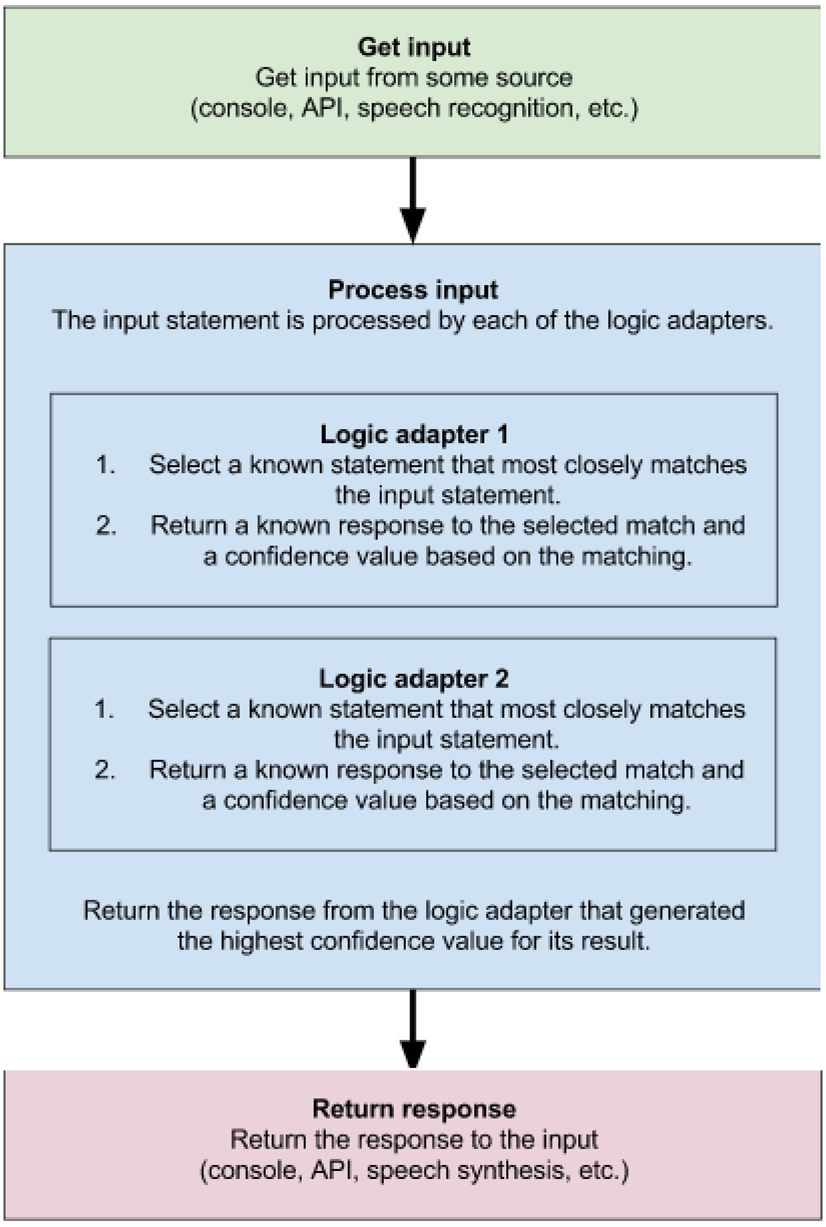
1.ChatterBot https://chatterbot.readthedocs.io/en/stable/

An untrained instance of ChatterBot starts off with no knowledge of how to communicate. Each time a user enters a statement, the library saves the text that they entered and the text that the statement was in response to and program selects the closest matching response by searching for the closest matching known statement that matches the input.

Flow chart


Get input
Get input from some source
(console, API, speech recognition, etc.)
Process input
The input statement is processed by each of the logic adapters.
Logic adapter 1
1. Select a known statement that most closely matches the input statement.
2. Return a known response to the selected match and a confidence value based on the matching.
Logic adapter 2
1. Select a known statement that most closely matches the input statement.
2. Return a known response to the selected match and a confidence value based on the matching.
Return the response from the logic adapter that generated the highest confidence value for its result.
Return response
Return the response to the input
(console, API, speech synthesis, etc.)

Setting the storage adapter

ChatterBot comes with built in adapter classes that allow it to connect to different types of databases.

SQLStorageAdapter which allows the chat bot to connect to SQL databases. By default, this adapter will create a SQLite database.

The database parameter is used to specify the path to the database that the chat bot will use. For this example we will call the database *sqlite:///database.sqlite3*. this file will be created automatically if it doesn't already exist.

```
bot = ChatBot(

    'Norman',

    storage_adapter='chatterbot.storage.SQLStorageAdapter',

    database_uri='sqlite:///database.sqlite3'

)
```

Specifying logic adapters

logic adapter is a class that takes an input statement and returns a response to that statement.

The TimeLogicAdapter returns the current time when the input statement asks for it. The MathematicalEvaluation adapter solves math problems that use basic operations.

```
bot = ChatBot(

  'Norman',

  storage_adapter='chatterbot.storage.SQLStorageAdapter',

  logic_adapters=[

    'chatterbot.logic.MathematicalEvaluation',

    'chatterbot.logic.TimeLogicAdapter'

  ],

  database_uri='sqlite:///database.sqlite3'

)
```

Getting a response from your chat bot

The TimeLogicAdapter returns the current time when the input statement asks for it. The MathematicalEvaluation adapter solves math problems that use basic operations.

```
bot = ChatBot(
    'Norman',
    storage_adapter='chatterbot.storage.SQLStorageAdapter',
    logic_adapters=[
        'chatterbot.logic.MathematicalEvaluation',
            'chatterbot.logic.TimeLogicAdapter'
        ],
        database_uri='sqlite:///database.sqlite3'
)
```

Getting a response from your chat bot

```
while True:
    try:
        bot_input = bot.get_response(input())
        print(bot_input)

    except(KeyboardInterrupt, EOFError, SystemExit):
        break
```

Training your chat bot

```
from chatterbot.trainers import ListTrainer

trainer = ListTrainer(bot)

trainer.train([
    'How are you?',
    'I am good.',
    'That is good to hear.',
    'Thank you',
    'You are welcome.',
])
```

How logic adapters select a response

A typical logic adapter designed to return a response to an input statement will use two main steps to do this. The first step involves searching the database for a known statement that matches or closely matches the input statement. Once a match is selected, the second step involves selecting a known response to the selected match. Frequently, there will be a number of existing statements that are responses to the known match.

Response selection methods determines which response should be used in the event that multiple responses are generated within a logic adapter.

`chatterbot.response_selection.get_first_response`*(input_statement, response_list, storage=None)*[source]

Parameters:

- **input_statement** (*Statement*) – A statement, that closely matches an input to the chat bot.
- **response_list** (*list*) – A list of statement options to choose a response from.
- **storage** (*StorageAdapter*) – An instance of a storage adapter to allow the response selection method to access other statements if needed.

Returns: Return the first statement in the response list.

Return type: Statement

`chatterbot.response_selection.get_most_frequent_response`*(input_statement, response_list, storage=None)*[source]

Parameters:

- **input_statement** (*Statement*) – A statement, that closely matches an input to the chat bot.
- **response_list** (*list*) – A list of statement options to choose a response from.
- **storage** (*StorageAdapter*) – An instance of a storage adapter to allow the response selection method to access other statements if needed.

Returns: The response statement with the greatest number of occurrences.

Return type: Statement

`chatterbot.response_selection.get_random_response`*(input_statement, response_list, storage=None)*[source]

Parameters:

- **input_statement** (*Statement*) – A statement, that closely matches an input to the chat bot.
- **response_list** (*list*) – A list of statement options to choose a response from.
- **storage** (*StorageAdapter*) – An instance of a storage adapter to allow the response selection method to access other statements if needed.

Returns: Choose a random response from the selection.

Return type: Statement

What kinds of machine learning does ChatterBot use?

1. Search algorithms

Search is a crucial part of how a chat bot quickly and efficiently retrieves the possible candidate statements that it can respond with.

Some examples of attributes that help the chat bot select a response include

- the similarity of an input statement to known statements
- the frequency in which similar known responses occur
- the likeliness of an input statement to fit into a category that known statements are a part of

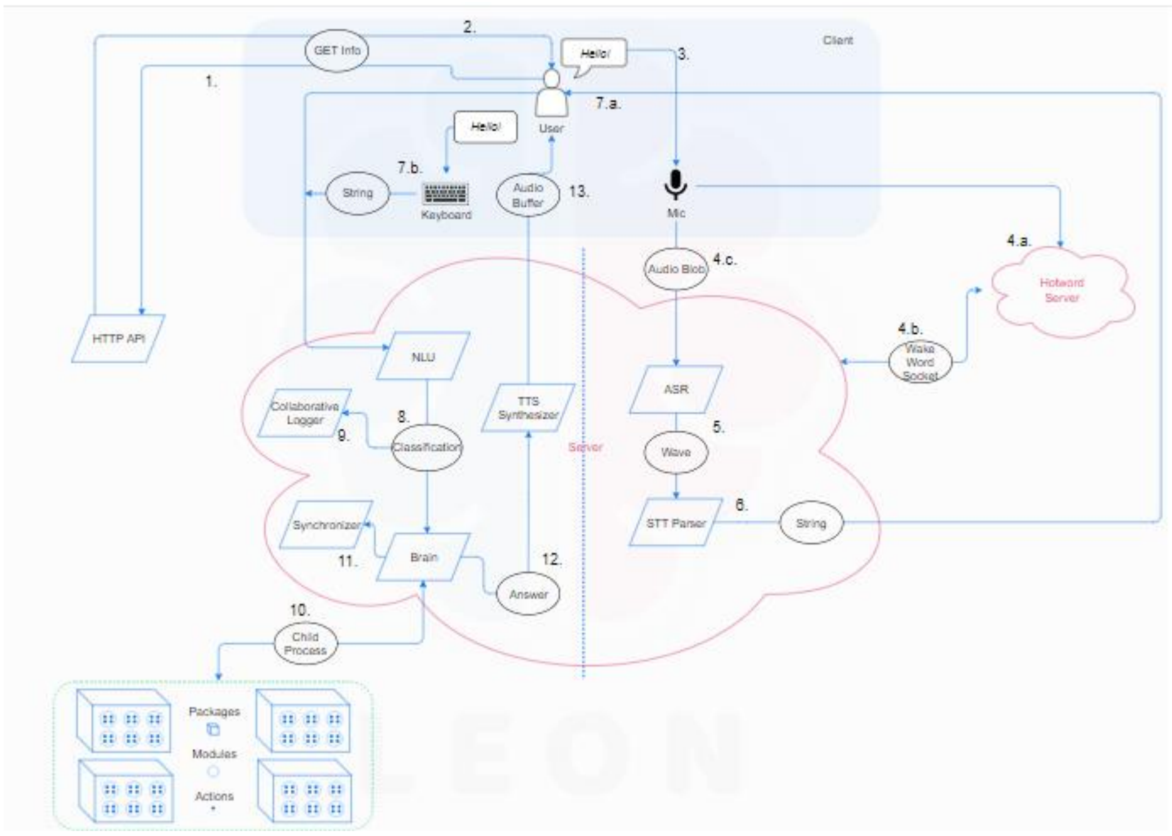
2. Classification algorithms

Several logic adapters in ChatterBot use naive Bayesian classification algorithms to determine if an input statement meets a particular set of criteria that warrant a response to be generated from that logic adapter.

2. Leon https://docs.getleon.ai/#what-is-leon

Leon is an **open-source personal assistant** who can live on your server.

Architecture



API

API stands for *Application Programming Interface* and it allows the communication between different nodes of a project. These nodes can be a server, library, etc.

ASR

ASR or *Automatic Speech Recognition* is the use of computer hardware and software-based techniques to identify and process human voice.

Answers

Answers are Leon's responses. Each package has its own set of answers with different translations.

Brain

Leon's brain is a major part of his core. This is where he executes his modules, talks, picks up sentences, etc.

Classifier

A classifier is a type of model. Once trained, it outputs a result via an algorithm. This result is used to make decisions.

Expressions

Expressions are the dataset that Leon uses to train his understanding.
Each package has its own dataset with different translations.

Modules

Modules are Leon's skills; thanks to them Leon can work his magic. Modules contain one or an infinity of actions.

NLU

NLU (*Natural Language Understanding*) helps computers understand human language.

Leon employs it to load the most appropriate classifier.

Packages

Leon's packages contain one or an infinity of modules. You can consider packages as a category of modules. This is where the answers and expressions are stored.

STT

STT, or *Speech-To-Text*, transforms an audio stream (speech) to a string (text).

Leon has multiple STT parsers; you can choose one (or several) to configure.

