Universitat de les
Illes Balears

escola politècnica superior

Treball Final de Grau

GRAU D’ENGINYERIA INFORMÀTICA

Chatbot API: A service to develop text-based interfaces

LUIS FOMINAYA

Tutor
Isaac Lera

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, June 28, 2017

CONTENTS

Contents i

Acronyms iii

Prologue v

1 Introduction 1
- 1.1 Project requirements . . . 3
  - 1.1.1 Notation . . . 3
  - 1.1.2 Requirement definition . . . 3
  - 1.1.3 Requirement validation . . . 3
- 1.2 Technical requirements and methodology . . . 4
  - 1.2.1 Language and platform . . . 4
  - 1.2.2 Style . . . 5
  - 1.2.3 Tools . . . 6
- 1.3 Tasks . . . 7
- 1.4 Planning . . . 7

2 Overview of the history of chatbots 9

3 Implementation 17

4 Named Entity Recognizer 21
- 4.1 Requirements . . . 22
- 4.2 Validation . . . 23
- 4.3 Design . . . 23
  - 4.3.1 Tokenizer . . . 23
  - 4.3.2 Recognizer . . . 24
- 4.4 Entity types . . . 25
  - 4.4.1 Numbers . . . 25
  - 4.4.2 Times and dates . . . 25
  - 4.4.3 Custom types . . . 26
  - 4.4.4 Free text . . . 27
- 4.5 Data sources . . . 27

5 Chatbot 31
- 5.1 Chatbot API . . . 31
- 5.2 Chatbot middleware . . . 35

5.3 Other chatbot-building APIs 35

6 Use-case: HotelBot **39**
6.1 Example conversation 42

7 Conclusions **43**
7.1 Future improvements 43
7.2 Participation in the Whenhub hackathon 44
7.3 Personal valoration 45

Bibliography **47**

ACRONYMS

AIaaS	Artificial Intelligence as a Service
API	Application programming interface
FaaS	Functions as a Service
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
JSON	Javascript Object Notation
NER	Named Entity Recognizer
REST	Representational State Transfer
UI	User Interface
UIB	Universitat de les Illes Balears
NLP	Natural Language Processing

PROLOGUE

This document describes the design and implementation of an API to process natural language commands and translate them into structured data that can be used to build interfaces based on natural language (either written or spoken). In contrast to other existing similar projects, here the focus is on improving developer experience to make the process of building any application as simple as possible.

An application built using this API won the "Best overall" prize in an online hackathon[1] held during the month of May, 2017 by Whenhub[2], earning a 5000$ cash prize. The application consisted in a web application that allowed users to create schedules and events by speaking or writing text, using Whenhub's API.

---

[1] https://whenhub.devpost.com/

[2] https://www.whenhub.com/

CHAPTER 1

INTRODUCTION

The program implemented in this project is a **chatbot-building API**. A chatbot is "a computer program that conducts a conversation via auditory or textual methods"[1]. This is the common definition for this type of program, except for this particular implementation the main purpose is to extract **user commands** from raw text. User commands are sentences in natural language intended to carry out actions such as seeing a list of available hotel rooms, booking a flight, ordering takeout food and so on.

This chatbot is meant to serve as replacement or support for graphical user interfaces on web applications. Therefore, it is designed to be used as a service over internet, which developers can use in their own applications.

It has also been designed to simplify developer tasks through a simple process: they only need to configure the API by defining the actions their application has in order to use it. These “actions” refer to user commands that are of interest, such as “show me the hotels”, which could trigger the action “show hotels” if the developer defined it to do so.

Once the API is set up, they can connect it to a chat, so that user sentences will be sent to the API, which will respond with structured information about which action the user is trying to carry out, and with which parameters. For example, in a hotel booking application, a user could say “show me hotels in Madrid”, and the API would interpret that sentence as “action = show hotels, location = Madrid”.

In my opinion, natural language interaction has advantages over graphical interaction, both economically and from the point of view of usability and accessibility.

Graphical user interfaces, specially the most usable ones, require large amounts of development time without ensuring usability issues will fully disappear. There is also the problem of making the application accessible, which can vastly increase development time and cost, making many companies discard the option, and which when implemented could lead to several versions of the same interface.

