Chatbots Made Easier With Rasa 2.0

Making chatbots has never been easier

Abhishek Verma 11 hours ago · 6 min read ★

Photo by Quang Anh Ha Nguyen from Pexels

What is Rasa?

> Rasa is an open source machine learning framework for automated text and voice-based conversations. Understand messages, hold conversations, and connect to messaging channels and APIs.

What's new in Rasa 2.0?

> Easy-to-use for beginners and replacement of markdown with YAML.

Installation of Rasa

The basic pre-requisite here is Anaconda. It helps to handle packages and keep different code dependencies from meddling with each other.

Whether you are in a Windows or Linux system, these commands shall work:

```
conda create -n rasa python=3.6
conda activate rasa

# pip install rasa[full] # adds all dependencies of rasa
pip install rasa
```

This is it!

If you use `pip install rasa[full]` it will install all dependencies (spaCy and MITIE) of Rasa for every configuration making your life easier.

Caution:

During installation, there is a conflict in dependencies of Tensorflow and Rasa i.e. NumPy. Tensorflow requires Numpy <0.19, while Rasa requires 0.19.4.

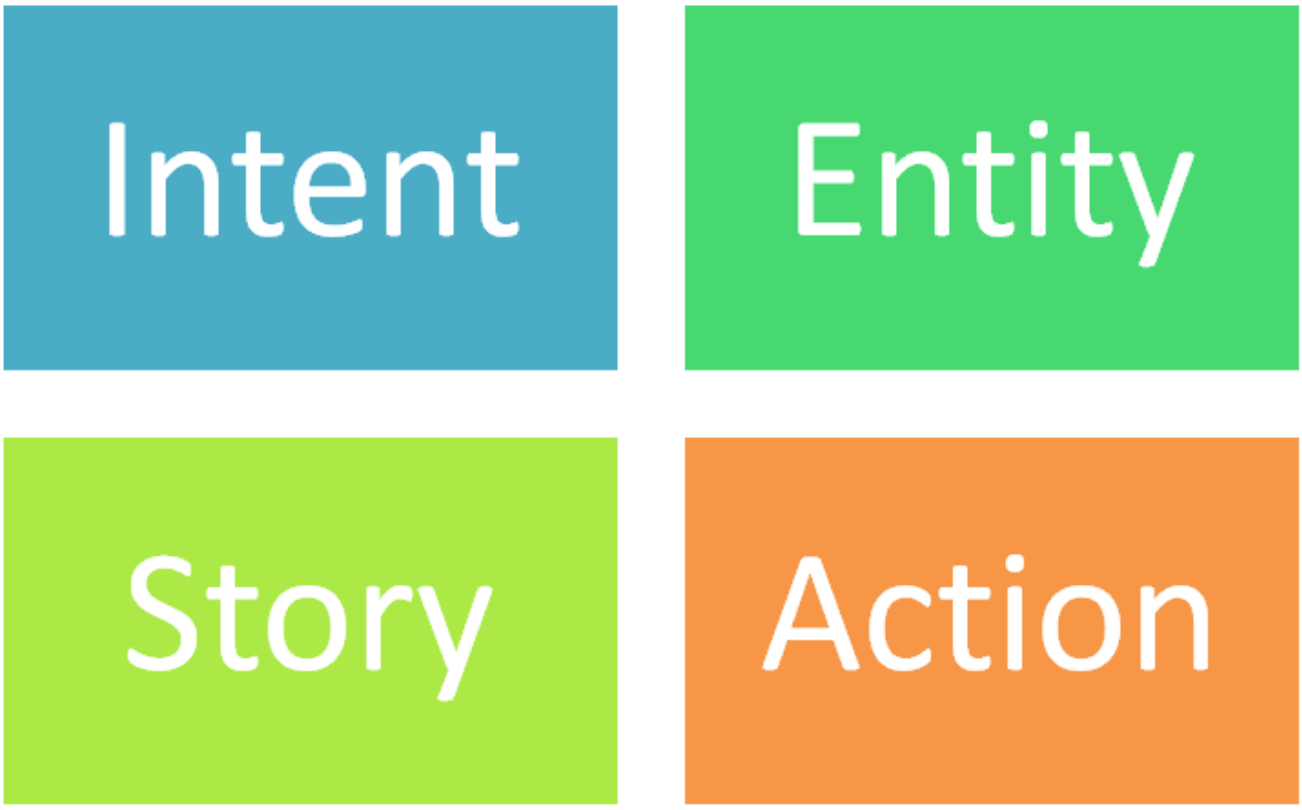
If you try running Rasa, you will run into a runtime error.

