel name as you like. If you want your bot to post to a channel, then you can give a channel name. I have given `@slackbot`. If you do not set the `slack_channel` keyword argument, messages will be delivered back to the user who sent them.

## Editing your Procfile

In this step we won't be creating any new Procfile, as we are working with the same codebase. We'll be changing our existing Procfile to the following to make it work for our slack bot. So, we just change the name of the script file from `facebook.py` to `slack.py` so that Heroku uses the given file to start up the application.

```
web: python -m spacy download en && python slack.py
```

## Final Deployment of Slack Bot to Heroku

To finally deploy our new Slack bot to Heroku, we'll be running a similar set of Heroku commands from command line to deploy our application.

```
$ git init
$ heroku git:remote -a <your-heroku-app-name>
$ git add .
$ git commit -am "deploy my bot"
$ git push heroku master
```

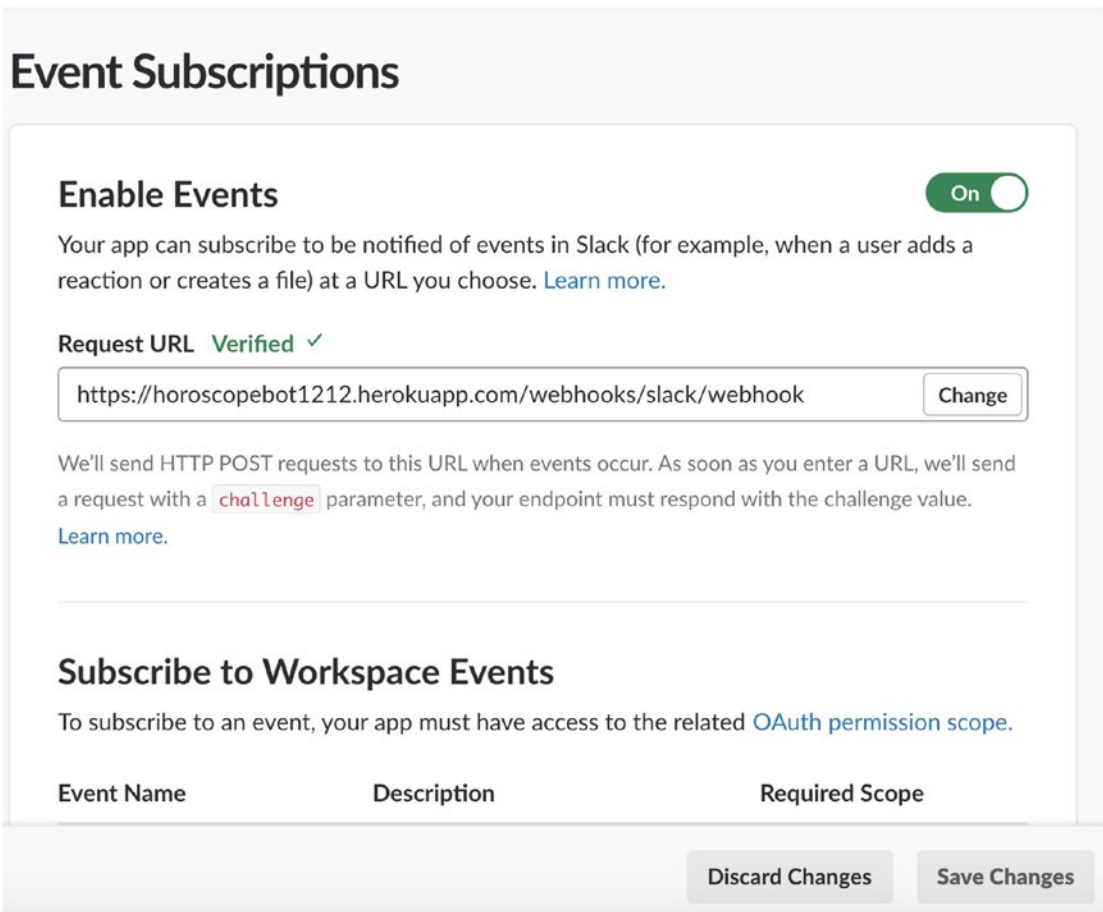
## Subscribe to Slack Events

Now, click on the “**Event Subscriptions**” tab and activate the events subscriptions functionality by toggling the button on the screen. Enter the Heroku app's webhook url for Slack.