This area is where chatbots excel, as they inherently follow the principles of universal design. A chatbot accurate enough to recognize text and speech equally could be used both by the impaired and the average user in the exact same way.

Another advantage of conversational UIs is that they free developers of designing, building and testing graphical interfaces, which may take up a lot of resources to build and maintain. With a chatbot-based application, a generic chat interface suffices to conduct all the actions, which can be copied from any chat interface template, and easily adapted.

This project is not so ambitious as to claim that it will provide a universally usable chatbot-building application. It will, however, allow developers to create alternative interfaces that offer the mentioned advantages over graphical user interfaces.

To provide some motivation as to why chatbots are useful, it is important to observe that there are many web applications for flight and hotel booking, restaurant ordering, and in general, carrying out CRUD operations for some domain. All these applications have a similar interaction structure:

1. *Create* something (e.g. a food order, a request, a flight booking, . . .).
2. *Read* one or multiple things (e.g. the user's social media posts, pending orders, all songs by a certain artist), sort and filter them.
3. *Update* something (e.g. the user's profile, one of the user's social media posts/status/tweets).
4. *Delete* something (e.g. a hotel/flight/restaurant reservation, a subscription to something, an email).
5. *Ask* a question (e.g. "what can the application do?", "are you a person?").

The goal of the project is to be able to carry out all these actions using an AI instead of a graphical interface. Because there is a large number of domains user sentences can refer to, developing a program that automatically knows all of them is not feasible. Instead, the developer creates the domain by specifying what users will talk about, mostly using the five categories of user interactions mentioned above.

Since the API is configurable, the developer can describe any action a user could be interested in performing. Using the hotel booking example, users may first book a hotel, then regret it and immediately want to undo the action. The developer needs only specify what are the keywords the user must say for the chatbot to trigger the actions it has been given, how the actions affect the application state and what calls to the backend are in order.

The user will be able to give voice or text commands to the chatbot and interpret the response as though it were a human conversation. If a user wants to book a room in a hotel in Madrid, there will be no need to try to understand what the interface does, but simply ask for it: "Can I book a room in Madrid?", and the chatbot would ask for more details, such as the number of nights, a maximum price or the minimum number of stars the hotel should have.

This hotel-booking chatbot would then return a list, which could contain text and potentially also images for the user to see and select. The application could store user preferences and payment information to simplify the process, making interaction as effortless as possible.

Throughout this report, the example of a hotel booking application called "HotelBot" will be used to illustrate the exposed concepts. The present chapter has already used example sentences that belong to the HotelBot application.

To sum up, the goals of this project are:

1. Develop an API that converts text into structured information
2. Provide enough flexibility to be able to develop any application
3. Make this service as easy as possible to use and integrate

1.1 Project requirements

1.1.1 Notation

RX will denote requirement “X” and **VX** will denote validation of requirement “X”.

1.1.2 Requirement definition

The project’s requirements are the following:

R1. The chatbot must be able to detect the action and parameters a user sentence intends to carry out according to what the developer has defined.

R2. The chatbot must be able to detect ambiguities in language that could lead to different parses of a user sentence, solve them whenever possible, and inform of their presence when they are unavoidable.

R3. The chatbot’s response time must be below a certain threshold that makes interaction with the chatbot responsive, using usability research to provide the threshold.

R4. The chatbot must allow developers to create deterministic conditions for the actions in their application to trigger.

R5. The chatbot must allow developers to use their preexisting data schemas to validate the action parameters.

R6. The chatbot API must be RESTful [2] and functional: The input is processed without modifying any local state, so that multiple identical queries can be cached. The system must also be able to work as a Function as a Service, or FaaS [3], to reduce operational costs.

1.1.3 Requirement validation

In order to validate the requirements, the following metrics and thresholds will be used:

V1. To validate this particular requirement, user interaction is required. It is important to notice, however, that the quality of the interaction will be directly proportional to the time spent creating actions to model different possible conversational paths.

V2. This requirement can be satisfied through unit tests that show the chatbot properly captures ambiguities in language such as “2017” referring to a quantity or a date.

V3. The maximum time to output a response must be within the 10 second window, according to a study conducted by Jakob Nielsen [4]. This can be validated by measuring the time it takes for the chatbot to respond. It is not possible to be below the 1-second window due to Internet delay times.