Synchronizer

The synchronizer allows you to synchronize your content via different methods (Google Drive, on your current device, etc.) restricted by the requested module's offerings.

TTS

TTS or (*Text-To-Speech*) transforms a string (text) to an audio stream (speech).

Leon has multiple TTS synthesizers; you can choose one (or several) to configure

Scenario

This scenario describes the steps of the above schema. Please note that most interactions are done through WebSockets.

1. Client (web app, etc.) makes an HTTP request to GET some information about Leon.
2. HTTP API responds information to client.
3. User talks with their microphone.
4. .
   - a. If hotword server is launched, Leon listens (offline) if user is calling him by saying `Leon`.
   - b. If Leon understands user is calling him, Leon emits a message to the main server via a WebSocket. Now Leon is listening (offline) to user.
   - c. User said `Hello!` to Leon, client transforms the audio input to an audio blob.
5. ASR transforms audio blob to a wave file.
6. STT parser transforms wave file to string (`Hello`).
7. .
   - a. User receives string and string is forwarded to NLU.
   - b. Or user type `Hello!` with their keyboard (and ignores steps 1. to 7.a.). `Hello!` string is forwarded to NLU.
8. NLU classifies string and pick up classification.
9. If collaborative logger is enabled, classification is sent to collaborative logger.
10. Brain creates a child process and executes the chosen module.
11. If synchronizer is enabled and module has this option, it synchronizes content.
12. Brain creates an answer and forwards it to TTS synthesizer.
13. TTS synthesizer transforms text answer (and send it to user as text) to audio buffer which is played by client.

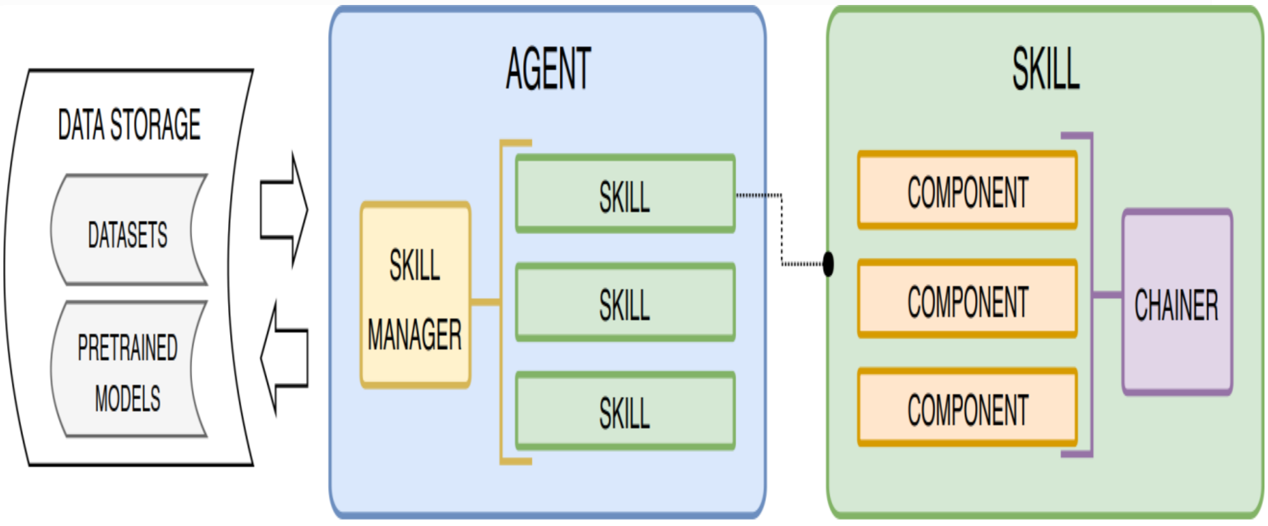
3. DeepPavlov https://github.com/deepmipt/DeepPavlov

open-source conversational AI library

Conceptual overview

Our goal is to enable AI-application developers and researchers with:

- set of pre-trained NLP models, pre-defined dialog system components (ML/DL/Rule-based) and pipeline templates;
- a framework for implementing and testing their own dialog models;
- tools for application integration with adjacent infrastructure (messengers, helpdesk software etc.);
- benchmarking environment for conversational models and uniform access to relevant datasets.



Question Answering Model for SQuAD dataset

Task definition

Question Answering on SQuAD dataset is a task to find an answer on question in a given context (e.g, paragraph from Wikipedia), where the answer to each question is a segment of the context:

Models

There are two models for this task in DeepPavlov: BERT-based and R-Net. Both models predict answer start and end position in a given context. Their performance is compared in pretrained models section of this documentation.

BERT

Pretrained BERT can be used for Question Answering on SQuAD dataset just by applying two linear transformations to BERT outputs for each subtoken. First/second linear transformation is used for prediction of probability that current subtoken is start/end position of an answer.

BERT for SQuAD model documentation **BertSQuADModel**

R-Net

Question Answering Model is based on R-Net, proposed by Microsoft Research Asia ("R-NET: Machine Reading Comprehension with Self-matching Networks") and its implementation by Wenxuan Zhou.

Model usage from Python

```
from deeppavlov import build_model, configs


model = build_model(configs.squad.squad, download=True)

model(['DeepPavlov is library for NLP and dialog systems.'], ['What is DeepPavlov?'])
```

Model usage from CLI

Training

Warning: training with default config requires about 10Gb on GPU. Run following command to train the model:

```
python -m deeppavlov train deeppavlov/configs/squad/squad_bert.json
```

Interact mode

Interact mode provides command line interface to already trained model.

To run model in interact mode run the following command:

```
python -m deeppavlov interact deeppavlov/configs/squad/squad_bert.json
```

Model will ask you to type in context and question.

Pretrained models:

SQuAD

We have all pretrained model available to download:

```
python -m deeppavlov download deeppavlov/configs/squad/squad_bert.json
```

It achieves ~88 F-1 score and ~80 EM on SQuAD-v1.1 dev set.

In the following table you can find comparison with published results. Results of the most recent competitive solutions could be found on SQuAD Leadearboad.

Model (single model)	EM (dev)	F-1 (dev)
DeepPavlov BERT	80.88	88.49
DeepPavlov R-Net	71.49	80.34

Model (single model)	EM (dev)	F-1 (dev)
BiDAF + Self Attention + ELMo	–	85.6
QANet	75.1	83.8
FusionNet	75.3	83.6
R-Net	71.1	79.5
BiDAF	67.7	77.3

Classification models in DeepPavlov

In DeepPavlov one can find code for training and using classification models which are implemented as a number of different **neural networks** or **sklearn models**. Models can be used for binary, multi-class or multi-label classification. List of available classifiers (more info see below):

- **BERT classifier** (see here) builds BERT 8 architecture for classification problem on Tensorflow.
- **Keras classifier** (see here) builds neural network on Keras with tensorflow backend.
- **Sklearn classifier** (see here) builds most of sklearn classifiers.

Pre-trained models

We also provide with **pre-trained models** for classification on DSTC 2 dataset, SNIPS dataset, “AG News” dataset, “Detecting Insults in Social Commentary”, Twitter sentiment in Russian dataset.

DSTC 2 dataset does not initially contain information about **intents**, therefore, `Dstc2IntentsDatasetIterator` (`deeppavlov/dataset_iterators/dstc2_intents_iterator.py`) instance extracts artificial intents for each user reply using information from acts and slots.

Speech recognition and synthesis (ASR and TTS)

DeepPavlov contains models for automatic speech recognition (ASR) and text synthesis (TTS) based on pre-build modules from NeMo (v0.10.0) - NVIDIA toolkit for defining and building Conversational AI applications. Named arguments for modules initialization are taken from the NeMo config file (please do not confuse with the DeepPavlov config file that defines model pipeline).

Speech recognition

The ASR pipeline is based on Jasper: an CTC-based end-to-end model. The model transcripts speech samples without any additional alignment information. `NeMoASR` contains following modules:

- AudioToMelSpectrogramPreprocessor - uses arguments from `AudioToMelSpectrogramPreprocessor` section of the NeMo config file.
- JasperEncoder - uses arguments from `JasperEncoder` section of the NeMo config file. Needs pretrained checkpoint.
- JasperDecoderForCTC - uses arguments from `JasperDecoder` section of the NeMo config file. Needs pretrained checkpoint.
- GreedyCTCDecoder - doesn't use any arguments.
- `AudioInferDataLayer` - uses arguments from `AudioToTextDataLayer` section of the NeMo config file.

NeMo config file for ASR should contain `labels` argument besides named arguments for the modules above. `labels` is a list of characters that can be output by the ASR model used in model training.

Speech synthesis

The TTS pipeline that creates human audible speech from text is based on Tacotron 2 and Waveglow models. `NeMoTTS` contains following modules:

- TextEmbedding - uses arguments from `TextEmbedding` section of the NeMo config file. Needs pretrained checkpoint.
- Tacotron2Encoder - uses arguments from `Tacotron2Encoder` section of the NeMo config file. Needs pretrained checkpoint.
- Tacotron2DecoderInfer - uses arguments from `Tacotron2Decoder` section of the NeMo config file. Needs pretrained checkpoint.
- Tacotron2Postnet - uses arguments from `Tacotron2Postnet` section of the NeMo config file. Needs pretrained checkpoint.
- `WaveGlow` - uses arguments from `WaveGlowNM` section of the NeMo config file. Needs pretrained checkpoint.
- `GriffinLim` - uses arguments from `GriffinLim` section of the NeMo config file.
- `TextDataLayer` - uses arguments from `TranscriptDataLayer` section of the NeMo config file.

NeMo config file for TTS should contain `labels` and `sample_rate` args besides named arguments for the modules above. `labels` is a list of characters used in TTS model training.

4. Microsoft's Bot Framework

https://github.com/microsoft/botframework-sdk

The framework consists of two major components, their Bot Builder SDK and their NLU system called 'LUIS'.

With the Bot Framework SDK, developers can build bots that converse free-form or with guided interactions including using simple text or rich cards that contain text, images, and action buttons.

Current Bot Framework SDK v4 preview features

- Dialog Generation :: The Bot Framework has a rich collection of conversational building blocks, but creating a bot that feels natural to converse with requires understanding and coordinating across language understanding, language generation and dialog management. To simplify this process and capture best practices, we've created the bf-generate plugin for the Bot Framework CLI tool. The generated dialogs make use of event-driven adaptive dialogs with a rich and evolving set of capabilities.

Channels and Adapters

There are two ways to connect your bot to a client experience:

- **Azure Bot Service Channel** - Language and SDK independent support via Azure Bot Service
- **Bot Framework SDK Adapter** - A per language Adapter component.

Bot Framework ecosystem

Bot Framework Composer

Bot Framework Composer is an integrated development tool for developers and multi-disciplinary teams to build bots and conversational experiences with the Microsoft Bot Framework. Within this tool, you'll find everything you need to build a sophisticated conversational experience.

Botkit

Botkit is a developer tool and SDK for building chat bots, apps and custom integrations for major messaging platforms. Botkit bots `hear()` triggers, `ask()` questions and `say()` replies. Developers can use this syntax to build dialogs - now cross compatible with the latest version of Bot Framework SDK.
In addition, Botkit brings with it 6 platform adapters allowing Javascript bot applications to communicate directly with messaging platforms: Slack, Webex Teams, Google Hangouts, Facebook Messenger, Twilio, and Web chat.

Botkit is part of Microsoft Bot Framework and is released under the MIT Open Source license

Bot Framework Virtual Assistant Solution Accelerator

The Bot Framework Solutions repository is home to the Virtual Assistant Solution Accelerator, which provides a set of templates, solution accelerators and skills to help build sophisticated conversational experiences.

- **Virtual Assistant.** Customers and partners have a significant need to deliver a conversational assistant tailored to their brand, personalized to their users, and made available across a broad range of canvases and devices.

  This brings together all of the supporting components and greatly simplifies the creation of a new bot project including: basic conversational intents, Dispatch integration, QnA Maker, Application Insights and an automated deployment.

- **Skills.** A library of re-usable conversational skill building-blocks enabling you to add functionality to a Bot. We currently provide: Calendar, Email, Task, Point of Interest, Automotive, Weather and News skills. Skills include LUIS models, Dialogs, and integration code delivered in source code form to customize and extend as required.

- **Analytics.** Gain key insights into your bot's health and behavior with the Bot Framework Analytics solutions, which includes: sample Application Insights queries, and Power BI dashboards to understand the full breadth of your bot's conversations with users.