If your app was deployed on Heroku properly with the modified Procfile, your webhook url for Slack will be `app_url + /webhooks/slack/webhook`, which looks like the following:

<https://horoscopebot1212.herokuapp.com/webhooks/slack/webhook>

You will see a verified tick mark after Slack sends a HTTP POST request to the above URL with a challenge parameter, and our endpoint must respond with the challenge value. This is similar to what we discussed while building Facebook chatbot's secret token. Check Figure 5-16 to understand more.





**Figure 5-16.** *Activate event subscriptions for your bot*

## Subscribe to Bot Events

In this step we'll simply scroll down the events subscriptions page and go to the "Subscribe to Bot Events" section, and click on "Add Bot User Event." Take reference of Figure 5-17 to understand where to navigate.

## Subscribe to Bot Events

Bot users can subscribe to events related to the channels and conversations they're part of.

Event Name	Description	
<a href="#">app_mention</a>	Subscribe to only the message events that mention your app or bot	
<a href="#">message.im</a>	A message was posted in a direct message channel	

Add Bot User Event

**Figure 5-17.** *Subscribe to bot events*

Subscribe to bot events is nothing but declaring the events for which the bot has to reply. We'll only be demonstrating two scenarios here: first, when somebody mentions the bot's name (i.e., **app\_mention**), and second, when somebody directly sends the bot the message (i.e., **message.im**).

Now, click on save changes, and you are done. It's time to test our Slack chatbot like we did in the previous section for Facebook.

## Post-Deployment Verification: Slack Bot

Let's go to our workspace we used to create the app, and under Apps on the left side you will find your bot. Try to talk to it and see if it does well. My bot does pretty well to give me my horoscope for today, which is good to read. If you have not been able to come till this point then check Figure 5-18 to see how my Slack bot responds.