V4. This condition will be validated if every sentence that triggers some action continues to do so in separate executions under different circumstances.

V5. It suffices to verify that the application specification format allows users to define the validation using some data validation standard such as JSON schema[1].

V6. The API will be proven to be functional if it can be accessed via a FaaS provider such as the Google Cloud Platform[2] or Amazon Lambda[3]. In order to be RESTful, it suffices to send multiple, identical requests and verify that the output is the same.

1.2 Technical requirements and methodology

1.2.1 Language and platform

The project to develop is going to be used mainly by web applications. Some of the code written for the API could be possibly moved to these applications to allow developers to further customize their chatbot. Thus, a language that could be executed by the browser would be preferable to others that cannot.

Also, requirement R6 establishes that the system must operate as a single function, not a stateful server. This will bias the choice of programming language and environment to one that can be easily used to create functional systems. The basic requirement is not that the language is fully functional, but that it supports **first-order functions**.

Although there are many natural language processing libraries in languages like Java and Python such as the Stanford CoreNLP[4], and NLTK[5] respectively, neither of these languages are a good fit. Java forces the use of classes, which create overhead to store some state. This makes sense if the application resides in memory a long time, but in a serverless environment, the lifetime of a class would be limited to the invocation, therefore incurring in overhead to define a class whose methods will only be called once.

Furthermore, Java requires a Java Virtual Machine to boot and load to memory each invocation [5], creating an unnecessary delay and consequent increase in expenses.

Three languages that fulfill these requirements are Javascript with Node.js, Golang[6] and Clojure[7]. Of these three, the first allows writing isomorphic javascript modules, which work both in a Node.js server and the browser, and there are more FaaS providers that offer their services in Node instead of Golang and Clojure, such as Google Cloud Functions[8].

Therefore, the implementation will use Javascript as its language and will be deployed to a FaaS provider.

---

[1] http://json-schema.org/

[2] https://cloud.google.com/

[3] https://aws.amazon.com/lambda/

[4] http://stanfordnlp.github.io/CoreNLP/

[5] http://www.nltk.org/

[6] https://golang.org/

[7] http://www.clojure.org/

[8] https://cloud.google.com/functions/

1.2.2 Style

The time it takes to adapt a codebase to new changes is critical in a production system, as new, unforeseen requirements can and will surface during its lifetime. Thus, it is desirable to make maintainable code that can adapt quickly to these changes.

Other important aspect to consider is the time it takes for a person to understand what the code does. The clearer the names of the functions, modules and even variables, the easier it is for a developer contributing to the codebase to read and edit a module.

However, making the system resilient to changes can be difficult, especially if they not only add but also modify existing parts of the system. Then, a new change can cause new bugs completely unrelated to the implemented functionality.

To address these and many other problems, some professional software engineers [6] recommend a series of principles or guidelines to follow. The most relevant for this project are the following:

1. **Descriptive names**. If there is a variable that represents the number of items, it should be called itemCount, instead of abbreviations such as ic, itCount, itCnt, or any other shortened form that forces the developer to figure out what the variable means.

2. **Consistency**. The name of variables, functions and modules should follow similar principles. Also, if there are several ways to declare a function or initialize a module, it is mandatory to choose one and use that for all cases, so long as there are no differences with the expected functionality.

3. **Unit testing**. To ensure that a new change does not break the existing code, it is paramount to develop unit tests for the different functions in the system. Doing so also removes the need to manually debug functions in most cases, especially if there is a lack of shared state.

Of these aspects, it is important to dwell on the last, as it is the one with the most relevance. A strong test suite that encompasses all the system allows developers working on the system to add code without fear of introducing bugs, as the code can be tested and most issues detected during development.

Unit tests also act as documentation for other developers, as they show what output is expected given a certain input. This benefit can be lost if the system relies heavily on modifying some shared state, because writing unit tests for functions depending on global variables becomes increasingly harder and requires complex setup.

Therefore, the approach to writing functions in this particular project is to make them as self-contained as possible; that is, as close as possible to pure functions [7]. By using pure functions and keeping state local to each function, anyone reading a function only needs to know about that function and the other functions it calls. No variable it uses will be unexpectedly modified by another function, making it self-contained and easily testable.

