Botshot Documentation

David Prihoda, Matus Zilinec, Jakub Drdak

May 26, 2019

Contents:

CHAPTER 1

Quick Start

Botshot is a framework for building chatbots in Python. It enables developers to build reliable and smart bots. Its main advantages are that it defines a concrete structure of the conversation and keeps track of the conversation history.

1.1 Dependencies

Botshot runs on top of the Django web framework. It is built with scalability in mind, and Celery is used for message processing. For persistence of conversation context, we use Redis. You don't need to know any of these tools, but it's a plus.

See also:

If you have never worked in Python, don't worry. It is a very intuitive and easy to learn language. Check out the official beginners guide.

CHAPTER 2

Installation

The easiest way to get started is using our `bots` command line tool.

Note: We strongly recommend using a virtual environment to avoid package conflicts. `virtualenv -p python3 bot_env/ && source bot_env/bin/activate`

First, install Botshot and dependencies from PyPI.
`pip3 install botshot`

Now go ahead and run:
`bots init my_bot`
This will bootstrap a chatbot with default configuration under `my_bot/`.

2.1 Running the chatbot

First, cd to the chatbot directory you just created.
`cd my_bot`
You can now run the chatbot with
`bots start my_bot`.

The web chat interface should now be running at http://localhost:8000/ .
Go ahead and say something to your chatbot!

2.1.1 Running from IDE (optional)

In case you're using an IDE such as PyCharm for development, it might be more convenient to run the chatbot from within it, to make use of the debugger.

You will need to manually start these three components:

- **Django server** Under **Edit configurations > Add configuration > Django server**
- **Celery tasks**

  Under **Edit configurations > Add configuration > Python**
  Script path: path to celery - bot_env/bin/celery
  Working directory: chatbot root

- **Redis database**

  You can either run `redis-server` from the terminal or with an init script, as you would any other database.
  If you're on Ubuntu and have installed redis with `apt install redis`, it should already be running.
  You can connect to your database using `redis-cli`.

CHAPTER 3

Configuration

If you used the `bots` init tool, you don't need to configure anything at the moment.

TODO The dialogue manager is quite well thought out and has a powerful set of rules that you can utilize to handle any conversational situation.

3.1 Project structure

As Botshot runs on top of Django, its projects use the familiar Django structure.

It might look similar to this:

```
my_bot/  # the root directory
    my_bot/  # your chatbot (django) app
        chatbots/            # actual chatbot logic (read on)
        botshot_settings.py  # chatbot-specific settings
        settings.py          # web server settings
    manage.py  # script that starts web server
```

3.2 Settings

In `botshot_settings.py`, you will find a python dict `BOT_CONFIG`.
Most configuration values, such as for NLU and messaging endpoints, belong here.

`settings.py` contains configuration of the web server.
Notably, you can:

- link SQL databases under `DATABASES`,
- permit listening on URLs under `ALLOWED_URLS`.

For more information, see django settings.

3.3 Environment variables

If you plan to develop a larger chatbot, you shouldn't hardcode configuration values. You can provide them as environment variables instead. This can be done with a so-called env file.

You can create a file, for example `.env`, and put your config values in it like in a shell script:

```
DEBUG=false
FB_PAGE_TOKEN=foo
NLU_TOKEN=bar
ROOT_PASSWORD=12345
```

Then, to make it load automatically, you can for example:

- (virtualenv) add this line to *env/bin/activate*: `export $(grep -v '^#' .env | xargs)`
- (PyCharm) use the EnvFile plugin, add the file in run configuration

3.3.1 BOT_CONFIG reference

- WEBCHAT_WELCOME_MESSAGE

CHAPTER 4

Parts of a chatbot

So anyway, what does a good chatbot consist of?

Well, multiple things. For one, it needs to be able to **send and receive messages**, or it wouldn't be a chatbot, right?
It also needs to **understand messages** and know their meaning (partially at least).
It should also **keep track of conversation** and remember the **conversation history**. In other words, it shouldn't lose the thread in the middle of a conversation.
Finally, it should be able to **generate messages** that answer the users' input.

Let's go over these things, really quickly.

4.1 Messaging endpoints

Botshot supports popular messaging platforms such as *Facebook messenger* and *Telegram*. For each of these platforms, there is an endpoint that converts the messages to a universal format, so you don't need to worry about compatibility.

To enable support for messaging platforms, just add the associated API token to `BOT_CONFIG`. Read more at *Messaging endpoints*.

Note: Botshot is fully extensible. If you need to support your private messaging API, just implement the endpoint and it will work.

4.2 Natural language understanding (NLU)

Natural language understanding is the process of analyzing text and extracting machine-interpretable information.
NLU is actually a topic in AI, but you don't have to be an expert to make a chatbot. Today, there are many available services that do the job for you.