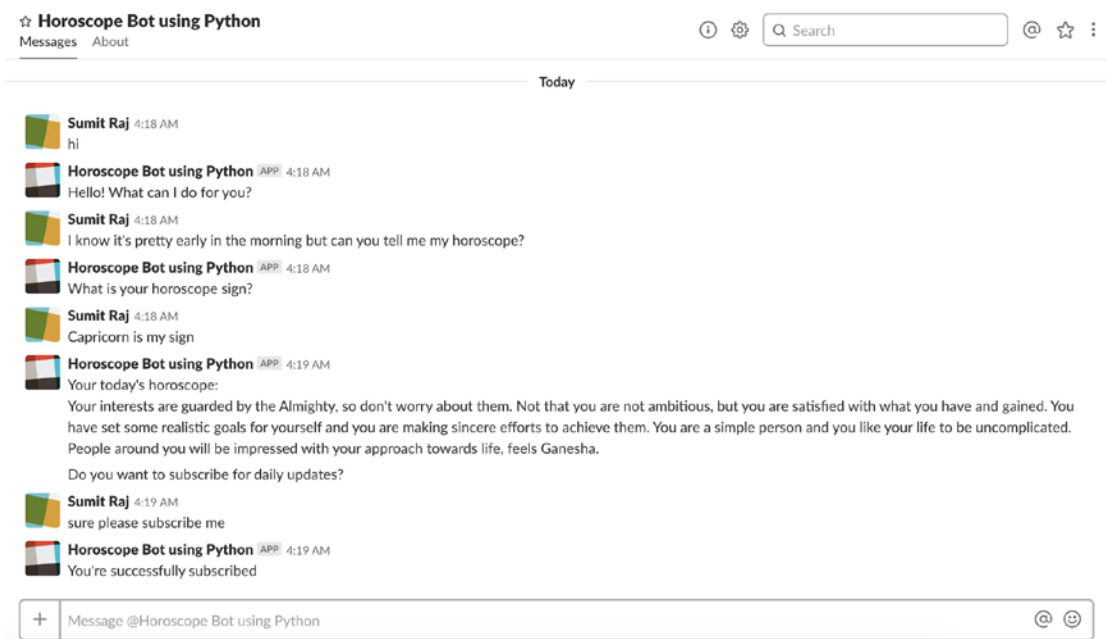


Figure 5-18. Testing the Slack chatbot

So, folks we are done with our Slack Bot. In the next section, we'll be deploying our bot to our own UI. Building your own UI may require some front-end skills, but do not worry—we have plans for that.

## Deploying the Chatbot on Your Own

Well the caption sounds cool, doesn't it? So far, we have been deploying our chatbot on the web using Facebook or Slack, or we could have used Telegram, etc., but now it's time to deploy everything on our own—our own servers, our own data, and our own configured model using our own user interface. If you are an organization or a budding entrepreneur, you may have your bot idea on Facebook, Twitter, or Slack, but you would always want it to be working on your own websites as well so that your brand value increases more and more as the user base increases.

In this section we are going to use all our hard work so far to finally build a chatbot, which is fully functional and independent from any third-party API calls or tools like Dialogflow, wit.ai, Watson, etc. You will have all the control in the world to tweak your chatbot the way you want and, most importantly, scale it the way you want to millions of people easily.



So, let's get started.

The first step is to ensure that two of our apps that we have deployed so far in the previous sections are up and running. You already know how to do a basic sanity check. You always need your dialog manager app and actions app to be running to use your chatbot model on any platform.

Now, in the same project directory where we have been creating `facebook.py` and `slack.py`, we'll create a new file called `myown_chatbot.py`. The scripts created previously, like `facebook.py` and `slack.py`, are standalone scripts that we created so that we can just tell Heroku in a command which script to run to crank up the application. Now, we are creating our own script that will expose the request/response between a user and the chatbot via REST APIs.

Deploying your own chatbot has two parts. In the first part, we'll be writing a script to create a custom channel and deploy it as REST APIs. In the second part, we need our own UI, because so far, we have been using Facebook's and Slack's chat screens for the conversations.

## Writing a Script for Your Own Chatbot Channel

This script is similar to what we have learned and written so far, but it needs us to override some of the existing methods of `rasa_core` so that we can define our own rule for API authentication. I have done the basic string check for token verification in the following code. This is not suggested for production-level systems, so make sure to write that part with care if you are building a chatbot for larger systems.

Create a new file named `myown_chatbot.py` and add the following contents to it:

```
import os

from rasa_core.channels.rasa_chat import RasaChatInput
from rasa_core.agent import Agent
from rasa_core.interpreter import RasaNLUInterpreter
from rasa_core.utils import EndpointConfig

# load your trained agent
interpreter = RasaNLUInterpreter("models/nlu/default/horoscopebot/")
MODEL_PATH = "models/dialogue"
action_endpoint = EndpointConfig(url="https://horoscopebot1212-actions.
herokuapp.com/webhook")
```

```
agent = Agent.load(MODEL_PATH, interpreter=interpreter, action_endpoint=
action_endpoint)
```

```
class MyNewInput(RasaChatInput):
    def _check_token(self, token):
        if token == 'mysecret':
            return {'username': 1234}
        else:
            print("Failed to check token: {}".format(token))
            return None
```

```
input_channel = MyNewInput(url='https://horoscopebot1212.herokuapp.com')
# set serve_forever=False if you want to keep the server running
s = agent.handle_channels([input_channel], int(os.environ.get('PORT',
5004))), serve_forever=True)
```

A couple of points to be noted here:

- `_check_token` method in `rasa_core` basically looks like the following, which makes an API call to get the user object. This primarily does the job of user-level authentication/authentication. In the earlier overridden method, we have kept it simple to make it work and understand its usage.

```
def _check_token(self, token):
    url = "{}users/me".format(self.base_url)
    headers = {"Authorization": token}
    logger.debug("Requesting user information from auth server {}. "
                "{}".format(url))
    result = requests.get(url,
                          headers=headers,
                          timeout=DEFAULT_REQUEST_TIMEOUT)

    if result.status_code == 200:
        return result.json()
    else:
        logger.info("Failed to check token: {}. "
                    "Content: {}".format(token, request.data))
        return None
```

- Using Rasa's own `_check_token` method may require you to write one API or web service that accepts the request and returns the response in the specified way.
- Make sure to change the action's server endpoint to your own url.
- Remember the `mysecret` string in the code will be used to make the API calls later.

## Writing the Procfile and Deploying to the Web

By this time you must be pretty familiar with creating Procfiles for Heroku deployment. We'll again use our existing Procfile and do the modifications there to deploy our API-based chatbot to the web. Feel free to create a new Procfile after creating backups of existing ones.

The following is what my Procfile content looks like:

```
web: python -m spacy download en && python myown_chatbot.py
```

Once you are done, just execute the next set of commands that we already learned while deploying our Facebook Messenger and Slack Bot.

```
$ git init
$ heroku git:remote -a <your-heroku-app-name>
$ git add .
$ git commit -am "deploy my bot"
$ git push heroku master
```

After the last command you will get some logs from Heroku related to deployment version, changes made to the app, etc.

## Verifying Your Chatbot APIs

After getting a successful message for deployment, let's test if our chatbot APIs are working or not. In order to quickly do the sanity testing, hit the following url:

```
<your-basic-app-url>+/webhooks/rasa/
```

**Example:**

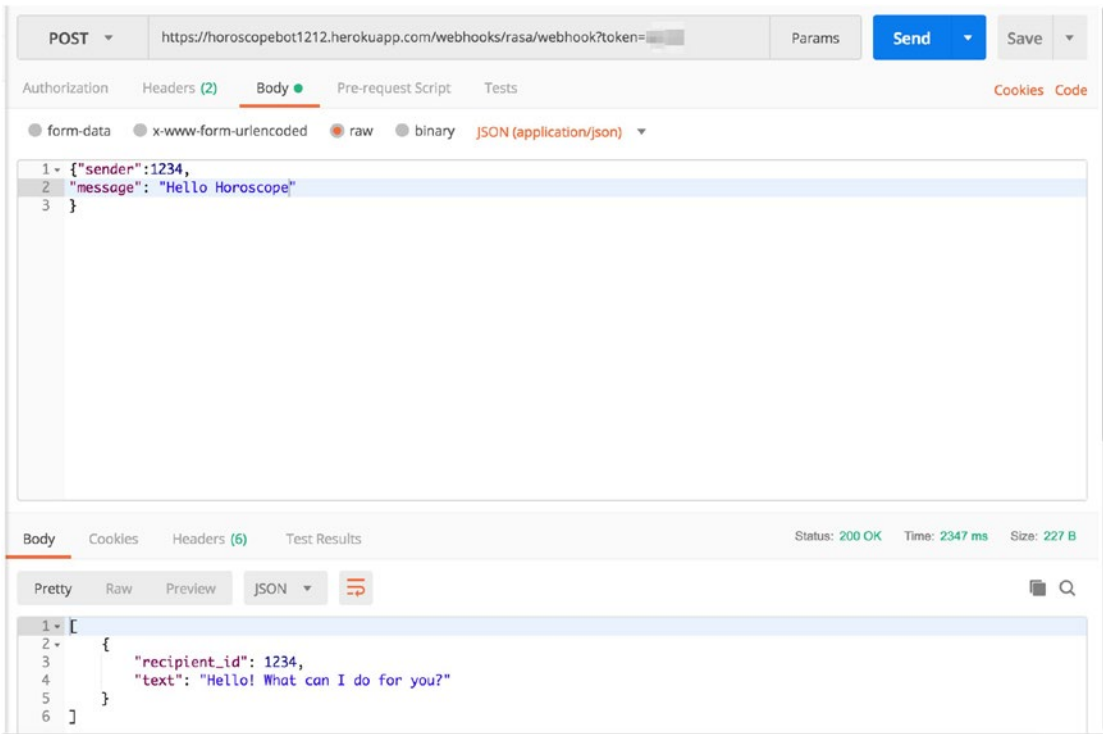
<https://horoscopebot1212.herokuapp.com/webhooks/rasa/>