There are, however, some clear problems with this approach. The first one is that certain features require state. For example, it would be useful that the chatbot takes into account previous conversation information, such as whether or not the user has answered the question the chatbot asked, which action is the user currently trying to do, etc. In this case, state is unavoidable. Still, it is relevant how this state is handled. For this particular use-case, all the chatbot state will be handled by modules that encapsulate how the state is stored and modified, and allow the bulk of the application logic to access and modify this state only through an interface. This way, it is easier to see how the state is handled, as the modules abstract and encapsulate the state's implementation.

The second one is the time for writing unit tests, executing them and making sure they are correct is very time-consuming. It can be argued that the time cost is worth the aforementioned advantages. But that does not change the fact that writing unit tests effectively increases the amount of time to produce code when compared to untested code, even if the latter may become riddled with bugs.

1.2.3 Tools

Out of the many tools that can be used to develop Javascript software, the following have been chosen:

1. **Visual Studio Code**[9]: The first and foremost tool to write code is a program to write. This tool in particular was picked because it is developed precisely for Javascript and provides powerful Intellisense (useful information about the code), cutting down time when writing code.
2. **Git**[10]: Git is an open-source distributed version control system with the benefit of easily handling different versions or branches of the codebase simultaneously. Its main advantage is the low amount of disk space that the different versions of a file require.
3. **Gitlab**[11]: In addition to a local code repository, a remote repository where code is stored is also useful for increased redundancy and protection against potential data loss. It was preferred over other popular alternatives such as Github[12] and Bitbucket[13] as it allows developers to create unlimited private projects with unlimited collaborators.
4. **Git-bash**[14]: This tool provides Windows users with the ability to execute both git commands and Unix commands, used during development.
5. **Google Chrome**[15]: In order to develop a UI, or use one to manually interact with the system, a program capable of rendering a web is necessary. It was chosen for being the most used browser [8] and providing excellent developer tools.

As the entire project is written in vanilla Javascript, no transpilers or compilers are required other than the ones mentioned before. The development cycle consists in

---

[9] https://code.visualstudio.com/

[10] https://git-scm.com/

[11] https://gitlab.com/

[12] https://github.com/

[13] https://bitbucket.org/

[14] https://git-for-windows.github.io/

[15] https://www.google.com/chrome/

writing the functions and their unit tests, then running the tests and making any appropriate changes to fix the tests that do not pass.

1.3 Tasks

The main tasks in general terms for this project are the following:

1. Produce a chatbot API that lets developers create chatbot applications using it.
2. Create a working chatbot based on this API that serves as a real application.
3. Provide documentation of the API for developers.

1.4 Planning

Due to the lack of any real time constraint other than finishing the project within the second semester of the 2016-2017 academic course, there has been no attempt to plan the development of the different modules, nor will there be. Development will take the time it needs, so long as the final product must be tested and polished.

CHAPTER 2

OVERVIEW OF THE HISTORY OF CHATBOTS

The history of chatbots starts shortly after that of artificial intelligence, which is considered to have originated as a field in 1956, during a workshop at the Dartmouth College [9]. The first chatbot was created in 1966 by Joseph Weizenbaum, and it was called ELIZA. It was a program to mock a rogerian psychiatrist [10]. The program identified keywords in user text to then generate simple responses.

Eight years later the program PARRY was developed by Kenneth Colby, a psychiatrist, to simulate a paranoid individual. It passed a variation of the Turing test where 33 judges voted 52% of the time that it was an actual human [11]. In 1972, PARRY and ELIZA were connected via ARPANET and carried out the first conversation without human interaction recorded in history.

Later chatbots began to be accessible through internet, such as Jabberwacky[1] developed in 1988, Albert One [2] deployed in 1995, A.L.I.C.E [3] by Richard Wallace also created the year 1995, and Mitsuku [4] in 2005. A.L.I.C.E and Mitsuku were both written in AIML [5], an XML dialect also developed by Wallace with open-source support to create chatbots, and which is still in use today.

In 2000, SmarterChild came about. It was able to respond to queries such as "what was the Yankee's score last night" [12], and could be reached through popular messaging platforms at the time as AIM [6].