Azure Bot Service

Azure Bot Service enables you to host intelligent, enterprise-grade bots with complete ownership and control of your data. Developers can register and connect their bots to users on Skype, Microsoft Teams, Cortana, Web Chat, and more. [Docs]

- **Direct Line JS Client**: If you want to use the Direct Line channel in Azure Bot Service and are not using the WebChat client, the Direct Line JS client can be used in your custom application. [Readme]

- **Direct Line Speech Channel**: We are bringing together the Bot Framework and Microsoft's Speech Services to provide a channel that enables streamed speech and text bi-directionally from the client to the bot application. To sign up, add the 'Direct Line Speech' channel to your Azure Bot Service.

- **Better isolation for your Bot - Direct Line App Service Extension** : The Direct Line App Service Extension can be deployed as part of a VNET, allowing IT administrators to have more control over conversation traffic and improved latency in conversations due to reduction in the number of hops. Get started with Direct Line App Service Extension here. A VNET lets you create your own private space in Azure and is crucial to your cloud network as it offers isolation, segmentation, and other key benefits.

Bot Framework Emulator

The Bot Framework Emulator is a cross-platform desktop application that allows bot developers to test and debug bots built using the Bot Framework SDK. You can use the Bot Framework Emulator to test bots running locally on your machine or to connect to bots running remotely. [Download latest | Docs]

Bot Framework Web Chat

The Bot Framework Web Chat is a highly customizable web-based client chat control for Azure Bot Service that provides the ability for users to interact with your bot directly in a web page. [Stable release | Docs | Samples]

Bot Framework CLI

The Bot Framework CLI Tools hosts the open source cross-platform Bot Framework CLI tool, designed to support building robust end-to-end development workflows. The Bot Framework CLI tool replaced the legacy standalone tools used to manage bots and related services. BF CLI aggregates the collection of cross-platform tools into one cohesive and consistent interface.

Related Services

Language Understanding

A machine learning-based service to build natural language experiences. Quickly create enterprise-ready, custom models that continuously improve. Language Understanding Service(LUIS) allows your application to understand what a person wants in their own words. [Docs | Add language understanding to your bot]

QnA Maker

QnA Maker is a cloud-based API service that creates a conversational, question-and-answer layer over your data. With QnA Maker, you can build, train and publish a simple question and answer bot based on FAQ URLs, structured documents, product manuals or editorial content in minutes. [Docs | Add qnamaker to your bot]

Dispatch

Dispatch tool lets you build language models that allow you to dispatch between disparate components (such as QnA, LUIS and custom code). [Readme]

Speech Services

Speech Services convert audio to text, perform speech translation and text-to-speech with the unified Speech services. With the speech services, you can integrate speech into your bot, create custom wake words, and author in multiple languages. [Docs]

Adaptive Cards

Adaptive Cards are an open standard for developers to exchange card content in a common and consistent way, and are used by Bot Framework developers to create great cross-channel conversatational experiences.

- **Open framework, native performance** - A simple open card format enables an ecosystem of shared tooling, seamless integration between apps, and native cross-platform performance on any device.
- **Speech enabled from day one** - We live in an exciting era where users can talk to their devices. Adaptive Cards embrace this new world and were designed from the ground up to support these new experiences.

5. Rasa https://github.com/RasaHQ/rasa

Rasa is an open source machine learning framework to automate text-and voice-based conversations. With Rasa, you can build contextual assistants on:

- Facebook Messenger
- Slack
- Google Hangouts
- Webex Teams
- Microsoft Bot Framework
- Rocket.Chat
- Mattermost
- Telegram
- Twilio
- Your own custom conversational channels

or voice assistants as:

- Alexa Skills
- Google Home Actions

Rasa helps you build contextual assistants capable of having layered conversations with lots of back-and-forth. In order for a human to have a meaningful exchange with a contextual assistant, the assistant needs to be able to use context to build on things that were previously discussed – Rasa enables you to build assistants that can do this in a scalable way.

NLU Training Data https://rasa.com/docs/rasa/user-guide/rasa-tutorial/

The first piece of a Rasa assistant is an NLU model. NLU stands for Natural Language Understanding, which means turning user messages into structured data. To do this with Rasa, you provide training examples that show how Rasa should understand user messages, and then train a model by showing it those examples.

Rasa's job will be to predict the correct intent when your users send new, unseen messages to your assistant.

Model Configuration

The configuration file defines the NLU and Core components that your model will use. In this example, your NLU model will use the `supervised_embeddings` pipeline.

The language and pipeline keys specify how the NLU model should be built. The policies key defines the policies that the Core model will use.

4. Write Your First Stories

At this stage, you will teach your assistant how to respond to your messages. This is called dialogue management, and is handled by your Core model.

Core models learn from real conversational data in the form of training "stories". A story is a real conversation between a user and an assistant. Lines with intents and entities reflect the user's input and action names show what the assistant should do in response.

5. Define a Domain

The next thing we need to do is define a Domain. The domain defines the universe your assistant lives in: what user inputs it should expect to get, what actions it should be able to predict, how to respond, and what information to store.

Rasa Core's job is to choose the right action to execute at each step of the conversation. In this case, our actions simply send a message to the user. These simple utterance actions are the `actions` in the domain that start with `utter_`. The assistant will respond with a message based on a response from the `responses` section. See Custom Actions to build actions that do more than just send a message.

6. Train a Model

Anytime we add new NLU or Core data, or update the domain or configuration, we need to re-train a neural network on our example stories and NLU data. To do this, run the command below. This command will call the Rasa Core and NLU train functions and store the trained model into the `models/` directory. The command will automatically only retrain the different model parts if something has changed in their data or configuration.

```
rasa train
```

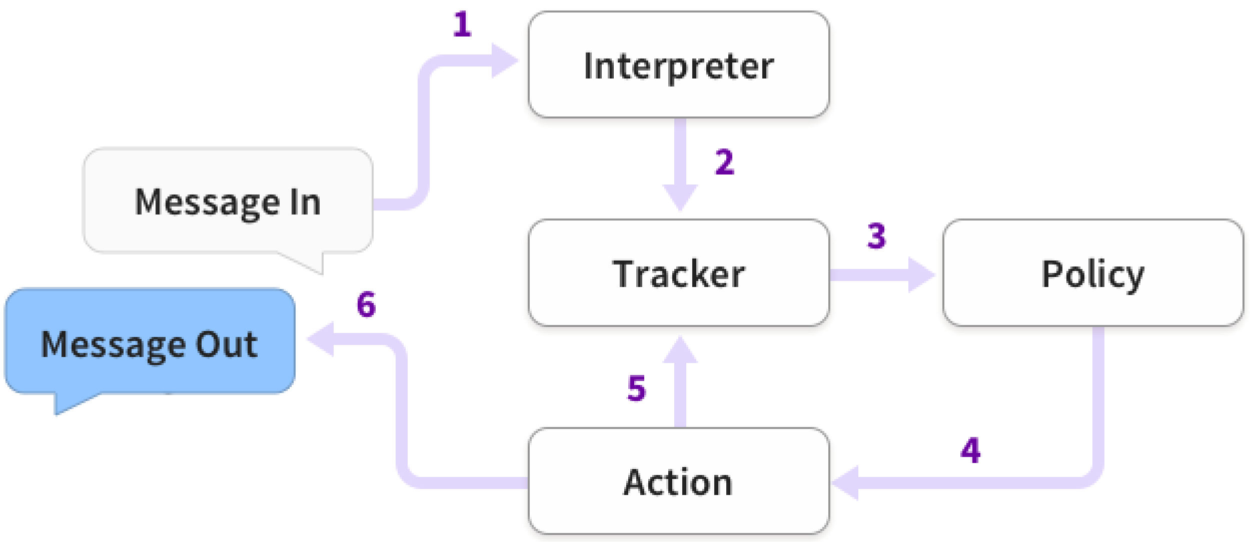
```
echo "Finished training."
```

Architecture

SUGGEST EDITS

Message Handling

This diagram shows the basic steps of how an assistant built with Rasa responds to a message:



The steps are:

1. The message is received and passed to an Interpreter, which converts it into a dictionary including the original text, the intent, and any entities that were found. This part is handled by NLU.
2. The Tracker is the object which keeps track of conversation state. It receives the info that a new message has come in.
3. The policy receives the current state of the tracker.
4. The policy chooses which action to take next.
5. The chosen action is logged by the tracker.
6. A response is sent to the user.

6.Errbot

https://errbot.readthedocs.io/en/latest/

Errbot is a chatbot, a daemon that connects to your favorite chat service and brings your tools into the conversation.

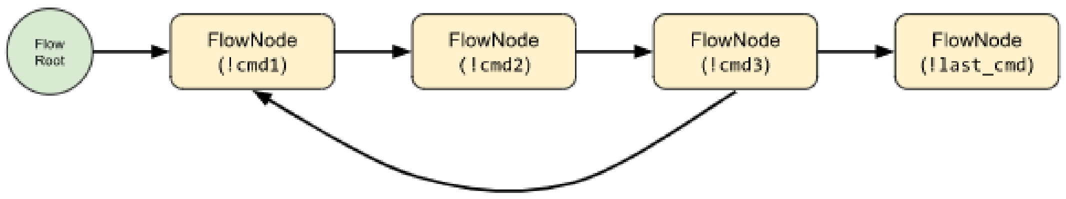
Core features

- Multi User Chatroom (MUC) support
- A dynamic plugin architecture: Bot admins can install/uninstall/update/enable/disable plugins dynamically just by chatting with the bot
- Advanced security/access control features (see below)
- A *!help* command that dynamically generates documentation for commands using the docstrings in the plugin source code
- A per-user command history system where users can recall previous commands
- The ability to proxy and route one-to-one messages to MUC so it can enable simpler XMPP notifiers to be MUC compatible (for example the Jira XMPP notifier)

1.Flows Concepts

1.1. Static structure

Flows are represented as graphs. Those graphs have a root (FlowRoot), which is basically their entry point, and are composed of nodes (FlowNodes). Every node represents an Errbot command.



Example of a flow construction.

.2. Execution

At execution time, Errbot will keep track of who started the flow, and at what step (node) it is currently. On top of that, Errbot will initialize a context for the entire conversation. The context is a simple Python dictionary and it is attached to only one conversation. Think of this like the persistence for plugins, but linked to a conversation only.

If you don't specify any predicate when you build your flow, every single step is "manual". It means that Errbot will wait for the user to execute one of the possible commands at every step to advance along the graph.

Predicates can be used to trigger a command automatically. Predicates are simple functions saying to Errbot, "this command has enough in the context to be able to execute without any user intervention". At any time if a predicate is verified after a step is executed, Errbot will proceed and execute the next step.

2. Basic Flow Definition

2.1. Flows are like plugins

They are defined by a `.flow` file, similar to the plugin ones:

```
[Core]

Name = MyFlows

Module = myflows



[Documentation]

Description = my documentation.



[Python]

Version = 2+
```

Now in the `myflows.py` file you will have pretty familiar structure with a `BotFlow` as type and @botflow as flow decorator:

```
from errbot import botflow, FlowRoot, BotFlow



class MyFlows(BotFlow):

    """ Conversation flows for Errbot"""
    @botflow
    def example(self, flow: FlowRoot):
        """ Docs for the flow example comes here """
        # [...]
```

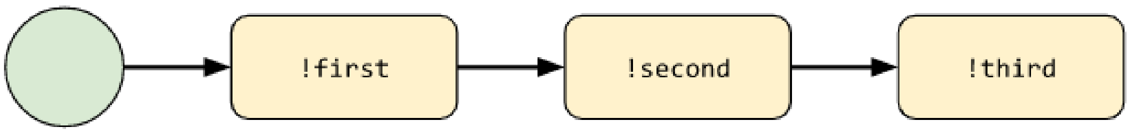
Errbot will pass the root of the flow as the only parameter to your flow definition so you can build your graph from there.

2.2. Making a simple graph

Within your flow, you can connect commands together. For example, to make a simple linear flow between `!first`, `!second` and `!third`:

```
@botflow
def example(self, flow: FlowRoot):
    first_step = flow.connect('first')           # first is a command
name from any loaded plugin.
    second_step = first_step.connect('second')
    third_step = second_step.connect('third')
```

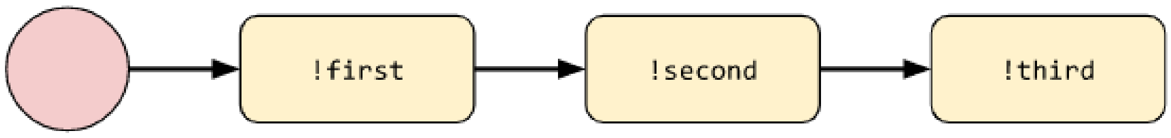
You can represent this flow like this:



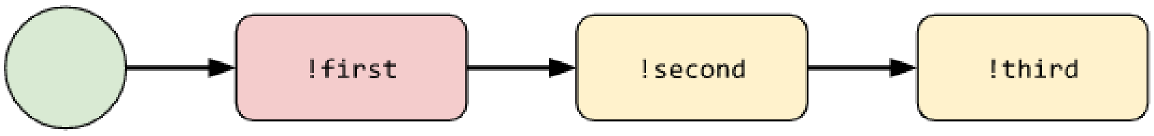
O is the state “not started” for the flow `example`.