To solve that, just downgrade your Numpy:

```
pip install numpy==0.19
```

Before We Start

Let's understand some basic concepts in Rasa.



Source: Author

Intent

What is the user intending to ask about?

Entity

What are the important pieces of information in the user's query?

Story

What is the possible way the conversation can go?

Action

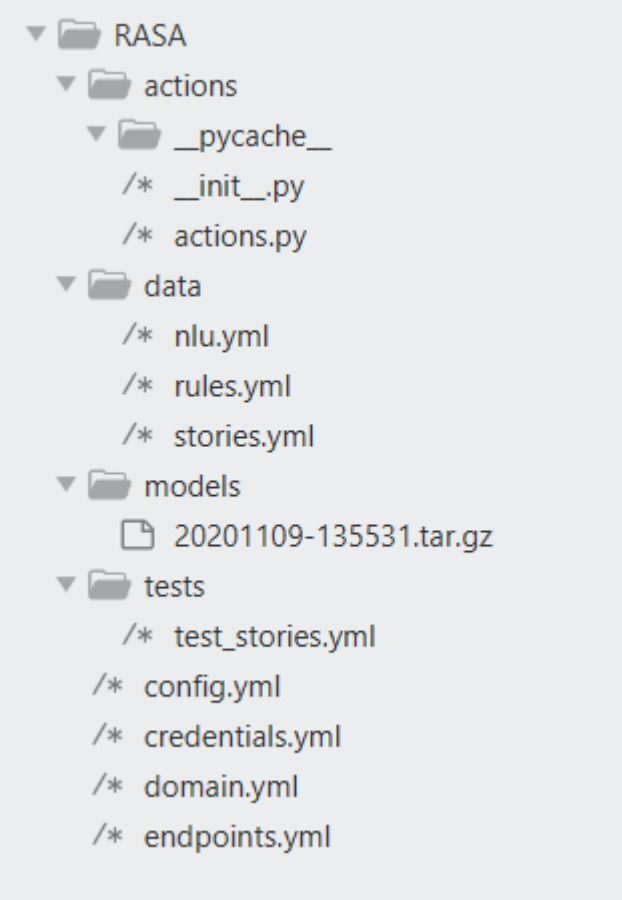
What action should the bot take upon a specific request?

Let's Start

The first thing to do is:

```
rasa init
```

You will then be prompted to create a sample Rasa project in the current folder.



Source: Author

This is what the sample project will look like. **The models file is created once we train the model.**

First Stop: nlu.yml

```
version: "2.0"
```

```
nlu:
- intent: greet
  examples: |
    - hey
    - hello
    - hi
    - hello there
    - good morning
    - good evening
    - moin
    - hey there
    - let's go
    - hey dude
    - goodmorning
    - goodevening
    - good afternoon

- intent: goodbye
  examples: |
    - good afternoon
    - cu
    - good by
    - cee you later
    - good night
    - bye
    - goodbye
    - have a nice day
    - see you around
    - bye bye
    - see you later
```

nlu.yml (Source: Author)

This is what the file looks like. It is quite intuitive to start with.

In order to create your own custom chatbot, you will have to code some intents into it. If you are making a chatbot for a city, intents would be tourist destinations, restaurants,

etc. *Make sure intent names aren't duplicated.*

To put a new intent, we just have to follow suit of any of the intents above. The **intent** tells us the name of the intent. The **examples** tell us what the language regarding this intent be like. Make sure to at least give 2 examples.

Second Stop: actions.py

```python
# This files contains your custom actions which can be used to run
# custom Python code.
#
# See this guide on how to implement these action:
# https://rasa.com/docs/rasa/custom-actions


# This is a simple example for a custom action which utters "Hello World!"

from typing import Any, Text, Dict, List

from rasa_sdk import Action, Tracker
from rasa_sdk.executor import CollectingDispatcher


class ActionHelloWorld(Action):

    def name(self) -> Text:
        return "action_hello_world"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

        dispatcher.utter_message(text="Hello World!")

        return []
```

actions/actions.py (Source: Author)