Opening this url in the browser should give you a response like the following. If it gives you a status of “ok,” then you are good to go—just relax, sit back, and debug.

```
{ "status": "ok" }
```

Sometimes, just this verification may not be enough, so let’s test it for real by trying to check if the chatbot is working to identify the intents and giving responses based on that.

I will be using the POSTMAN tool (POSTMAN is a very nice GUI-based tool to do API testing). You can use any tool you are comfortable with. We are just going to test one of the intents our chatbot is supposed to understand and respond to. I tested the greetings intent and it worked like a charm. The bot came back with an expected response as shown in Figure 5-19.



**Figure 5-19.** Testing chatbot API in POSTMAN

## Creating the Chatbot UI

As we discussed earlier, as a part of the second step, we need to have our own UI to give a user-friendly place for conversation between the chatbot and the user. If you are a front-end developer or you have a front-end developer on the team, you can easily give the chatbot APIs we built to this point, and the front-end team should be easily able to integrate this with their chatbot UI. They can make use of regular HTTP calls to consume these APIs. Websockets are a better way of doing chatbots, but that is not under the scope of this book to explain.

If you are not familiar with front-end technologies like HTML/CSS/Javascript, then I do recommend *Pro HTML5 with CSS, JavaScript, and Multimedia* (Apress, 2017).

For our reader's—or, I should say, learner's—convenience we have created a basic UI required for a chatbot and user conversation. You will find the entire working code on github or Apress's website. I am just going to tell you the configuration needed to make it work for your bot.

Once you download the code for this chapter you will find the folder inside the main folder called `my_own_chatbot`. Go to this folder and go to assets -> js -> `script.js` file.

Change the following line of javascript code to your own endpoint url. If your app name was different then below url will be different in your case. Use your own url with the token in the javascript code as shown in the url below.

```
var baseUrl = "https://horoscopebot1212.herokuapp.com/webhooks/rasa/
webhook?token=YOUR-SECRET";
```

Save the file and open the `index.html` file in the browser and you could easily see a chatbot UI ready. But making API calls from simple HTML being served locally raises a CORS issue. So, to avoid this situation we are going to change our existing `myown_chatbot.py` a bit to serve the HTML from our Heroku app itself.

Change your `myown_chatbot.py` to the below and then we'll discuss the changes made.

```
import os

from rasa_core.channels.rasa_chat import RasaChatInput
from rasa_core.channels.channel import CollectingOutputChannel, UserMessage
from rasa_core.agent import Agent
from rasa_core.interpreter import RasaNLUIInterpreter
from rasa_core.utils import EndpointConfig
from rasa_core import utils
```

```

from flask import render_template, Blueprint, jsonify, request

# load your trained agent
interpreter = RasaNLUInterpreter("models/nlu/default/horoscopebot/")
MODEL_PATH = "models/dialogue"
action_endpoint = EndpointConfig(url="https://horoscopebot1212-actions.
herokuapp.com/webhook")

agent = Agent.load(MODEL_PATH, interpreter=interpreter, action_endpoint=
action_endpoint)

class MyNewInput(RasaChatInput):
    @classmethod
    def name(cls):
        return "rasa"

    def _check_token(self, token):
        if token == 'secret':
            return {'username': 1234}
        else:
            print("Failed to check token: {}".format(token))
            return None

    def blueprint(self, on_new_message):
        templates_folder = os.path.join(os.path.dirname(os.path.abspath(__
file__)), 'myown_chatbot')

        custom_webhook = Blueprint('custom_webhook', __name__, template_
folder = templates_folder)

        @custom_webhook.route("/", methods=['GET'])
        def health():
            return jsonify({"status": "ok"})

        @custom_webhook.route("/chat", methods=['GET'])
        def chat():
            return render_template('index.html')

        @custom_webhook.route("/webhook", methods=['POST'])
        def receive():