Botshot provides easy integration with the most popular tools, such as Facebook's Wit.ai, Microsoft's Luis.AI, or the offline Rasa NLU.
We also have our own NLU module, designed specifically for use with Botshot. Contact us if you're interested.
Although you don't have to, you should really use NLU. The above tools are very easy to use.

NLU tools are useful for these purposes:

1. **Intent detection aka "What does the user want?"**

   Classify the message into a *category*.
   Input: `{"text": "Hi there!"}`
   Output: `{"intent": "greeting", "confidence": 0.9854, "source": "botshot_nlu"}`

2. **Entity extraction aka "How does the user want it?"**

   Extract *entities* such as *dates*, *places* and *names* from text.
   Input: `{"text": "Are there any interesting events today?"}`
   Output: `{"query": "events", "date": "2018-01-01"}`

To enable support for NLU platforms, just add the associated API token to `BOT_CONFIG`. Read more at **'Natural language understanding'_**.

Note: You can use your own machine learning models for NLU if you wish. See **'Entity extractors'_** for more details.

4.3 Dialogue Management & Conversation Context

If you're building a chatbot that does more than say "hello", you will need a way to keep track of the conversation

and remember what the user has said before. | Botshot has a Dialogue Manager that does exactly this. You can picture the conversation as a state machine with states like *greeting* and transitions that fire when a message is received. | There is also a Context Manager that you can query about past conversations and NLU entities.

Note: We would be really happy if we could just train a neural network instead. However, there are still many problems that need to be solved before production use.

Alright, enough chit chat. Let's get coding!

CHAPTER 5

Conversation

Let's build a chatbot!

The central part of your chatbot is the conversation (meaning, what and when should the bot say).
In most chatbots, you will represent the conversation by a graph.

Each node in this graph represents a specific state of the conversation, such as "greeting" or "asking for directions".
The chatbot can then move between states, for example, when the user asks something.

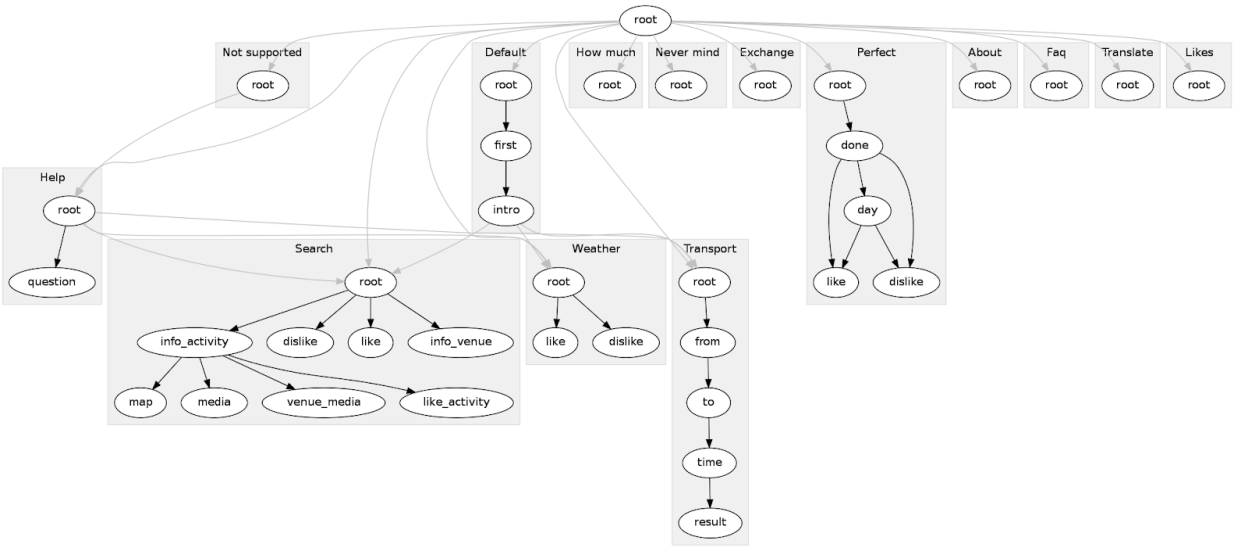

Fig. 1: Graph of conversational states in a chatbot.

5.1 Flows and states

Botshot provides a **dialog manager** that makes it easy to define and move between these **states**.
We further group states into so-called **flows**, conversations about a particular topic. These are similar to packages or modules in a programming language.

Usually, you will want to define the set of possible states, the **transitions** between these states and the **actions** that the bot should do in these states.
You can define the conversation in YAML, in JSON[1], or directly in code.
Actions can either be written in Python, or they can be hardcoded in the conversation.

