Botshot Documentation

Matus Zilinec, David Prihoda, Jakub Drdak

Dec 17, 2019

Contents:

CHAPTER 1

Quick Start

Botshot is a framework for building chatbots in Python. It enables developers to build reliable and smart bots. Botshot 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 understand the internal workings of these components.

See also:

Python is a very intuitive programming language. If you have never worked in Python, check out the official beginners guide.

CHAPTER 2

Installation

The easiest way to get started is using the `bots` script.

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`

You will also need to install and start the Redis database. On Ubuntu, run:
`sudo apt install redis-server`

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`.

The web chat interface should now be running at http://127.0.0.1:8000/ .
Go ahead and trying sending a message to your chatbot!

2.1.1 Windows support

Install Botshot on Windows Subsystem for Linux (WSL). This was tested on Windows 10 1903 build. Follow the [instalation guide](https://docs.microsoft.com/cs-cz/windows/wsl/install-win10) if you haven't installed WSL yet.

Running Botshot on Windows natively is not supported yet.

2.1.2 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, for example to make use of the debugger.

You will need to manually start the required 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.

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
    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

Anyway, what does a good chatbot consist of?

Well, for one, it needs to be able to **send and receive messages**, or it wouldn't be a chatbot, right?
Then it needs to **understand messages** and 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 now go over the main components that Botshot provides.

4.1 Messaging endpoints

Botshot supports popular messaging platforms such as *Facebook*, *Telegram* or *Alexa*. The messages are converted to a universal format, so you don't need to worry about compatibility.

For each of these platforms, there is a corresponding chat interface. To enable support for a messaging platform, just enable the associated interface in the config. Read more at *Messaging endpoints*.

Note: Botshot is fully extensible. If you need to support another messaging API, just implement your own chat interface.

4.2 Natural language understanding (NLU)

Natural language understanding is the process of analyzing text and extracting machine-interpretable information.
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.
Or you can use our own Botshot NLU module.

When a message is received, the chatbot should:

1. **Analyze what the user wants, aka Intent detection**

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

2. **Find out details about the query, aka Entity extraction**

   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"}`

You won't need to use a NLU service to get started, but you should really use one to keep your future users happy.
Read more at **'Natural language understanding'_**.

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

4.3 Dialog Management & Context

The Dialog manager is a system responsible for tracking the user's progress in conversation.
You can picture the conversation as a state machine, with each state representing a specific point in the conversation (like *greeting*).
The user can move between these states by sending messages, tapping buttons and so on.
Dialog manager also stores conversation context. That is, if you're building a chatbot that does more than say "hello", you will probably also want to remember what the user has said before.

4.4 Actions

Each state of the conversation has an attached action that returns a response. The most common way of generating responses is using Python code. You can define a function that Botshot will call when a message is received. In this function, you can call your business logic, call any required APIs and generate a response.

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

CHAPTER 5

Conversation

Let's finally build a chatbot!

5.1 Flows and states

Botshot provides a **dialog manager** that makes it easy to define different **states** of the conversation and move between them.
We group states into smaller independent modules, so-called **flows**. A **flow** usually defines a conversation about a specific topic.

You can define the conversation in YAML, in JSON[1], or directly in code.

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 definition from the implementation.
If you used the `bots` script, there is already a default flow in `bot/bots/default/flow.yml`.
You can enable or disable each `flow.yml` by adding or removing it under `FLOWS` in `settings.py`:

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

You can see an example `flow.yml` below.

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

```
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
```

The file contains a flow named "greeting", that has two states, "root" and "joke".
The "root" state is always required - it is where the chatbot will first move when the user sends a "greeting" message.
The state's action can be hardcoded directly in the definition, like in the "root" state, or it can reference a Python function, like in the "joke" state.

All the ways in which you can move between states are described in the optional section below.
You can now skip directly to **'Actions'_** or continue reading.

State transitions

Each conversation starts in the "root" state of the "default" flow.
The user can then system of transitions between states.
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
        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 returning some text.

6.1 Hardcoded actions

You can simply define a message to be sent directly from the YAML flow.
In the example below, each time the user visits the "root" state, the bot sends the message "How are you?" and moves to the state "root" of the flow "next".

```
greeting:
    states:
    - name: "root"
      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 also define the action as a regular Python function. In this function, you can implement business logic and send messages from the bot.

```
greeting:
    states:
```

(continues on next page)

(continued from previous page)

```
    - name: "root"
    # relative import ...
      action: actions.foo
    # ... or absolute import
      action: chatbot.bots.default.conditions.bar
```

The function should take one parameter - `dialog`. This object can be used to send messages and query conversation context and user profile.
To move to another state, the function can optionally return the new state's name (equivalent to `next` in YAML).

```
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:"
```

You can send messages by calling `dialog.send()`. This method takes one argument - the message!
The argument can be a string, a standard template from `botshot.core.responses`, or a list of messages.
See *Message Templates*.

The context of the conversation is stored in `dialog.context`. See **'Context'_** for more information.

Note: The function will be run asynchronously. While waiting for the function to return, a typing indicator will be displayed. (three dots in messenger)

6.2.1 Sending more messages at once

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

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

6.2.2 Sending delayed messages

You can schedule messages to be sent at a time in the future, or when the user is inactive for a period of time. Read more in **'Scheduling messages'_**.

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

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

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

Amazing, you're now ready to implement your first action. But wait, the chatbot does not yet understand anything the user said, what about that? Continue reading and you'll find out in *Natural language processing*.

6.2.3 Message templates

This section contains a list of all message templates that can be sent using `dialog.send()`. The templates are universal, but they will render slightly different on each messaging service.

Text message

A basic message with text. Can contain buttons or quick replies.

`TextMessage(text, buttons=[], replies=[])`

Buttons are shown below the message. They can be used to provide additional related actions.

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

Quick replies are used to suggest a user's response. They contain text that is sent back to the chatbot when clicked. They can also contain payload.

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

(continues on next page)

(continued from previous page)

```
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)
```

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

Media message

MediaMessage can be used to send an image with optional buttons. The image should be located on a publicly available URL.

Example:

```
from botshot.core.responses import MediaMessage

message = MediaMessage(url="http://placehold.it/300x300", buttons=[LinkButton("Open",
↪"http://example.com"))
```

renders as:

Card template

The card template displays a card with a title, subtitle, an image and buttons. It can be used to present structured information about an item or product.

```
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

The carousel template is a horizontal list of cards.

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

List template

The list template is a vertical list of cards.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search