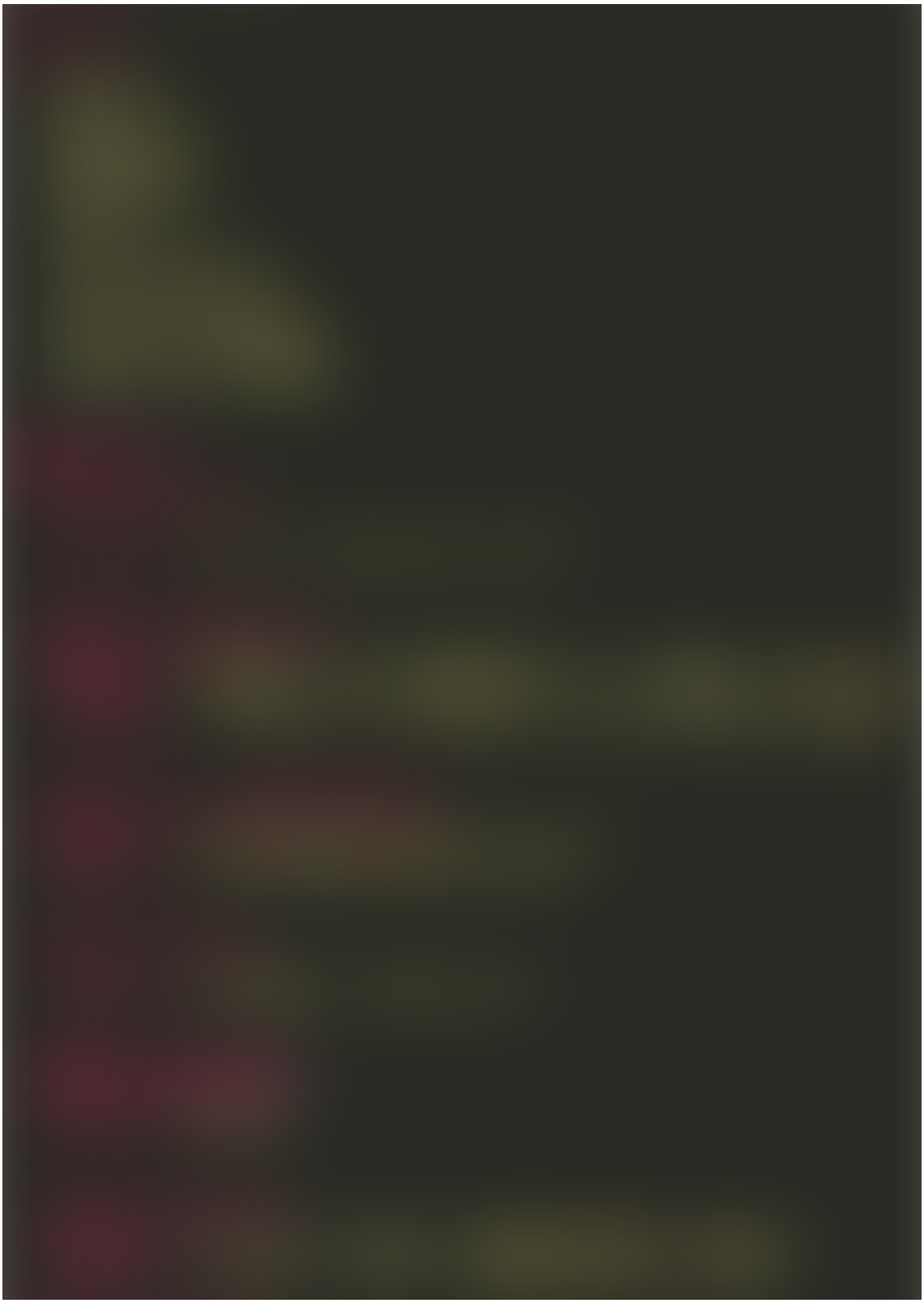
In order to create your own custom actions, this example can be followed.

You will have to create a child class of **Action** and it will have two functions, **name** and **run**. The function **name** will be returning the action name that will be used by the chatbot. The function **run** will hold the functionality performed by your action such as computing some numbers, fetching data from the Internet, etc. Using

dispatcher.utter_message, you can send a reply through the chatbot such as the answer to the computation.

Make sure the name of any two actions don't clash.

Third Stop: domain.yml



domain.yml (Source: Author)

The **domain.yml** contains the domain knowledge of the chatbot i.e. the information it needs to operate (what will people ask and what it has to reply or do?).