You can start this flow manually by doing `!flows start example`.

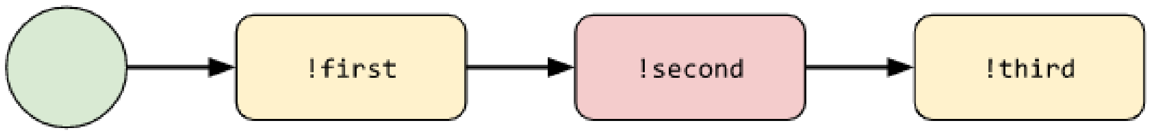
The bot will tell you that it expects a `!first` command:



Once you have executed `!first`, you will be in that state:



The bot will tell you that it expects `!second`, etc.



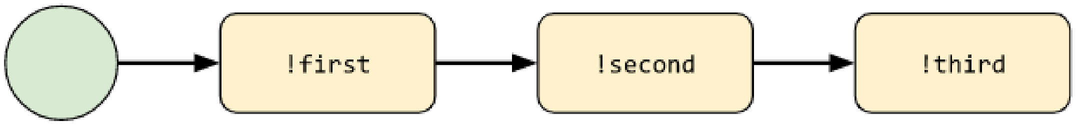
2.3. Making a flow start automatically

Now, usually flows are linked to a first action your users want to do. For example: `!poll new`, `!vm create`, `!report init` or first commands like that that suggests that you will have a follow-up.

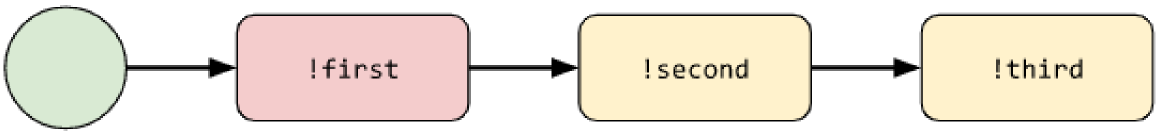
To trigger a flow automatically on those first commands, you can use `auto_trigger`.

```
@botflow

def example(self, flow: FlowRoot):

    first_step = flow.connect('first', auto_trigger=True)

    second_step = first_step.connect('second')

    third_step = second_step.connect('third')
```

You can still represent this flow like this:



BUT, when a user will execute a `!first` command, the bot will instantly instantiate a Flow in this state:

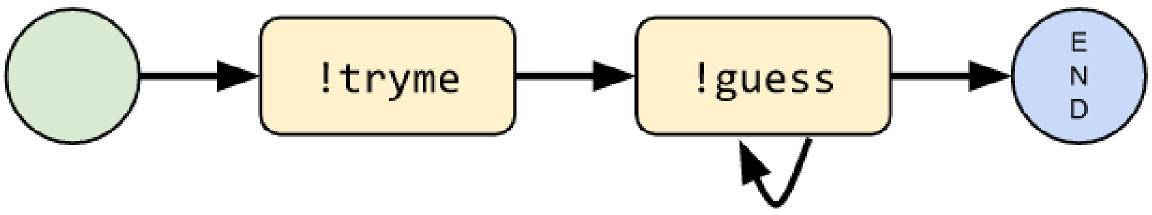


And tell the user that `!second` is the follow-up.

2.4. Flow ending

If a node has no more children and a user passed it, it will automatically end the flow.

Sometimes, with loops etc., you might want to explicitly mark an END FlowNode with a predicate. You can do it like this, for example for a guessing game plugin:



In the flow code…

```
from errbot import botflow, FlowRoot, BotFlow, FLOW_END


class GuessFlows(BotFlow):

    """ Conversation flows related to polls"""
```

```
@botflow
def guess(self, flow: FlowRoot):
    """ This is a flow that can set a guessing game."""
    # setup Flow
    game_created = flow.connect('tryme', auto_trigger=True)
    one_guess = game_created.connect('guessing')
    one_guess.connect(one_guess)  # loop on itself
    one_guess.connect(FLOW_END, predicate=lambda ctx: ctx['ended'])
```

Architecture

Backends are just a specialization of the bot, they are what is instanciated as the bot and are the entry point of the bot.

Following this logic a backend must inherit from **ErrBot**.

ErrBot inherits itself from **Backend**. This is where you can find what Errbot is expecting from backend.

You'll see a series of methods simply throwing the exception **NotImplementedError**. Those are the one you need to implement to fill up the blanks.

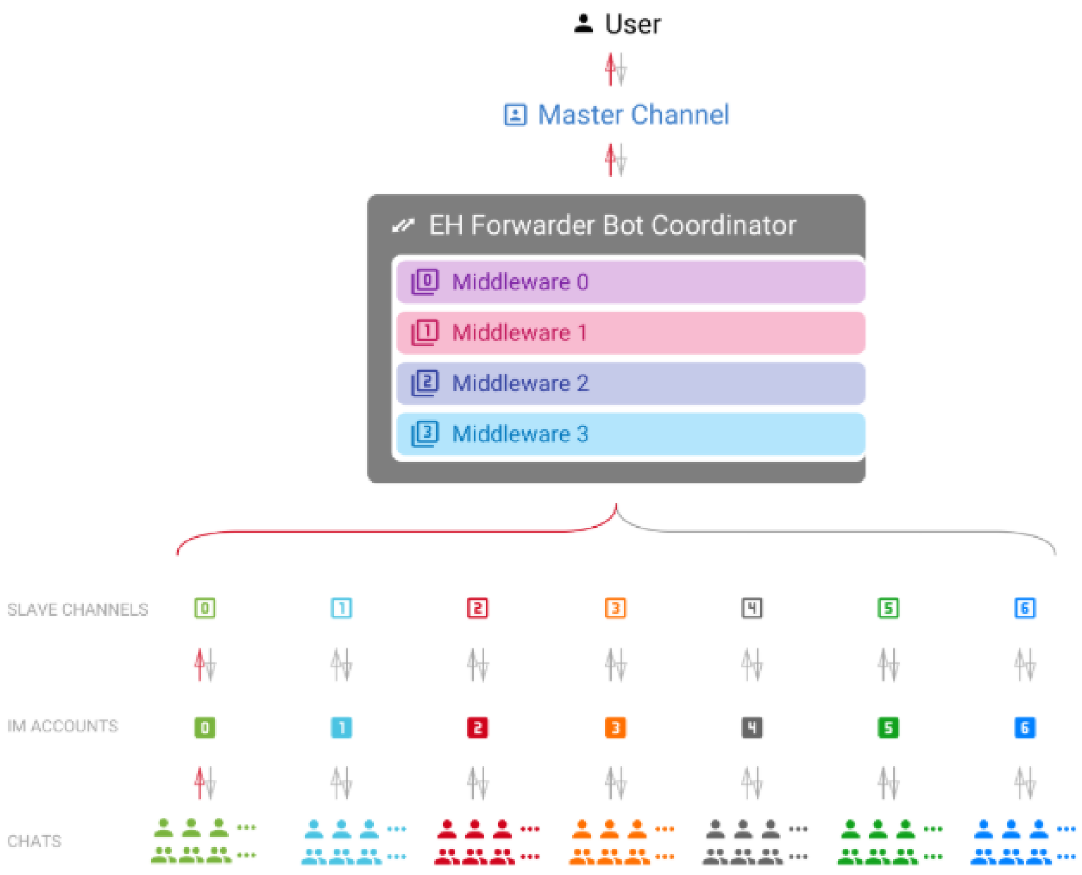
7.EH Forwarder Bot

https://ehforwarderbot.readthedocs.io/en/latest/getting-started.html

EH Forwarder Bot (EFB) is an extensible message tunneling chat bot framework which delivers messages to and from multiple platforms and remotely control your accounts.

Walk-through — How EFB works

EH Forwarder Bot is an extensible framework that allows user to control and manage accounts from different chat platforms in a unified interface. It consists of 4 parts: a Master Channel, some Slave Channels, some Middlewares and a Coordinator.



master channel

The channel that directly interact with [the User](). It is guaranteed to have one and only one master channel in an EFB session.

slave channel

The channel that delivers messages to and from their relative platform. There is at lease one slave channel in an EFB session.

coordinator

Component of the framework that maintains the instances of channels, and delivers messages between channels.

middleware

Module that processes messages and statuses delivered between channels, and make modifications where needed.

Concepts to know

module

A common term that refers to both channels and middlewares.

the User

the User Themself

This term 1 can refer to the user of the current instance of EH Forwarder Bot, operating the master channel, and the account of an IM platform logged in by a slave channel.

chat

A place where conversations happen, it can be either a private chat, a group chat, or a system chat.

private chat

A conversation with a single person on the IM platform. Messages from a private conversation shall only has an author of the User Themself, the other person, or a “system member”.

For platforms that support bot or something similar, they would also be considered as a “user”, unless messages in such chat can be sent from any user other than the bot.

For chats that the User receive messages, but cannot send message to, it should also be considered as a private chat, only to raise an exception when messages was trying to send to the chat.

group chat

A chat that involves more than two members. A group chat MUST provide a list of members that is involved in the conversation.

system chat

A chat that is a part of the system. Usually used for chats that are either a part of the IM platform, the slave channel, or a middleware. Slave channels can use this chat type to send system message and notifications to the master channel.

chat member

A participant of a chat. It can be the User Themself, another person or bot in the chat, or a virtual one created by the IM platform, the slave channel, or a middleware.

message

Messages are delivered strictly between the master channel and a slave channel. It usually carries an information of a certain type.

Each message should at least have a unique ID that is distinct within the slave channel related to it. Any edited message should be able to be identified with the same unique ID.

status

Information that is not formatted into a message. Usually includes updates of chats and members of chats, and removal of messages.

Slave Channels

The job of slave channels is relatively simple.

1. Deliver messages to and from the master channel.
2. Maintains a list of all available chats, and group members.
3. Monitors changes of chats and notify the master channel.

Features that does not fit into the standard EFB Slave Channel model can be offered as Additional features.

Master Channels

Master channels is relatively more complicated and also more flexible. As it directly faces the User, its user interface should be user-friendly, or at least friendly to the targeted users.

The job of the master channel includes:

1. Receive, process and display messages from slave channels.
2. Display a full list of chats from all slave channels.
3. Offer an interface for the User to use “extra functions” from slave channels.
4. Process updates from slave channels.
5. Provide a user-friendly interface as far as possible.