Nevertheless, it was not until the year 2010 that chatbots began to be endorsed by large companies, when Apple developed Siri [7], an intelligent personal assistant that interpreted natural language queries, performed online search and even gave users voice control over the phone's applications (dialing, texting, alarm clock, calendar, and

---

[1] http://www.jabberwacky.com/

[2] https://en.wikipedia.org/wiki/Albert_One

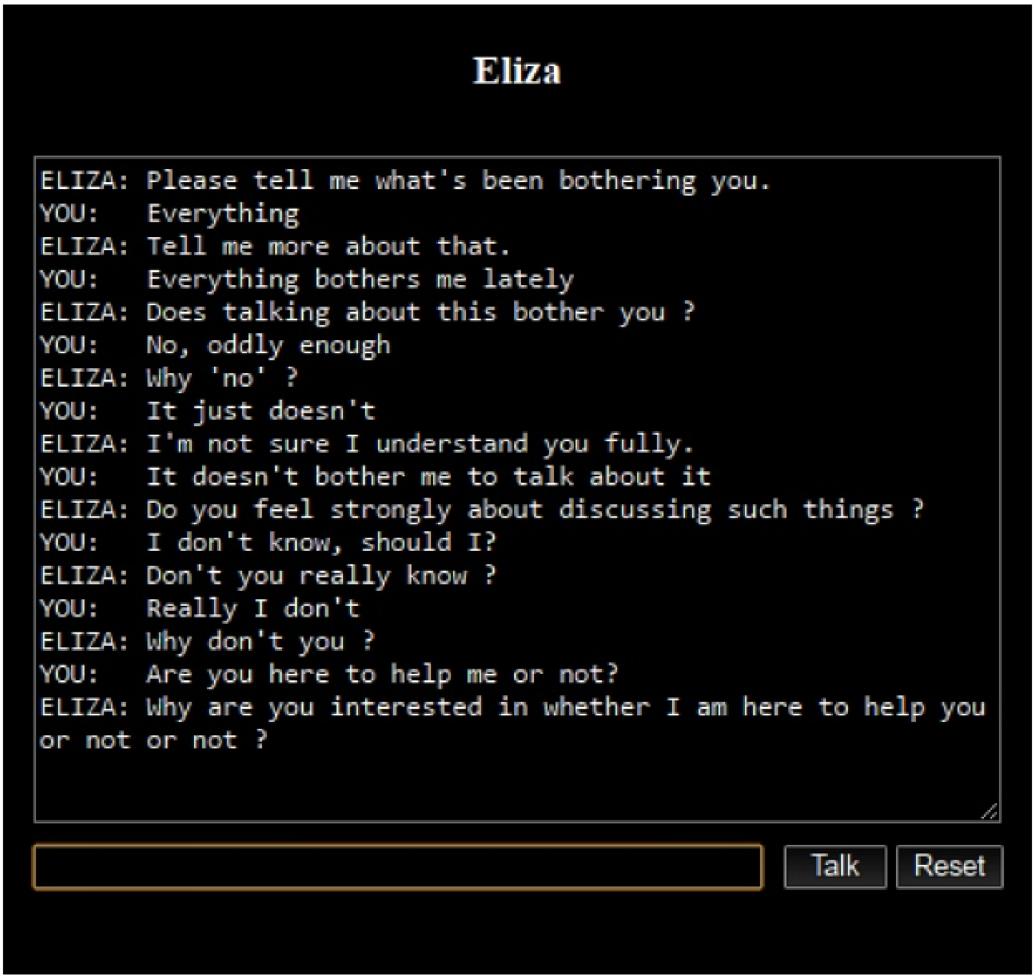
[3] http://www.alicebot.org/

[4] http://www.mitsuku.com/

[5] http://www.alicebot.org/TR/2001/WD-aiml/

[6] https://www.aol.com/aim

[7] https://www.apple.com/ios/siri/



```
ELIZA: Please tell me what's been bothering you.
YOU:   Everything
ELIZA: Tell me more about that.
YOU:   Everything bothers me lately
ELIZA: Does talking about this bother you ?
YOU:   No, oddly enough
ELIZA: Why 'no' ?
YOU:   It just doesn't
ELIZA: I'm not sure I understand you fully.
YOU:   It doesn't bother me to talk about it
ELIZA: Do you feel strongly about discussing such things ?
YOU:   I don't know, should I?
ELIZA: Don't you really know ?
YOU:   Really I don't
ELIZA: Why don't you ?
YOU:   Are you here to help me or not?
ELIZA: Why are you interested in whether I am here to help you
or not or not ?
```

Figure 2.1: An example conversation with ELIZA.

others). It could be accessed purely through voice, and allowed iOS users to carry out actions without having to type or interact with a GUI.