In case, you want to use actions in your chatbot, then, you also need to add the **actions** section to this file. If not, skip this part.

```
actions:
  - action_1
  - action_2
```

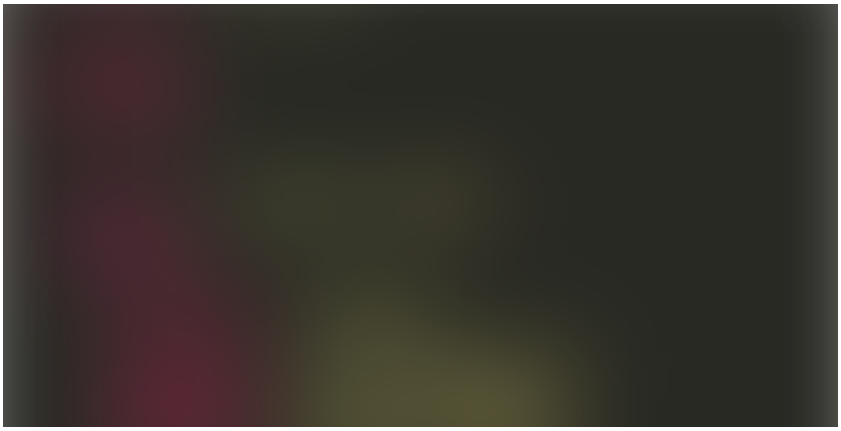
Under the **actions** section, put the name of the actions you have created in the previous step.

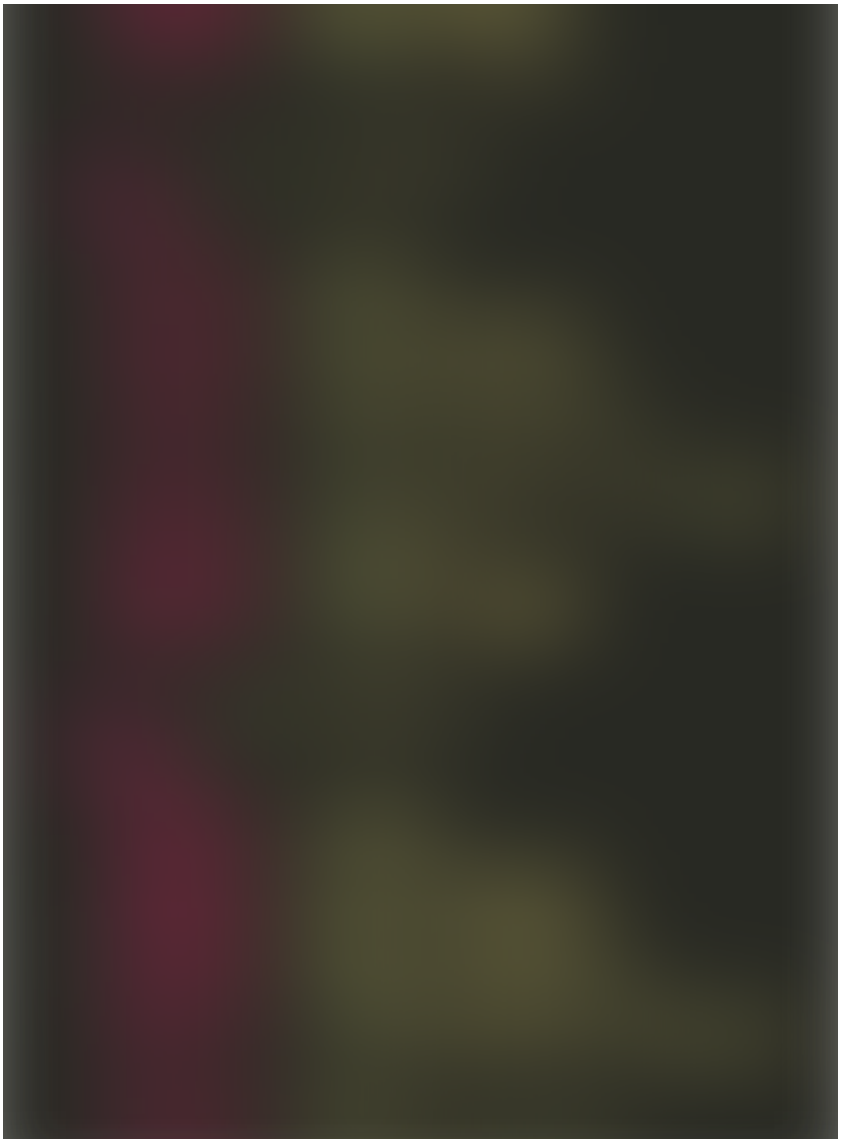
Under the **intents** section, you have to put in your custom intents.