Middlewares

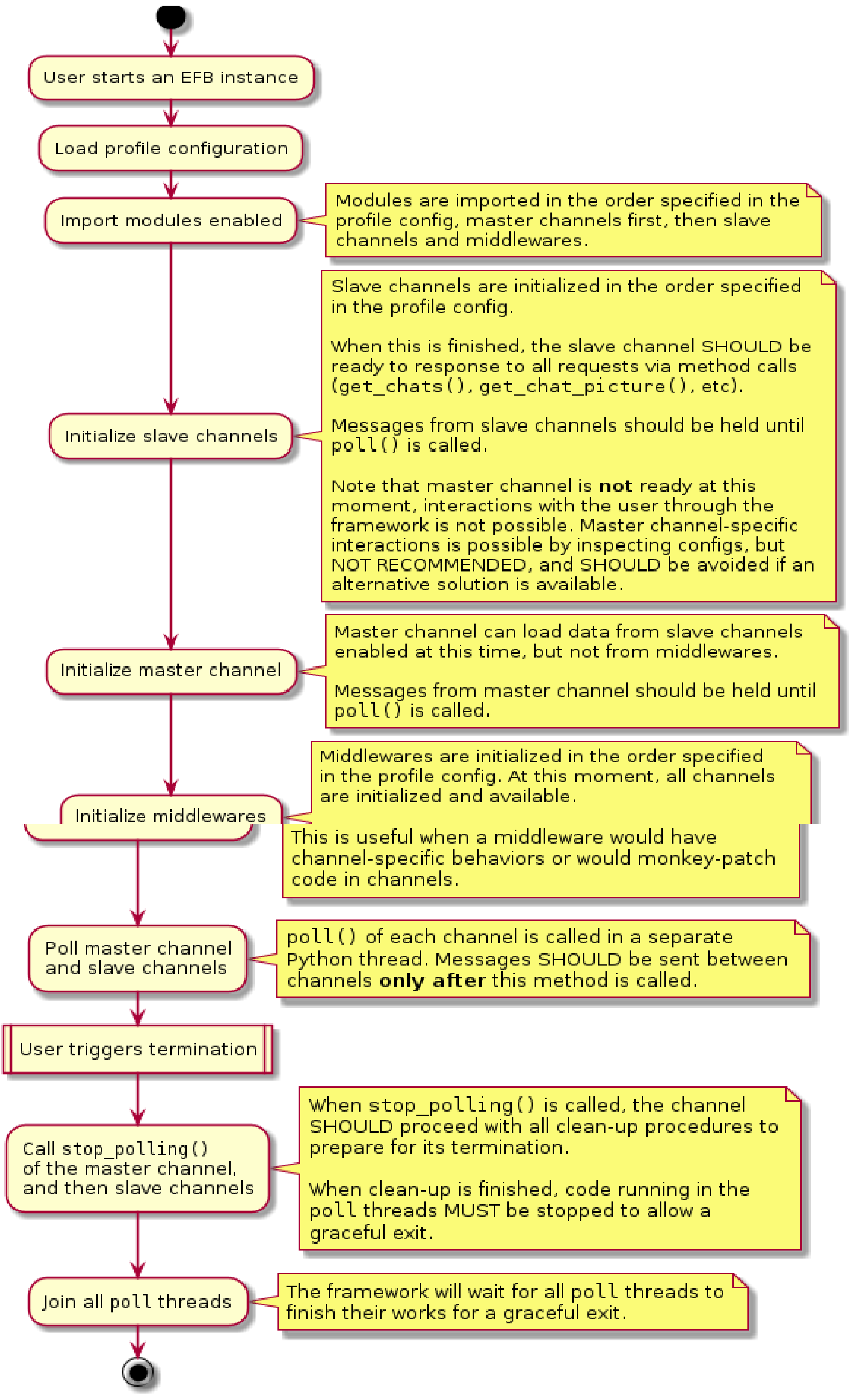
Middlewares can monitor and make changes to or nullify messages and statuses delivered between channels. Middlewares are executed in order of registration, one after another. A middleware will always receive the messages processed by the preceding middleware if available. Once a middleware nullify a message or status, the message will not be processed and delivered any further.

Lifecycle

This section talks about the lifecycle of an EFB instance, and that of a message / status.

Lifecycle of an EFB instance

The diagram below outlines the lifecycle of an EFB instance, and how channels and middlewares are involved in it.


User starts an EFB instance
Load profile configuration
Import modules enabled
Initialize slave channels
Initialize master channel
Initialize middlewares
Poll master channel and slave channels
User triggers termination
Call stop_polling() of the master channel, and then slave channels
Join all poll threads

Modules are imported in the order specified in the profile config, master channels first, then slave channels and middlewares.

Slave channels are initialized in the order specified in the profile config.

When this is finished, the slave channel SHOULD be ready to response to all requests via method calls (`get_chats()`, `get_chat_picture()`, etc).

Messages from slave channels should be held until `poll()` is called.

Note that master channel is **not** ready at this moment, interactions with the user through the framework is not possible. Master channel-specific interactions is possible by inspecting configs, but NOT RECOMMENDED, and SHOULD be avoided if an alternative solution is available.

Master channel can load data from slave channels enabled at this time, but not from middlewares.

Messages from master channel should be held until `poll()` is called.

Middlewares are initialized in the order specified in the profile config. At this moment, all channels are initialized and available.

This is useful when a middleware would have channel-specific behaviors or would monkey-patch code in channels.

`poll()` of each channel is called in a separate Python thread. Messages SHOULD be sent between channels **only after** this method is called.

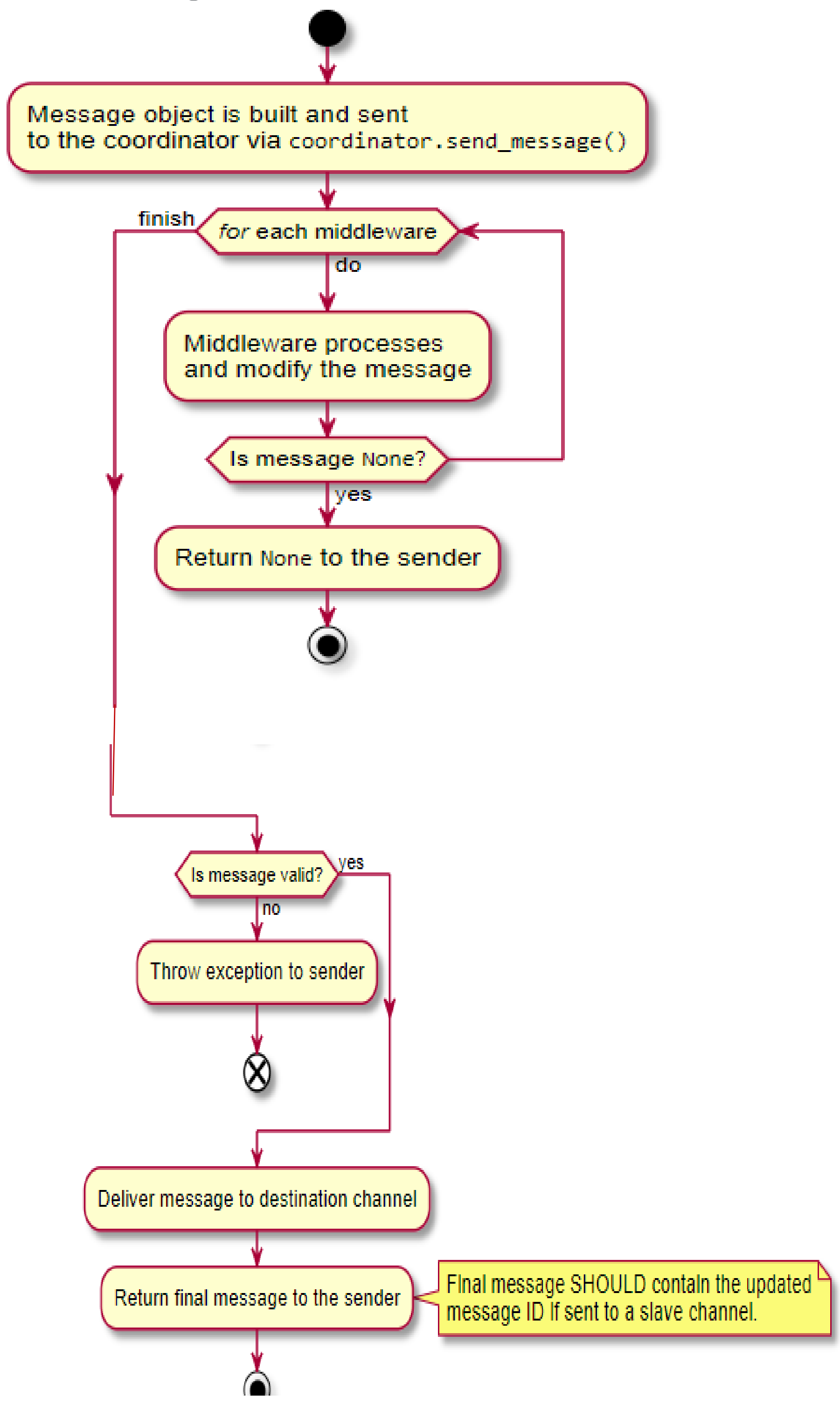
When `stop_polling()` is called, the channel SHOULD proceed with all clean-up procedures to prepare for its termination.

When clean-up is finished, code running in the `poll` threads MUST be stopped to allow a graceful exit.

The framework will wait for all `poll` threads to finish their works for a graceful exit.

Lifecycle of a message

The diagram below outlines the lifecycle of a message sending from a channel, going through all middlewares, sent to the destination channel, and returned back to the sending chan


Message object is built and sent
to the coordinator via coordinator.send_message()
finish
for each middleware
do
Middleware processes
and modify the message
Is message None?
yes
Return None to the sender
Is message valid?
yes
no
Throw exception to sender
Deliver message to destination channel
Return final message to the sender
Final message SHOULD contain the updated
message ID if sent to a slave channel.

Lifecycle of a message

8.Sevabot https://sevabot-skype-bot.readthedocs.io/en/latest/

Sevabot is a generic purpose hack-it-together Skype bot

- Has extensible command system based on UNIX scripts
- Send chat messages from anywhere using HTTP requests and webhooks
- Bult-in support for Github commit notifications and other popular services

It is based on Skype4Py framework

`Skype4Py` is a Python library which allows you to control Skype client application.
It works on Windows, OSX and Linux platforms with Python 2.x versions.

The bot is written in Python 2.7.x programming language, but can be integrated with any programming languages over UNIX command piping and HTTP interface.

The underlying Skype4Py API is free - **you do not need to enlist and pay Skype development program fee**.

Use cases

Developer oriented use cases include

- Get monitoring alerts to Skype from monitoring system like Zabbix
- Get alerts from continuous integration system build fails (Travis CI, Jenkins)
- Get notifications of new commits and issues in your software project (Git, SVN)
- Control production deployments from Skype chat with your fellow developers with in-house scripts

Benefits

Skype is the most popular work related chat program around the world. Skype is easy: anyone can use Skype.

Skype group chat provides noise-free medium with a context. People follow Skype more actively than email; the discussion in the group chat around the notification messages feels natural.

For example our organization has an admin group chat where the team members get notifications what other people are doing (commits, issues) and when something goes wrong (monitoring). This provides pain free follow up of the daily tasks.

Creating custom commands

The bot can use any UNIX executables printing to stdout as commands

- Shell scripts
- Python scripts, Ruby scripts, etc.

All commands must be in one of *modules* folders of the bot. The bot comes with some built-in commands like `ping`, but you can add your own custom commands by

- There is a `custom/` folder where you can place your own modules
- Enable `custom` folder in settings.py
- Create a a script in `custom` folder. Example `myscript.sh`:

```
#!/bin/sh

echo "Hello world from my sevabot command"
```

- Add UNIX execution bit on the script using `chmod u+x myscript.sh`
- In Sevabot chat, type command `!reload` to relaod all scripts
- Now you should see command `!myscript` in the command list
- The following environment variables are exposed to scripts `SKYPE_USERNAME`, `SKYPE_FULLNAME` from the person who executed the command

Stateful modules

You can have Python modules which maintain their state and have full access to Skype4Py instance. These modules can e.g.

- Perform timed background tasks with Skype
- Parse full Skype chat text, not just !commands
- Reach to calls, initiate calls
- Send SMS, etc.

9. Difference between Personal Assistant and Chatbots

9.1 Intelligence

Chatbots: Chatbots are typically text-based and are programmed to reply to only a certain set of questions or statements. If the question asked is other than the learned set of responses by the customer, they will fail.

Virtual Assistants: On the other hand, Virtual Assistants have a much sophisticated interactive platform. They understand not just the language but also the meaning of what the user is saying. They can learn from instances and provide an unpredictability to their behavior.

9.2 Natural Language Processing

Chatbots: Chatbots are not programmed to respond to a change in use of language. It does not have high language processing skills.

Virtual Assistants: Virtual Assistants mainly concentrate on natural language processing (NLP) and Natural Language Understanding (NLU).

9.3 Tasks

Chatbots: Chatbot has a limited use and does not have sophisticated algorithms in areas customer support and automated purchases baked in**.**

Virtual Assistants: Virtual Assistants have a wider scope and can perform a range of tasks, for example comparing products or finding the best product based on the given features. For example sharing jokes, playing music, stock market updates and even controlling the electronic gadgets in the room.