```

```

sender_id = self._extract_sender(request)
text = self._extract_message(request)
should_use_stream = utils.bool_arg("stream", default=False)

if should_use_stream:
    return Response(
        self.stream_response(on_new_message, text,
                             sender_id),
        content_type='text/event-stream')
else:
    collector = CollectingOutputChannel()
    on_new_message(UserMessage(text, collector, sender_id))
    return jsonify(collector.messages)

return custom_webhook

input_channel = MyNewInput(url='https://horoscopebot1212.herokuapp.com')
# set serve_forever=False if you want to keep the server running
s = agent.handle_channels([input_channel], int(os.environ.get('PORT',
5004))), serve_forever=True)

```

Here are the changes that we made:

- Override the existing name and blueprint method in our class, which lets us create our own endpoint and also gives us the liberty to define how it should behave.
- We created a new endpoint/`chat` and served the `index.html` file, which is nothing but the UI for chatbot. So, this will be the home link for our chatbot.
- We had to import some necessary classes and methods, like `utils`, `CollectingOutputChannel`, and `UserMessage` as needed to make things work.

Save the file and deploy the changes again to our Heroku app using the following commands:

```

$ git add .
$ git commit -am "deploy my bot"
$ git push heroku master

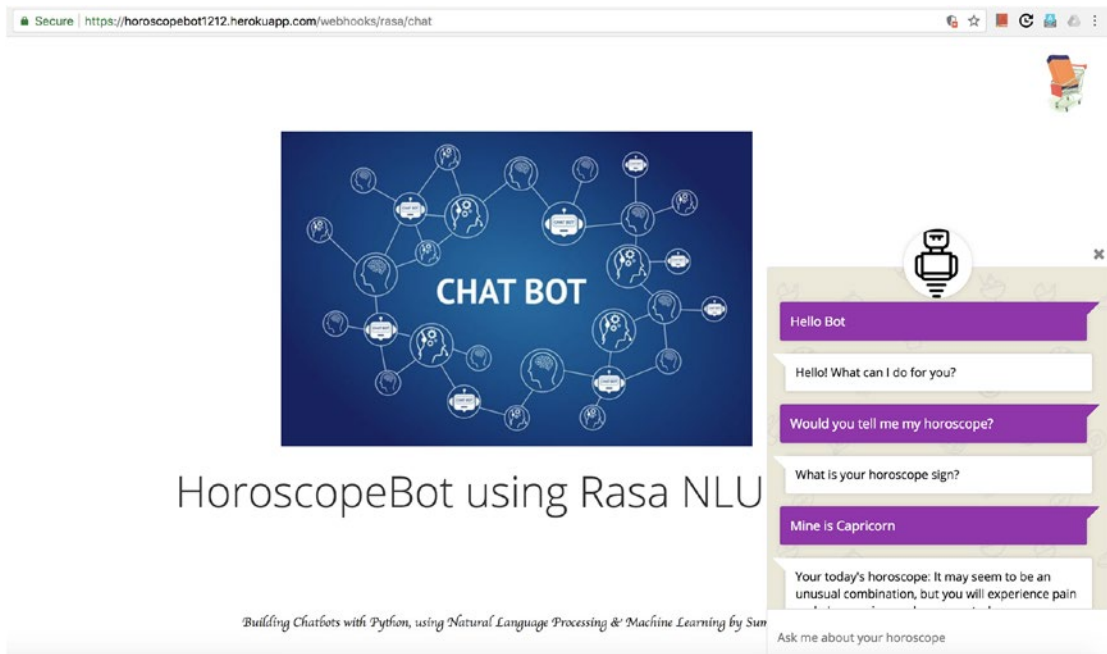
```

Once deployed successfully—Voila! We have our bot ready to be shared with the entire world, which works using two Heroku apps: one for dialog management and one for actions.