The next step would be to create utterances under the **responses** section i.e. the answers your chatbot will be giving to the user when a question regarding the intent is asked. *Make sure response names are not duplicated.*

Give unique response names and under them, give an answer you would like to give when asked a question about a certain intent.

Fourth Stop: stories.yml





stories.yml (Source: Author)

This is the place where we stitch everything together.

Under the **stories** section are the various ways, a conversation can flow between the user and the chatbot. For your chatbot, you put your stories here.

Following the suit of the above examples, you will first put a story name and the steps in it. You can put in the name of utterances and actions in the *action part of the story*.

So, your intents go into the **intent** section of a story while your actions and utterances go into the **action** part of the story.

Training the Chatbot

Now, you have put all the information about your custom chatbot in the various files. Let's train your chatbot.

```
rasa train
```

Simply, type the above command and let the magic happen!

Running the Chatbot

Now, you can integrate this chatbot into your own website, Facebook Messenger, Slack, Telegram, Twilio, Microsoft Bot Framework, Cisco Webex Teams, RocketChat, Mattermost, and Google Hangouts Chat. You can also connect the chatbot to any other platform but you would have to create your own custom connector.

For now, let's test it on localhost.

Note: If you have used **actions** in your chatbot, then, you need to uncomment the **action_endpoint** section in **endpoints.yml**. Then, open a new terminal making sure you are in the Rasa folder. Then, you have to run `rasa run actions`

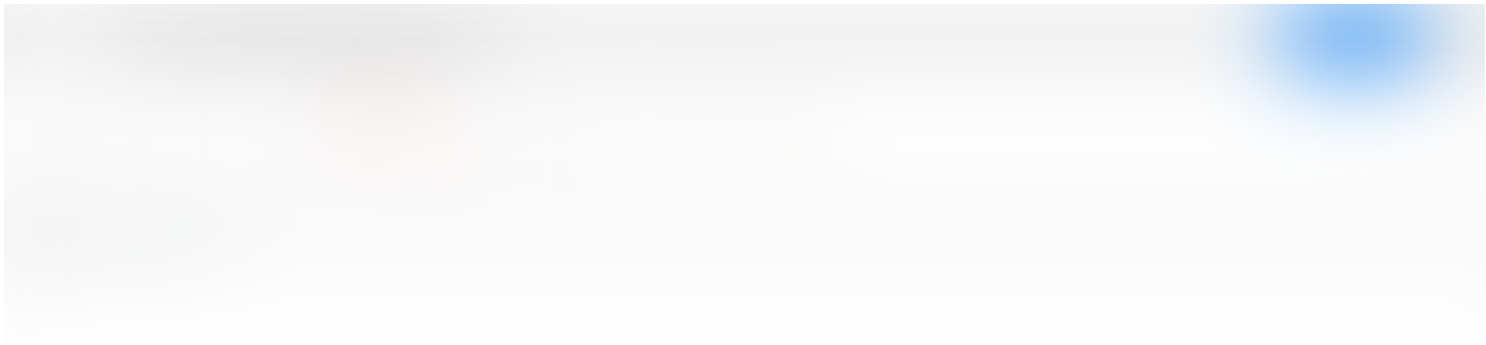
In order to run Rasa locally, just run the following command:

```
rasa run
```

This would expose a REST endpoint at 5000 port in the localhost.

Talking with the Chatbot

In order to talk with the chatbot, you can use cURL commands or Postman. I prefer Postman because of its ease of use.



Source: Author

You have to make a **POST** request to the URL:

http://localhost:5005/webhooks/rest/webhook

In the **Body** of the request, select the **raw** option in Postman.

In JSON format, you have to specify two keys: **sender** and **message**.

Put a dummy value in the sender and the message will be your query to the chatbot.

You will be returned a JSON with keys: **recipient_id** and **text**. **text** will have the reply of the chatbot to your query and **recipient_id** will be the same as the **sender** key you sent in the request.

Summary

We understood how using creating a chatbot is easy using Rasa. We understood the basic concepts in Rasa framework i.e. intent, entity, action, and story. We understood what are the steps to create a custom chatbot suited to our own needs. We also understood the various pitfalls that can happen while installing and running Rasa.