9.4 Technology

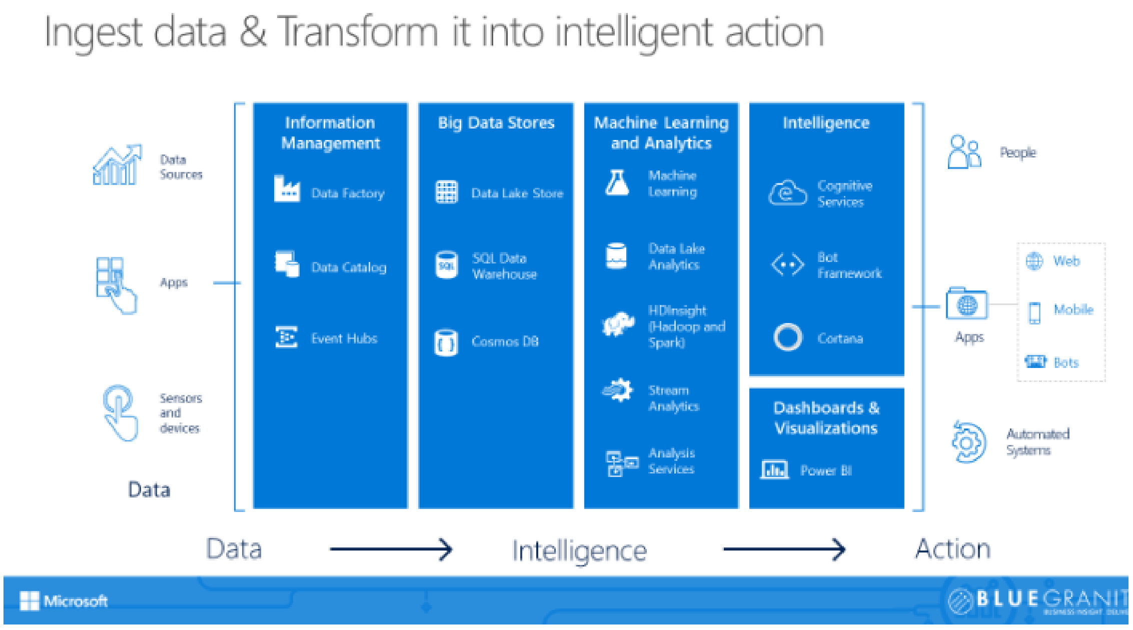
Chatbots: Two chatbot models predominantly used are generative model and the selective model.

The selective model, also called as ranking model, ranks the information by the user with its current memory information and goes into a sequence to come up with the best response.

Virtual Assistants: Virtual assistants use artificial neural networks to learn from situations.

10. Cortana : It is a virtual assistant developed by Microsoft which uses Bing search engine to perform tasks . Cortana is currently available English, Portuguese, French, German, Italian, Chinese and Japanese editions, depending on the software platform and region in which it is used.

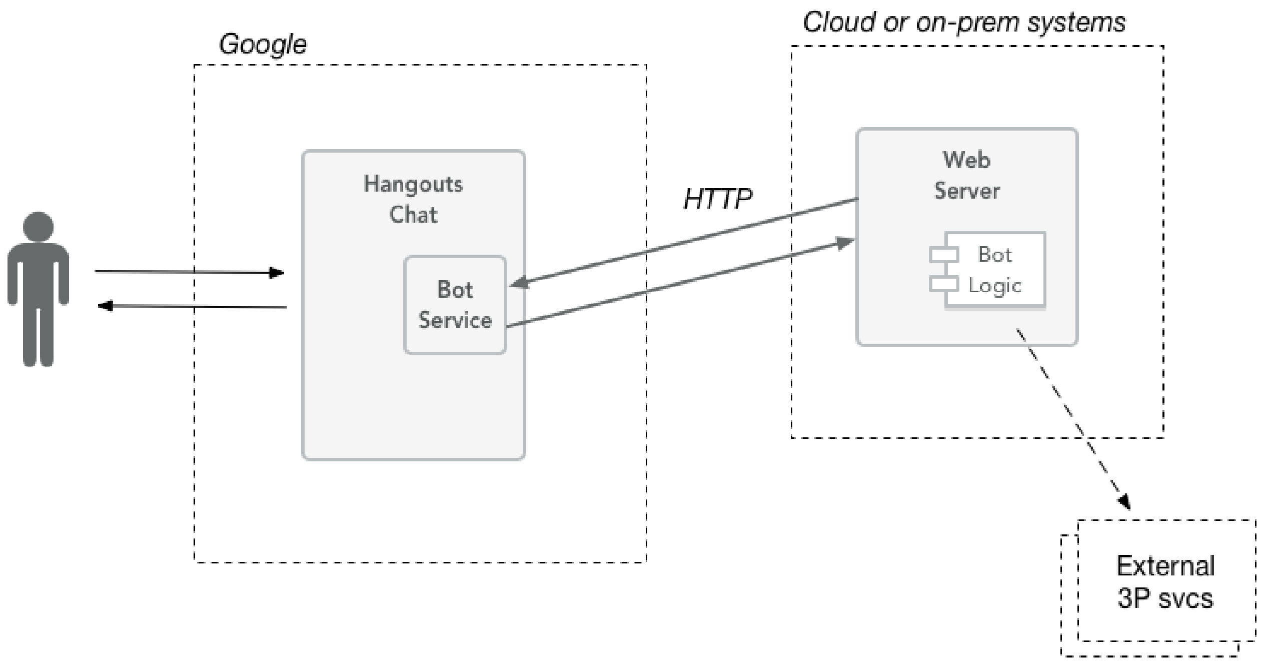
- Microsoft's Cortana assistant is deeply integrated into its Edge browser.
- Cortana can find opening hours when on restaurant sites, show retail coupons for websites, or show weather information in the address bar.
- Cortana stores personal information such as interests, location data, reminders, and contacts in the "Notebook". It can draw upon and add to this data to learn a user's specific patterns and behaviors.
- Cortana has a built-in system of reminders which for example can be associated with a specific contact; it will then remind the user when in communication with that contact, possibly at a specific time or when the phone is in a specific location.
- Cortana on Windows mobile and Android is capable of capturing device notification and sending them to a Windows 10 device.



11. Google Hangouts bot : Hangups bot is a chat bot designed for working with Google Hangouts.

Features

- **Mentions** : If somebody mentions you in a room, receive a private hangout from the bot with details on the mention, including context, room and person who mentioned you**.**
- **Syncouts :** A syncout is two Hangout group chats that have their messages forwarded to each other, allowing seamless interaction between the two rooms. Primarily used to beat the 150-member chat limit, but it can also be used for temporarily connecting teams together to interact.
- **Cross-chat Syncouts :** Half of your team is on Slack? No problem! You can connect them into the same room to communicate. Support for other chat clients coming soon.
- **Hubot Integration:** Hangupsbot allows you to connect to Hubot, instantly providing you access to hundreds of developed chat tools and plugins.
- **Plugins and sinks** : The bot has instructions for developing your own plugins and sinks, allowing the bot to interact with external services such as your company website, Google scripts and much more.
- **Plugin mania** : games, nickname support, subscribed keywords, customizable API – the list goes on!



12. Telegram bot

To name just a few things, you could use bots to:

- **Get customized notifications and news**. A bot can act as a smart newspaper, sending you relevant content as soon as it's published.
- **Integrate with other services**. A bot can enrich Telegram chats with content from external services. Gmail Bot, Gif bot, IMDB bot, Wiki bot, Music bot, YouTube bot, GitHub.
- **Accept payments from Telegram users**. A bot can offer paid services or work as a virtual storefront.
- **Create custom tools**. A bot may provide you with alerts, weather forecasts, translations, formatting or other services. For example Markdown bot, Sticker bot, Vote bot, Like bot
- **Build single- and multiplayer games**. A bot can offer rich HTML5 experiences , from simple arcades and puzzles to 3D-shooters and real-time strategy games. GameBot, Gamee
- **Build social services**. A bot could connect people looking for conversation partners based on common interests or proximity.

How do bots work?

At the core, Telegram Bots are special accounts that do not require an additional phone number to set up. Users can interact with bots in two ways:

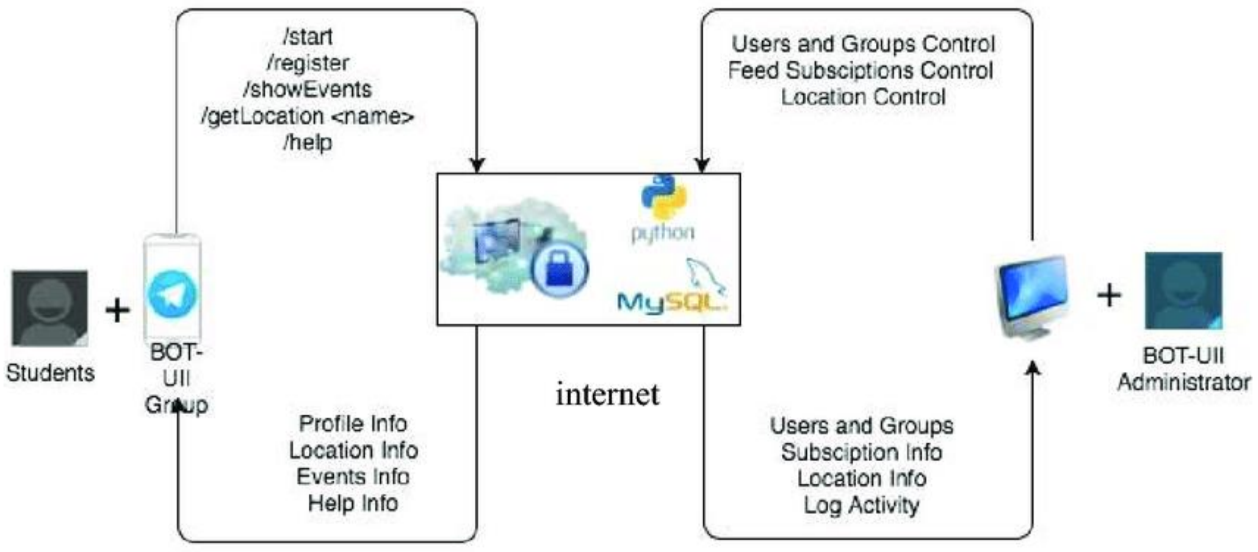
- Send messages and commands to bots by opening a chat with them or by adding them to groups. This is useful for chat bots or news bots like the official TechCrunch bot.
- Send requests directly from the input field by typing the bot's @username and a query. This allows sending content from inline bots directly into any chat, group or channel.

How are bots different from humans?

- Bots have no online status and no last seen timestamps, the interface shows the label **'bot'** instead.
- Bots have limited cloud storage — older messages may be removed by the server shortly after they have been processed.
- Bots can't initiate conversations with users. A user **must** either add them to a group or send them a message first. People can use t.me/<bot_username> links or username search to find your bot.
- Bot usernames always end in 'bot' (e.g. @TriviaBot , @Github_bot).

- When added to a group, bots do not receive all messages by default
- Bots never eat, sleep or complain (unless expressly programmed otherwise).

Diagram Flow



13. WhatsApp Bot

- Made using WhatsApp API.
- The demo bot will be able to react to and reply to commands sent to WhatsApp as regular messages.

Functionality

- The output of the command list.
- The output of the current chat ID.
- The output of the actual server time of the bot running on.
- The output of your name.
- Sending files of different formats (PDF, jpg, doc, mp3, etc.)
- Sending of prerecorded voice messages.
- Sending of geo-coordinates (locations).
- Setting up a conference (group).

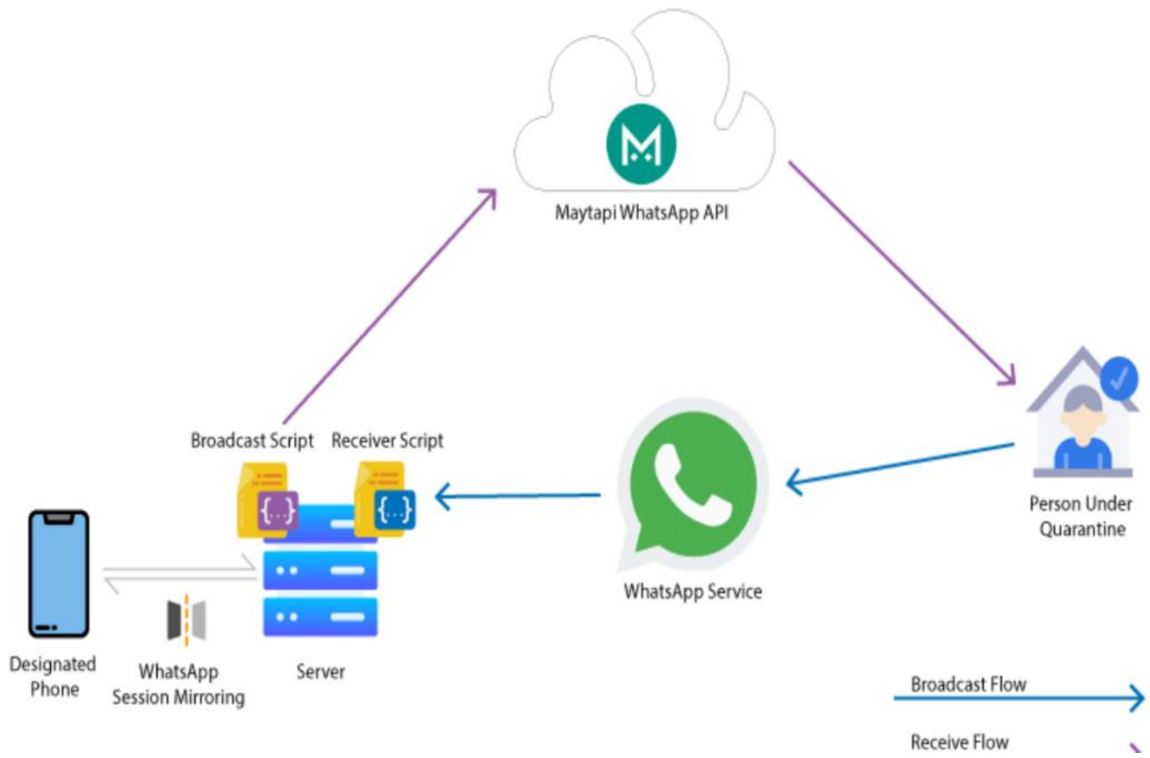
How it works

The Web Hook URL is a link to which JSON-data containing information about incoming messages or notifications will be sent using the POST method. So we need a server to make the bot run and this server will accept and process this information. server allows us to conveniently respond to incoming requests and process them.