Siri was followed by other natural language understanding software from industry titans such as Google Now [8] in 2012, Amazon's Alexa [9] in 2015, and Microsoft's Cortana in 2015.

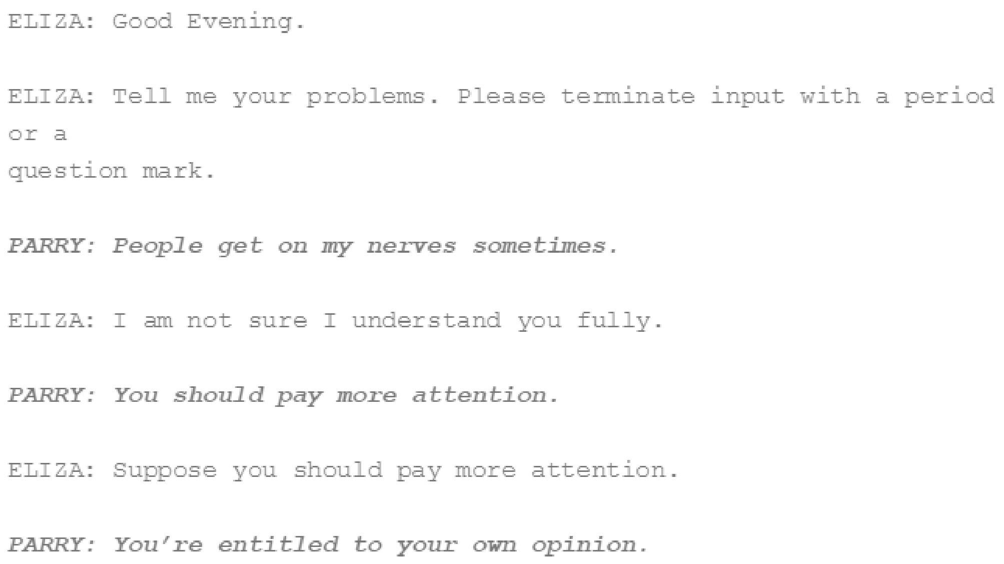
However, these are not projects to create artificial intelligence as earlier chatbots. The focus, instead, is on carrying out useful actions, and they are extensible platforms where developers create their own chatbots, which can then be accessed by users of these platforms (iOS, Windows, Google, Amazon's Echo, and other IoT devices that are now being connected to them). In 2016, Facebook launched a platform for their Messenger application, currently at 1.2 billion monthly users [10], where developers can create chatbots that users interact with to order food, buy cinema tickets, and many

---

[8] https://www.google.com/search/about/

[9] https://developer.amazon.com/alexa

[10] https://techcrunch.com/2017/04/12/messenger/



ELIZA: Good Evening.

ELIZA: Tell me your problems. Please terminate input with a period or a
question mark.

PARRY: People get on my nerves sometimes.