5.1.1 Defining the conversation in YAML

We prefer to use YAML over definitions in code, as it is cleaner and allows to separate structure from the content.
If you used the `bots` script, there is already a default flow at `my_bot/chatbots/default/flow.yml`.
Each `flow.yml` has to be enabled in `bot_settings.py`:

```
BOT_CONFIG = {
    "FLOWS": {  # we recommend creating a separate directory and file for each flow
        "chatbot/bots/default/flow.yml"
    ...
```

The structure of `flow.yml` is as follows.

```
greeting:             # flow "greeting"
  states:
    - name: root          # state "root" of flow "greeting"
      action:             # action run when state is trigerred
        text: "Hello there!"      # send a message with text "Hello there!"
        next: "greeting.joke:"    # move to state "joke" and execute it (:)
    - name: joke          # state "joke" of flow "greeting"
      action: actions.show_a_joke  # actions can be either hardcoded messages or␣
↪functions

city_info:    # a flow - dialogue that shows facts about a city
  states:
    - name: root
      action: actions.show_city_info
  accepts:    # entities that trigger this flow
  - city_name
```

You can now skip to the next page see how the actions are implemented in Python, or you can continue reading about how the dialog manager works.

[1] YAML (stands for YAML Ain't Markup Language) is a superset of JSON.

State transitions

Each conversation starts in the default.root state.
The system of transitions between states is quite well thought out.
This is important, get ready.

1. **Intent transition** First, the dialogue manager checks whether an **intent** was received from the NLU. If that's true, it looks for a flow with the same name as the intent. So for example, when user's message was "Hello there!", the recognized intent is *greeting* and the chatbot tries to move to the *greeting* flow. If such a state exists, the bot executes its *root* state's action (which in this case says "Hello there!"). You can override this with your own regex:

```
greeting:
  intent: "(greeting|goodbye)"
  states:
  ..
```

2. **Entity transition** If no intent was detected, the DM tries to move by NLU **entities**. For example, if the message was "New York", we can hardly know what intent the user had, but we might have extracted the entity *city_name* using our NLU tool. Therefore, the bot moves to the *city_info* flow, as it **accepts** this entity. Accepted entities are specified like this:

```
greeting:
  accepts:
  - username
...
```

3. **Manual transitions** You can also move between states manually using the `next` attribute, or from code. Remember that `next: "default.root"` just moves to the state, but `"default.root:"` also runs its action. You can use relative names as well. `next: "root"`

Supported entities

If neither **intent** nor **entity transition** was triggered, the bot checks if the current state is able to handle the received message.
It does this by checking the current state's **supported entities** against the message's entities.
These can be specified using the `supports:` attribute below.
This way, you can **prevent a transition** from happening, if the message is supported.

If there is at least one supported entity in the message, Botshot finally executes the current state's action.
The action can either be a harcoded message or a python function with custom logic that generates a response. You can read about actions in the next page.

Otherwise, Botshot first tries to execute the **unsupported** action of the current state, which would usually say something like "Sorry, I don't get it". If no such action exists, it moves to state `default.root`.

If the user sends a supported message *after* the bot didn't understand, the conversation is reverted to the original state, as if nothing had happened.

```
free_text:
- name: prompt
  action:
    text: "Are you satisfied with our services?"
    next: "input"  # move without executing the state
- name: input    # wait for input
  supports:      # entities the state can handle
  - yesno
  unsupported:  # what to say otherwise
    text: "Sorry, I don't get it."
    replies: ["Yes", "No", "Nevermind"]
```

You might only want to support a specific set of entity-values.

```
...
supports:
- intent: greeting  # this intent won't trigger an intent transition
- place:            # list of supported entity-values
  - Prague
  - New York
```

Note: An accepted entity is implicitly supported.

As they say, a picture is worth a thousand words: (TODO picture)

In the next page, we shall discuss sending messages to the user.

Requirements (optional)

A common pattern in chatbots is to ask for additional information before answering a query. Consider this conversation:

- **USER** Hey bot, book me a hotel in Prague.
- **BOT** Sure thing, when would you like to check in?
- **USER** Tomorrow
- **BOT** And how many nights are you staying for?
- **USER** For a week I suppose.
- **BOT** Cool! These are the best hotels that I know: ...