There are several ways in which we can make WhatsApp bot one of them is using Python Flask, Twilio SMS API and Heroku.

Steps to create a bot

1. Configure the Twilio WhatsApp Sandbox.
2. Create a Python Virtual Environment.
3. Receiving messages from user end , generally it comes with post method and sending responses for the messages.
4. Writing chatbot logics
5. Deploying the bot on Heroku



Workflow of WhatsApp bot to track COVID-19

14. Facebook Messenger Bot

There are 3 main steps to get this done.

1. Create a server which listens to messages from Facebook (using Flask).
2. Define a function for sending messages back to users (using requests).
3. forward a https connection to your local machine (using ngrok).

- The first step is to create a http server which listens to messages sent by Facebook, gets a response to that message, and eventually sends the response back to the user on Facebook. We will use flask to create this server. The basic idea is the following:

1. Create a flask app that listens for messages sent to localhost:5000/webhook. When messages are sent on Facebook they will arrive as http requests to this URL.

2. The `listen()` function handles these `http` requests and checks that they contain a valid Facebook message.

3. If the message is valid, the `get_bot_response()` function is called and the response is sent back to Facebook Messenger.

In addition, you will create a function called verify_webhook() that handles the initial authentication between Facebook and your app.

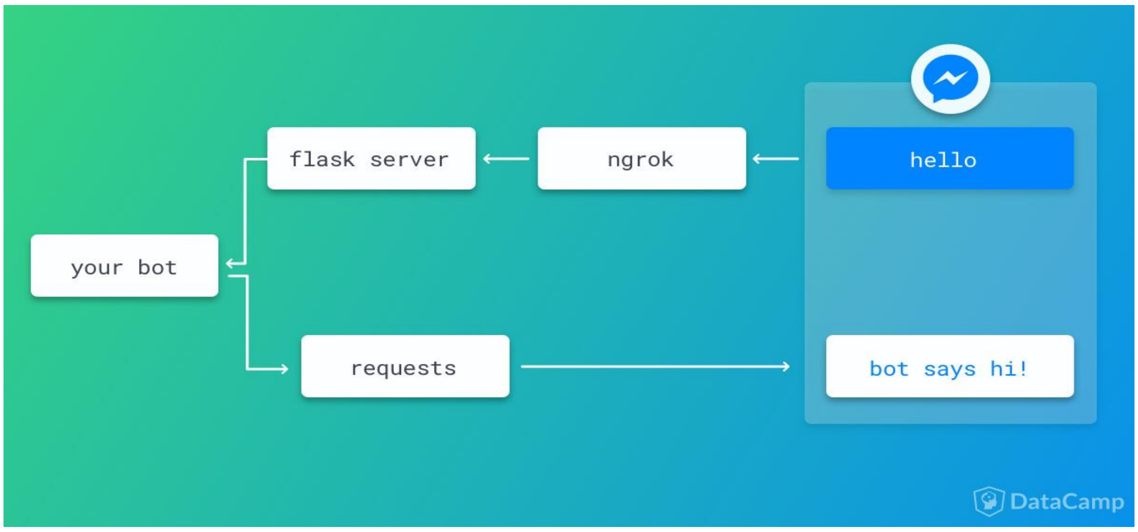
- **Send Messages Back to Users**

Next, you will write a function that that sends a response back to Facebook Messenger using a python library called requests. In particular, you use the post() method, which creates a HTTP POST request. A POST request is the typical way to send information to a server.

- **Start your bot server**

Now that you have everything set up, it's time to start your chatbot server! In a separate terminal tab, run

Your bot is now ready to send and receive messages via Facebook Messenger.



6. Instagram Bots

Requirements: Min. Python 2.7, Working instagram account , instabot.py file must be in UTF-8 encoding if you use Python 3, or ASCII in Python 2 (PEP)!

1.

1) Download and Set up Python and pip to your computer.

```
$ sudo apt-get install python3
```

```
$ sudo apt-get install pip3
```

2) Install python lib request.

for python 3, run $ pip3 install requests

for earlier pythons, run $ pip install requests

3) Install Instabot Git Repository.

```
$ git clone git://github.com/LevPasha/instabot.py.git
```

4) Edit Example.py or create customized python file as per your need. (Next Step)

2. Log in to your Instagram.

3. Run the script.

HELLOHAPTIk/CHATBOT_NER https://github.com/hellohaptik/chatbot_ner
Named Entity Recognition for chatbots

Chatbot NER is an open source framework custom built to supports entity recognition in text messages. After doing thorough research on existing NER systems, team at Haptik felt the strong need of building a framework which is tailored for Conversational AI and also supports Indian languages. Currently Chatbot-ner supports **English, Hindi, Gujarati, Marathi, Bengali and Tamil** and their code mixed form. Currently this framework uses common patterns along with few NLP techniques to extract necessary entities from languages with sparse data. API structure of Chatbot ner is designed keeping in mind usability for Conversational AI applications. Team at Haptik is continuously working towards porting this framework for **all Indian languages and their respective local dialects**.

Supported Entities

Entity type	**Code reference**	**Description**	**example**	**Supported languages - ISO 639-1 code**
Time	TimeDetector	Detect time from given text.	tomorrow morning at 5, कल सुबह ५ बजे, kal subah 5 baje	'en', 'hi', 'gu', 'bn', 'mr', 'ta'
Date	DateAdvancedDetector	Detect date from given text	next monday, agle somvar, अगले सोमवार	'en', 'hi', 'gu', 'bn', 'mr', 'ta'
Number	NumberDetector	Detect number and respective units in given text	50 rs per person, ५ किलो चावल, मुझे एक लीटर ऑइल चाहिए	'en', 'hi', 'gu', 'bn', 'mr', 'ta'
Phone number	PhoneDetector	Detect phone number in given text	9833530536, +91 9833530536, ९८३३४३०५३५	'en', 'hi', 'gu', 'bn', 'mr', 'ta'
Email	EmailDetector	Detect email in text	hello@haptik.co	'en'
Text	TextDetector	Detect custom entities in text string using full text search in Datastore or based on contextual model	Order me a **pizza**, **मुंबई** में मौसम कैसा है	Search supported for 'en', 'hi', 'gu', 'bn', ' model supported for 'en' only
PNR	PNRDetector	Detect PNR (serial) codes in given text.	My flight PNR is 4SGX3E	'en'

Entity type	Code reference	Description	example	Supported languages - ISO 639-1 code
regex	RegexDetector	Detect entities using custom regex patterns	My flight PNR is 4SGX3E	NA

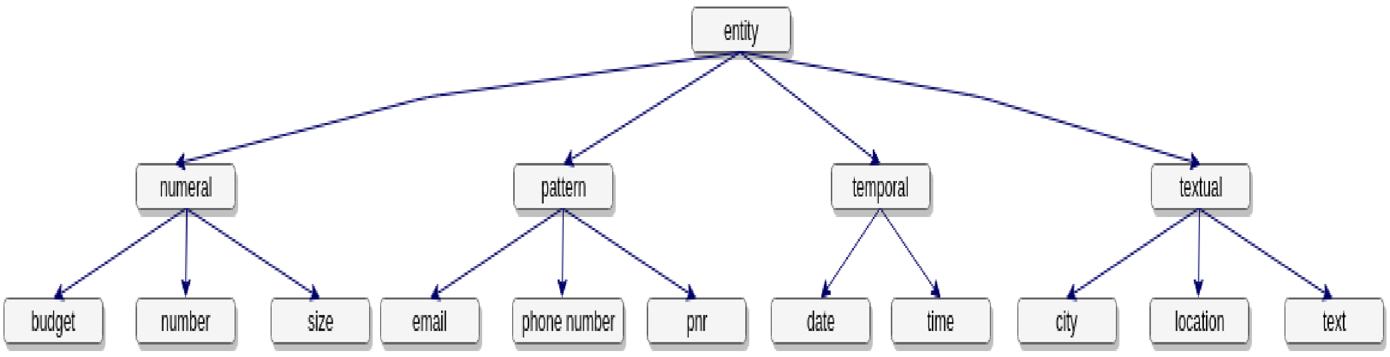
There are other custom detectors such as city, budget shopping size which are derived from above mentioned primary detectors but they are supported currently in English only and limited to Indian users only. We are currently in process of restructuring them to scale them across languages and geography and their current versions might be deprecated in future. So **for applications already in production**, we would recommend you to **use only primary detectors** mentioned in the table above.

API structure

Detailed documentation of APIs for all entity types is available here. Current API structure is built for ease of accessing it from conversational AI applications. However, it can be used for other applications also.

Framework Overview

In any conversational AI application, there are several entities to be identified and logic for detection on one entity might be different from other. We have organised this repository as shown below



We have classified entities into four main types i.e. *numeral*, *pattern*, *temporal* and *textual*.

- numeral: This type will contain all the entities that deal with the numeral or numbers. For example, number detection, budget detection, size detection, etc.
- pattern: This will contain all the detection logics where identification can be done using patterns or regular expressions. For example, email, phone_number, pnr, etc.
- temporal: It will contain detection logics for detecting time and date.
- textual: It identifies entities by looking at the dictionary. This detection mainly contains detection of text (like cuisine, dish, restaurants, etc.), the name of cities, the location of a user, etc.

Numeral, temporal and pattern have been moved to ner_v2 for language portability with more flexible detection logic. In ner_v1, currently only **text** entity has language support. We will be moving it to ner_v2 without any major API changes.

It uses ***CONDITIONAL RANDOM FIELDS***

CONDITIONAL RANDOM FIELDS

A. INTRODUCTION

Conditional random fields (CRFs) are a class of statistical modeling method often applied in pattern recognition and machine learning and used for structured prediction. CRFs fall into the sequence modeling family. Whereas a discrete classifier predicts a label for a single sample without considering "neighboring" samples, a CRF can take context into account; e.g., the linear chain CRF (which is popular in natural language processing) predicts sequences of labels for sequences of input samples.

CRFs are a type of discriminative undirected probabilistic graphical model. They are used to encode known relationships between observations and construct consistent interpretations and are often used for labeling or parsing of sequential data, such as natural language processing or biological sequences and in computer vision. Specifically, CRFs find applications in POS Tagging, shallow parsing, named entity recognition, gene finding and peptide critical functional region finding, among other tasks, being an alternative to the related hidden Markov models (HMMs). In computer vision, CRFs are often used for object recognition and image segmentation.

We have implemented a CRF model for Named Entity Recognition. In addition to commonly used primitive features we have incorporated glove embeddings in order to add semantic knowledge to the Entity Recognition algorithm.

Since the CRF model is used to detect named-entities we thus have assigned CRF Entities are assigned **text** type

B. SETUP

1. Word Embeddings

The module we have deployed incorporates glove embeddings inorder to incorporate semantic meaning to the algorithm.

```
import os
import wget
import zipfile
import gensim
from gensim.test.utils import datapath, get_tmpfile
from gensim.models import KeyedVectors
from gensim.scripts.glove2word2vec import glove2word2vec
import pickle

wget.download(url='http://nlp.stanford.edu/data/glove.twitter.27B.zip')

zip_ref = zipfile.ZipFile('glove.twitter.27B.zip', 'r')
zip_ref.extract(member='glove.twitter.27B.25d.txt')
os.remove('glove.twitter.27B.zip')

glove2word2vec('glove.twitter.27B.25d.txt', 'test_word2vec_new_nearby.txt')
word_vectors = KeyedVectors.load_word2vec_format('test_word2vec_new_nearby.txt')

os.remove('glove.twitter.27B.25d.txt')
os.remove('test_word2vec_new_nearby.txt')
file_handler = open('glove_vocab', 'wb')
pickle.dump(obj=word_vectors.wv.index2word,file=file_handler, protocol=2)
file_handler = open('glove_vectors', 'wb')
pickle.dump(obj=word_vectors.wv.vectors, file=file_handler, protocol=2)

if not os.path.exists('/app/models_crf/'):
    os.makedirs('/app/models_crf/')
```

2. Environment Keys

ADD the following keys to the environment

```
MODELS_PATH=/app/models_crf
EMBEDDINGS_PATH_VOCAB=/app/glove_vocab
EMBEDDINGS_PATH_VECTORS=/app/glove_vectors
```

Add the above specified dir/files to .gitignore

C. TRAINING

1. Input Training Data

Generate training data for CRF model in the CSV format. The csv files should consists of the sentences and their corresponding entities on which the model has to be trained. This CSV is just for convenience. Direct sentence_list and entity_list can be provided as input.