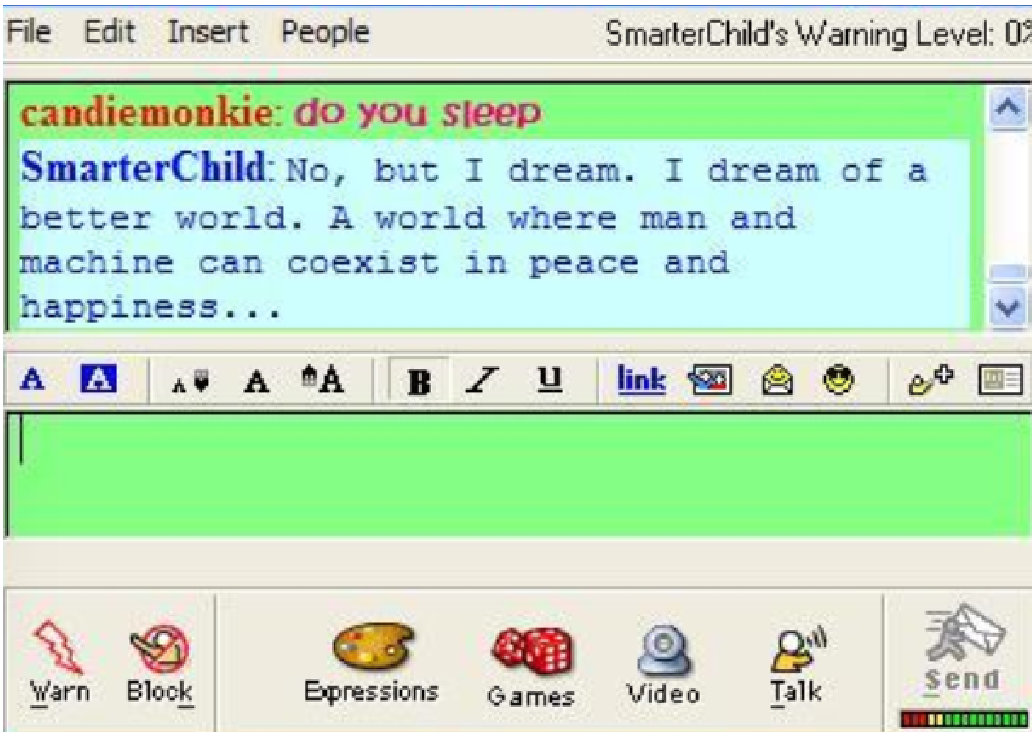
ELIZA: I am not sure I understand you fully.

PARRY: You should pay more attention.

ELIZA: Suppose you should pay more attention.

PARRY: You're entitled to your own opinion.

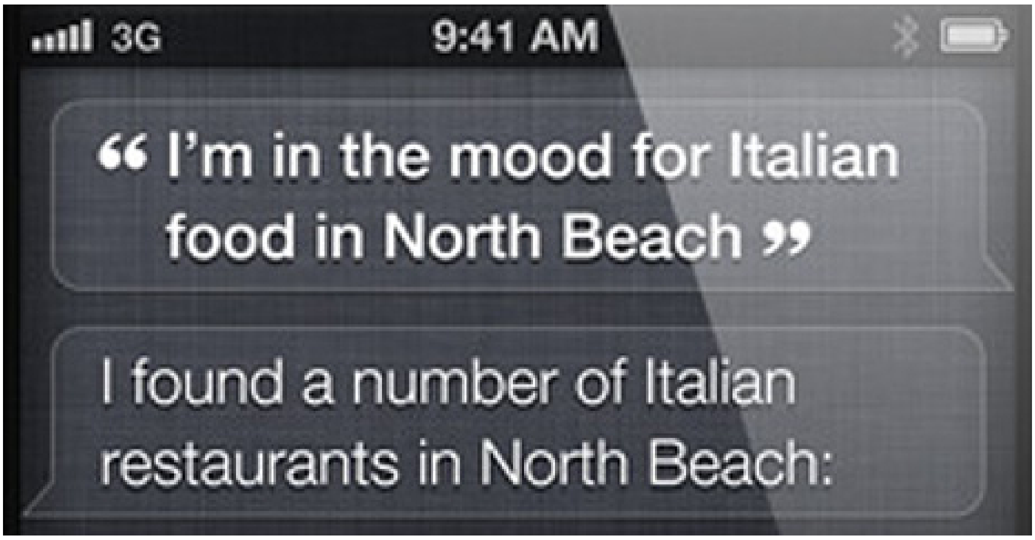
Figure 2.2: An excerpt of the conversation between PARRY and ELIZA.



candiemonkie: do you sleep

SmarterChild: No, but I dream. I dream of a better world. A world where man and machine can coexist in peace and happiness...

Figure 2.3: An example conversation with SmarterChild.



“ I’m in the mood for Italian food in North Beach ”

I found a number of Italian restaurants in North Beach:

Figure 2.4: An example conversation with Siri.