This sort of repetitive asking could get quite complicated and tedious. Fortunately, you can leave the logic to Botshot. Each state can have a list of requirements along with prompts to get them. Example:

```
greeting:
    states:
    - name: root
      action: actions.hotel_search
      require:

      - entity: "datetime"  # check if entity is present
```

(continues on next page)

(continued from previous page)

```
      action:
        text: "When would you like to check in?"
        replies: ["Today", "Tomorrow", "Next friday"]

    - condition: actions.my_condition  # a custom function returning boolean
      action: actions.my_prompt        # an action, see the next page
```

CHAPTER 6

Actions

If you read the previous page, you should already suspect what actions are.
To recap, each state has an action that triggers under some conditions. This action can be a hardcoded message or a python function.

6.1 Hardcoded actions

You can simply define a message to be sent directly from the YAML flow.
You can try making a flow just with these and test it in the web chat interface before moving on to Python code.

```
action:
  text: "How are you?"          # sends a text message
  replies:                      # sends quick replies below the message
  - "I'm fine!"
  - "Don't even ask"
  next: "next.root:"            # moves to a different state
```

6.2 Coding actions in Python

You can call a custom function anywhere where you would use a hardcoded action. The function can also be imported relatively to path of the YAML file.

```
# absolute import
action: chatbot.bots.default.conditions.bar
action: actions.foo  # relative import
```

The function takes one parameter - a `Dialog` object that you can use to send messages and access other APIs. It can also return the name of the next state (equivalent to `next` in YAML). The function should look at the conversation context (NLU entities), fetch any data it needs from external APIs and send messages to the user.

```
import random
from custom.logic import get_all_jokes

from botshot.core.dialog import Dialog
from botshot.core.responses import *

def show_joke(dialog: Dialog):
    jokes = get_all_jokes()
    joke = random.choice(jokes)
    dialog.send(joke)  # sends a string message
    return "next.root:"
```

Note: While waiting for the function to return, a typing indicator is displayed. (three dots in messenger)

6.2.1 Message templates

You can send messages by calling `dialog.send()`. This method requires one argument - the message!

The message should either be a string, a template from `botshot.core.responses`, or a list of these.
Here is a list of all the templates you can use:

Text message

A basic message with text. Can optionally have buttons or quick replies.

The officially supported button types are:

- `LinkButton(title, url)` Redirects to a webpage upon being clicked.
- `PayloadButton(title, payload)` Sends a special **'postback message'_** on click.
- `PhoneButton(title, phone_number)` Facebook only. Calls a number when clicked.
- `ShareButton()` Facebook only. Opens the "Share" window.

The supported quick reply types are:

- `QuickReply(title)` Used to suggest what the user can say. Sends "title" as a message.
- `LocationQuickReply()` Facebook only. Opens the "Send location" window.

```
msg = "Botshot is the best!"
dialog.send(msg)

msg = TextMessage("I'm a message with buttons!")
msg.add_button(LinkButton("I'm a button", "https://example.com"))
msg.add_button(PayloadButton("Next page", payload={"_state": "next.root:"}))
msg.add_button(ShareButton())
```

(continues on next page)

(continued from previous page)

```
# or ...
msg.with_buttons(button_list)
dialog.send(msg)

msg = TextMessage("I'm a message with quick replies!")
# or ...
msg.add_reply(LocationQuickReply())
msg.add_reply(QuickReply("Lorem ipsum ..."))
msg.add_reply("dolor sit amet ...")
# or ...
msg.with_replies(reply_list)
```

TODO picture, result on more platforms?

Note: Different platforms have different message limitations. For example, quick replies in Facebook Messenger can have a maximum of 20 characters.

Image message

TODO

Audio message

TODO

Video message

TODO

Card template

```
msg = CardTemplate(
    title="A card",
    subtitle="Hello world!",
    image_url="http://placehold.it/300x300",
    item_url="http://example.com"
)
msg.add_button(button)
```

Carousel template

```
msg = CarouselTemplate()
msg.add_element(
    CardTemplate(
        title="Card 1",
        subtitle="Hello world!",
        image_url="http://placehold.it/300x300"
    )
)
```

List template

```
msg = ListTemplate()
msg.add_element(
    CardTemplate(
        title="Card 1",
        subtitle="Hello world!",
        image_url="http://placehold.it/300x300"
    )
)
```

Sending more messages at once

```
messages = []
for i in range(3):
    messages.append("Message #{}".format(i))
dialog.send(messages)
```

TODO picture

> **Warning:** Avoid calling dialog.send() in a for loop. In bad network conditions, the messages might be sent in wrong order.

6.2.2 Scheduling messages

You can schedule a message to be sent in the future. You can optionally send it only if the user doesn't say anything first.

```
payload = {"_state": "default.schedule", "schedule_id": "123"}

# Regular scheduled message - use a datetime or number of seconds
dialog.schedule(payload, at=None, seconds=None)

# Runs only if the user remains inactive
dialog.inactive(payload, seconds=None)
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search