- **sentence_list**

  The columns consists of texts (text patterns) on which the module needs to be trained. The model learns to recognize the the entities present in the sentences in a supervised fashion.

- **entity_list**

  The column consists of the list of entities present in the corresponding sentences.`

Example

sentence_list	entity_list
My name is Pratik Jayarao	["Pratik Jayarao"]
My name is Pratik	["Pratik"]
My name is Pratik Sridatt Jayarao	["Pratik Sridatt Jayarao"]
My name is Pratik and this is my friend Hardik	["Pratik", "Hardik"]

sentence_list	entity_list
People call me Harsh Shah	["Harsh Shah"]
Chirag Jain is my name and I live in India	["Chirag Jain"]
Myself Aman Srivastava and I Engineer	["Aman Srivastava"]
People call me Yash Doshi and my friend Sagar Dedhia	["Yash Doshi", "Sagar Dedhia"]
Hi, how are you doing?	[]

Note Convert CSV file into sentence_list and entity_list utilizing the following code

```
from models.crf_v2.crf_train import CrfTrain
import ast
import pandas as pd

csv_path = '/app/crf_chat.csv'  #  The path where the csv file is stored.

data = pd.read_csv(csv_path, encoding='utf-8') #  Load the csv file into a pandas
DataFrame
sentence_list = list(data['sentence_list'])
entity_list = list(data['entity_list'].apply(lambda x: ast.literal_eval(x)))
```

2. Preprocess Data

The module is used to take input as the sentence_list and entity_list and converts it to features for training the Crf Model.

- **Tokenization and Label Generation**

  This module is used to tokenize and generate labels according to the IOB standards of Named Entity Recognition (NER)

```
from models.crf_v2.crf_preprocess_data import CrfPreprocessData
sentence_list =['book a flight from Mumbai to Delhi', 'Book a flight to
Pune']
entity_list = [['Mumbai', 'Delhi'], ['Pune']]
docs = CrfPreprocessData.pre_process_text(sentence_list, entity_list)
print(docs)

>>> {'labels': [['O', 'O', 'O', 'O', 'B', 'O', 'B'], ['O', 'O', 'O',
>>> 'O', 'B']],
>>> 'sentence_list': [['book', 'a', 'flight', 'from',
>>> 'Mumbai','to', 'Delhi'], ['Book', 'a', 'flight', 'to', 'Pune']]}
```

- **Parts Of Speech Tagging**

  This module is used to tag the parts of speech for the tokenized sentences

```
from models.crf_v2.crf_preprocess_data import CrfPreprocessData
docs = CrfPreprocessData.pos_tag(docs)
print(docs)

>>> {'labels': [['O', 'O', 'O', 'O', 'B', 'O', 'B'],
>>> ['O', 'O', 'O', 'O', 'B']],
>>> 'pos_tags': [['NN', 'DT', 'NN', 'IN', 'NNP', 'TO', 'VB'],
>>> ['VB', 'DT', 'NN', 'TO', 'VB']],
>>> 'sentence_list': [['book', 'a', 'flight', 'from',
>>> 'Mumbai', 'to', 'Delhi'],['Book', 'a', 'flight', 'to', 'Pune']]}
```

- **Load Word Embeddings**

  This module is used to load the word embeddings into the memory

```
from models.crf_v2.load_word_embeddings import LoadWordEmbeddings
word_embeddings = LoadWordEmbeddings()
vocab = word_embeddings.vocab
word_vectors = word_embeddings.word_vectors
```

- **Assign Word Embeddings**

  This module is used to load the word embeddings from the local disk and then each token is assigned a its coressponsding word embedding

```
from models.crf_v2.crf_preprocess_data import CrfPreprocessData
docs['word_embeddings'] =
[CrfPreprocessData.word_embeddings(processed_pos_tag_data=each,
vocab=vocab, word_vectors=word_vectors)
for each in docs[SENTENCE_LIST]]
```

- **Generate Feature From Data**

  This module is to generate feature for each token which acts as the input features for the CRF Model

  - **Window Size**

    We consider a window of two tokens in the forward and the backward direction for each token at a given time step.

    Example

    a. My name |*is Pratik **Jayarao** and this*| is my friend Hardik Patel.
    b. I want to |**travel from Mumbai to Delhi**|

  - **Token features**

    Each token present in the window is converted to the following features as an input

    a. **lower**

       The token is casted to the lower case.

b. **isupper**

   Flag to check if the complete token is in upper case

c. **istitle**

   Flag to check if the first letter of the token is capitalized

d. **isdigit**

   Flag to check if the token is a digit

e. **pos_tag**

   Part of speech tag for the given token

f. **word_embeddings**

Word Vectors associated with each token

```
from models.crf_v2.crf_preprocess_data import CrfPreprocessData
features = CrfPreprocessData.extract_crf_features(docs)
labels = docs['labels']
```

3. **Train Crf Model**

   This module takes input as the features and the and labels obtained from the preprocessing module and trains a CRF model on it. The module saves this model locally and this path is then returned.

   Model Cofigurations

   - C1 - L1 Regularization constant
   - C2 - L2 Regularization constant
   - Max Iterations - Max number of iterations to be executed

```
from models.crf_v2.crf_train import CrfTrain

crf_train = CrfTrain(entity_name='crf_chat')
model_path = crf_train.train_crf_model(x=features, y=labels, c1=0, c2=0,
max_iterations=1000)

print(model_path)

>>>'/app/models_crf/crf_chat'
```

4. **Training Example**

```
from models.crf_v2.crf_train import CrfTrain
import ast
import pandas as pd

entity_name = 'crf_chat' #  The name of entity which will be reflected in
the model and is the primary identifier of the entity.
```

```
csv_path = '/app/crf_chat.csv'  #  The path where the csv file is stored.

data = pd.read_csv(csv_path, encoding='utf-8') #  Load the csv file into a pandas DataFrame
sentence_list = list(data['sentence_list'])
entity_list = list(data['entity_list'].apply(lambda x: ast.literal_eval(x)))

crf_model = CrfTrain(entity_name=entity_name) #  Initialize the Crf Model
model_path = crf_model.train_crf_model_from_list(sentence_list=sentence_list, entity_list=entity_list)

print(model_path)
>>>'/app/models_crf/crf_chat'
```

D. CRF ENTITY DETECTION (Standalone)

The mdoule can be used to detect entities utilizing the previously trained CRF model. This module takes input as the entity name and the text from which the entity has to be extracted.

1. **Load Model**

   The crf model corresponding to the entity is loaded from the local disk into the memory and the tagger is initialized. This module is a Singelton class and hence the model if once loaded remaines in the memory.

2. **Extract Entity**

   This module is responsible to tag the entities from text and return the detected subtexts.

```
from models.crf_v2.crf_detect_entity import CrfDetection
crf_detection = CrfDetection(entity_name='crf_chat')
detected_text = crf_detection.detect_entity(text='People call me Aman Shah and my friend Krupal Modi')
print(detected_text)
>>> ['Aman Shah', 'Krupal Modi']
```

E. CRF-TEXT ENTITY DETECTION (Combined Module)

This module is used to run the previously trained CRF model alongside the tradional text entity detection (detection accomplished from datastore). This module takes input as the entity name and returns a combined result using both CRF Detection and Text Entity Detection.

1. **CRF Standalone Detection**

   The CRF Standalone Detection is triggered. The result i.e the list of entities detected from the CRF is returned.

2. **Text Entity Detection**

   This module is used to detect entities leveraging the Text entity fucntionality. This module returns the entity_value and the original_text.

3. **Verification Source**

   This module is responsible to assign the source (CRF / Datastore) from which the entity is detected.

4. **Ensemble Results**

   This module is responsible to combine the results detected from the Datastore and the CRF Model. The Datastore value is given higher priority. If the same entity is detected by the CRF model and the Datastore then the entity value is resolved to the entity value returned by the Datastore Module

```
from ner_v1.detectors.textual.text.text_detection_model import TextModelDetector
text_model_detector = TextModelDetector(entity_name='crf_chat',
live_crf_model_path='/app/models_crf/crf_chat')
output = text_model_detector.detect(message='my name is harsh')
print(output)
>>> [{'detection': 'message',
        'entity_value': {'crf_model_verified': True,
    'datastore_verified': False,
        'value': 'harsh'},
        'language': 'en',
        'original_text': 'harsh'}]
```

F. PYTHON API DOCUMENTATION

1. Training

```
import pandas as pd
import json
import requests
import ast

def convert_data_to_json(csv_path, entity_name):
    """
    This method is used to convert the CSV file into the appropriate json which
will be
    consumed by the Chatbot Ner docker.

    Args:
            csv_path (str): The local path where the training data is stored
        entity_name (str): Name associated with the entity (will be added in the
model
        path)
        language_script (str): The script of the training data

    Returns:
        external_api_data (str): json dump of the transformed training data
dictionary
        which will be used to train the crf model on the training data
    """

    data = pd.read_csv(csv_path, encoding='utf-8')

    sentence_list = list(data['sentence_list'])
    entity_list = list(data['entity_list'].apply(lambda x: ast.literal_eval(x)))
    external_api_data = json.dumps({
        'entity_name': entity_name,
        'sentence_list': sentence_list,
        'entity_list': entity_list,
    })
    return external_api_data

def train_crf_model(external_api_data, chatbot_ner_url, http_timeout):
  """
  This method is used to train the crf model on the transformed
(convert_data_to_json)
  training data.

  Args:
        external_api_data (str): json dump of the transformed training data
dictionary
        which will be used to train the crf model on the training data (output of
        convert_data_to_json)
        chatbot_ner_url (str): The base chatbot ner url for the docker.
        http_timeout (int): The time out for the api call to Chatbot Ner to train
the model

  Returns:
        live_crf_model_path (str): The path where the trained crf model is
stored.
  """

  url = chatbot_ner_url + '/entities/train_crf_model'
  params = {'external_api_data': external_api_data}
  response = requests.post(url, data=params, timeout=http_timeout)
  response_body = json.loads(response.text)
  live_crf_model_path = response_body.get('result').get('live_crf_model_path')
  return live_crf_model_path


#  CONSTANTS
CHATBOT_NER_SCHEMA = 'http://'
CHATBOT_NER_HOST = 'localhost'
CHATBOT_NER_PORT = '8081'
CHATBOT_NER_URL = CHATBOT_NER_SCHEMA + CHATBOT_NER_HOST + ':' + CHATBOT_NER_PORT
HTTP_TIMEOUT = 100
CSV_PATH = "/home/ubuntu/crf_chat.csv"
ENTITY_NAME = 'crf_chat'


external_api_data= convert_data_to_json(csv_path=CSV_PATH,
entity_name=ENTITY_NAME)

live_crf_model_path = train_crf_model(external_api_data=external_api_data,
chatbot_ner_url=CHATBOT_NER_URL, http_timeout=HTTP_TIMEOUT)

print(live_crf_model_path)
>>> u'/app/models_crf/crf_chat'
```

Note *Save the* ***live_crf_model_path***

2. Detection This code can be used to detect the entities from the given text

```
from ner_v1.detectors.textual.text.text_detection_model import TextModelDetector
text_model_detector = TextModelDetector(entity_name='crf_chat',
live_crf_model_path=live_crf_model_path)
output = text_model_detector.detect(message='my name is harsh')
print(output)
>>> [{'detection': 'message',
        'entity_value': {'crf_model_verified': True,
    'datastore_verified': False,
        'value': 'harsh'},
        'language': 'en',
        'original_text': 'harsh'}]
```

NIKI.AI https://www.analyticsinsight.net/niki-ai-ultimate-automated-ai-powered-bot/

Services: The chatbot currently provides services such as mobile recharges, utility bill payments, cab booking, bus ticketing and hotel bookings, events and movie ticketing, nearby searches and deals, with flights bosokings and health services in the pipeline. To the businesses, Niki.ai provides a plug and play technology in the form of Niki Chatbot Software Development Kit, that can be easily integrated everywhere including operating systems, on messaging platforms, and on the brand's applications (app and web).