Open the following url in the browser where we should see our custom chatbot UI:

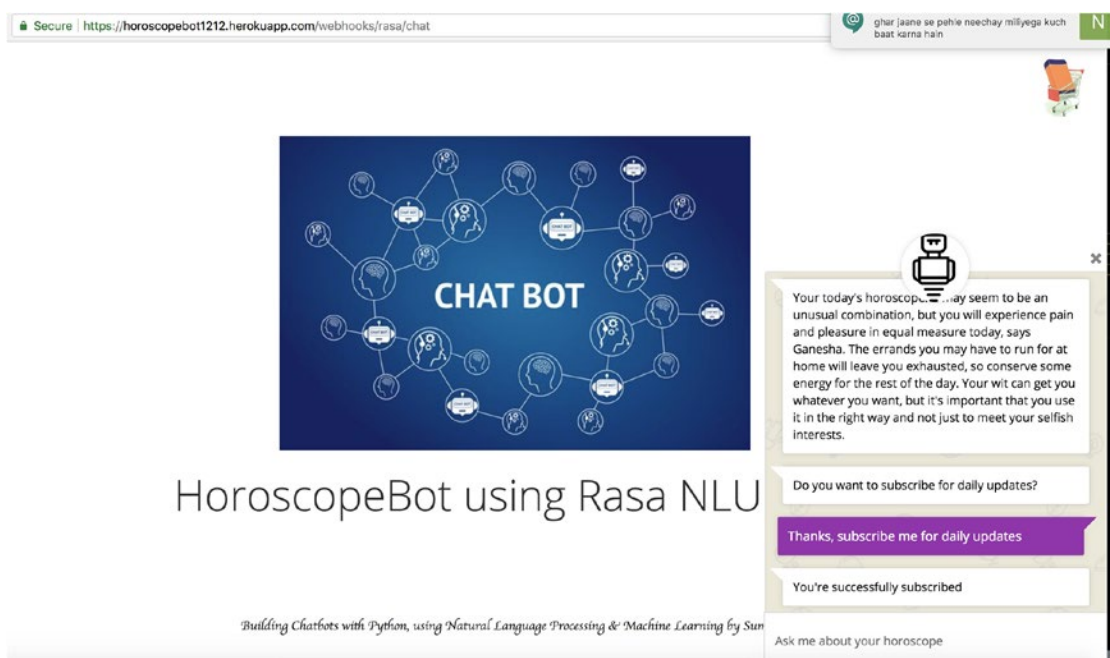
<https://horoscopebot1212.herokuapp.com/webhooks/rasa/chat>

Figures 5-20.1 and 5-20.2 show how my own chatbot looks during the conversation.



**Figure 5-20.1.** Your own custom chatbot on your own website





**Figure 5-20.2.** *Your own custom chatbot on your own website*

Using Heroku’s custom domain name features, one can easily point the same app to their own website’s name, like `www.example.com`. Do that when you feel your chatbot is good enough to be opened to the entire world for profit or non-profit purposes.

So, that’s all folks! That’s how chatbots are built with Python using Natural Language Processing and machine learning. I hope this chapter along with the previous chapters was helpful to you and you could learn the practical approach toward learning and building chatbots.

## Summary

In this chapter, we learned how to deploy our own app to our own servers using Heroku. We learned how to integrate our chatbot with Facebook using Facebook’s developer platform. We also learned to create a custom chatbot for a Slack app and tested it as well. Finally, as promised at the end of chapter-III, removing all dependencies from any social media platform platform, we created our own UI and deployed it on Heroku and tested it. We saw it working like a charm—it worked while training it. As we have a basic model

up and working, now you can handle cases where chatbot doesn't work well. Determine if it's an issue related to data or training, actions server, or custom code handling. Once you find the root case, fix that, deploy again, and check to see if chatbot improves. We build big software by starting small.

I am looking forward to hearing from you and curious to know what chatbot you built after going through this book. I will be happy to help you anytime if you are stuck on any concepts, code execution, or deployment.

Thanks and Cheers.