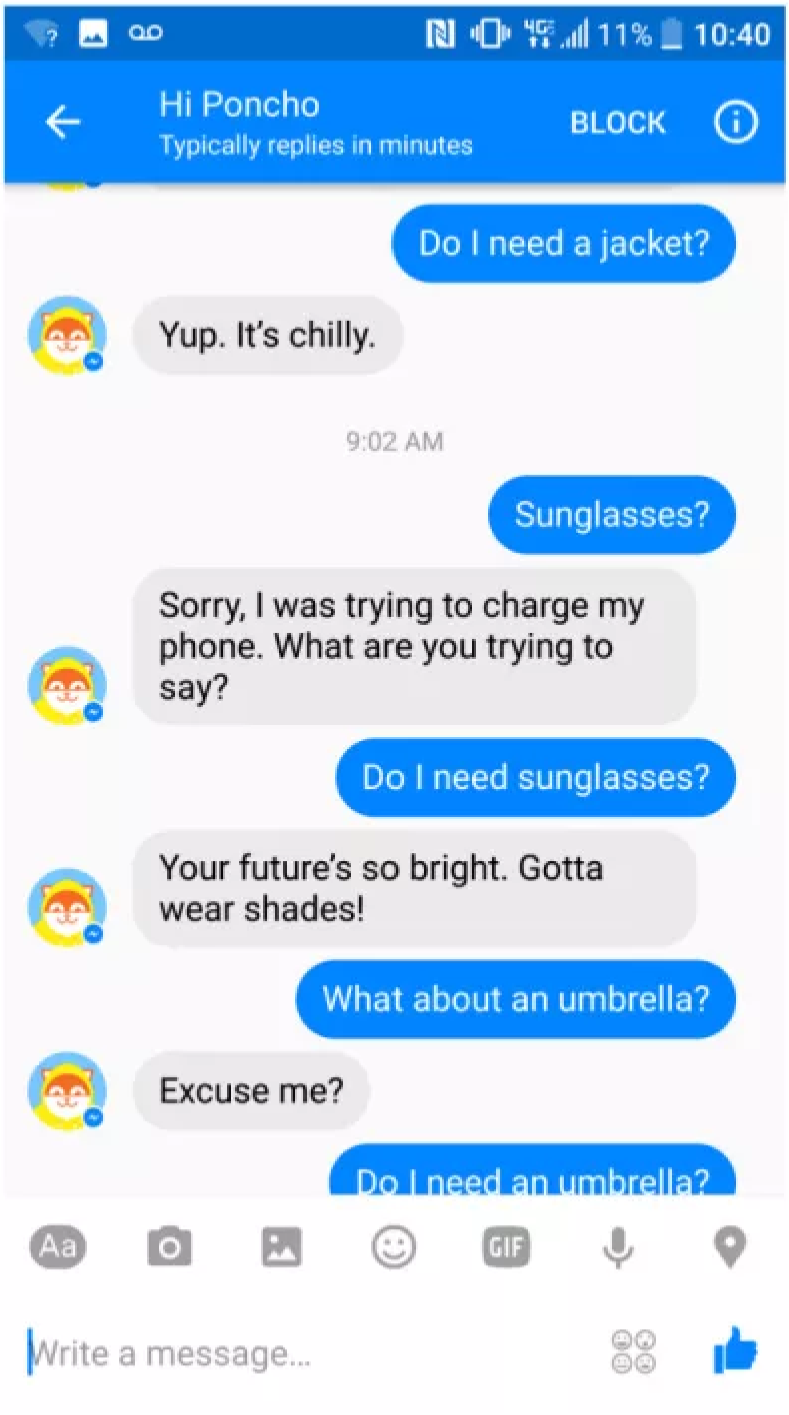
other actions. Other platforms where the chat is a key feature, such as Telegram [11] and Slack [12], also provide developer tools to create chatbots that reside in their platforms.

Apart from these popular platforms, several APIs provide developers with the tools to create chatbots using abstractions over how natural language is processing, and are mentioned below, in the Chatbot chapter, under the "Other chatbot-building APIs" section, where they are compared to this project.

---

[11] https://telegram.org/

[12] https://slack.com/apps



Hi Poncho

Typically replies in minutes

Do I need a jacket?

Yup. It's chilly.

9:02 AM

Sunglasses?

Sorry, I was trying to charge my phone. What are you trying to say?

Do I need sunglasses?

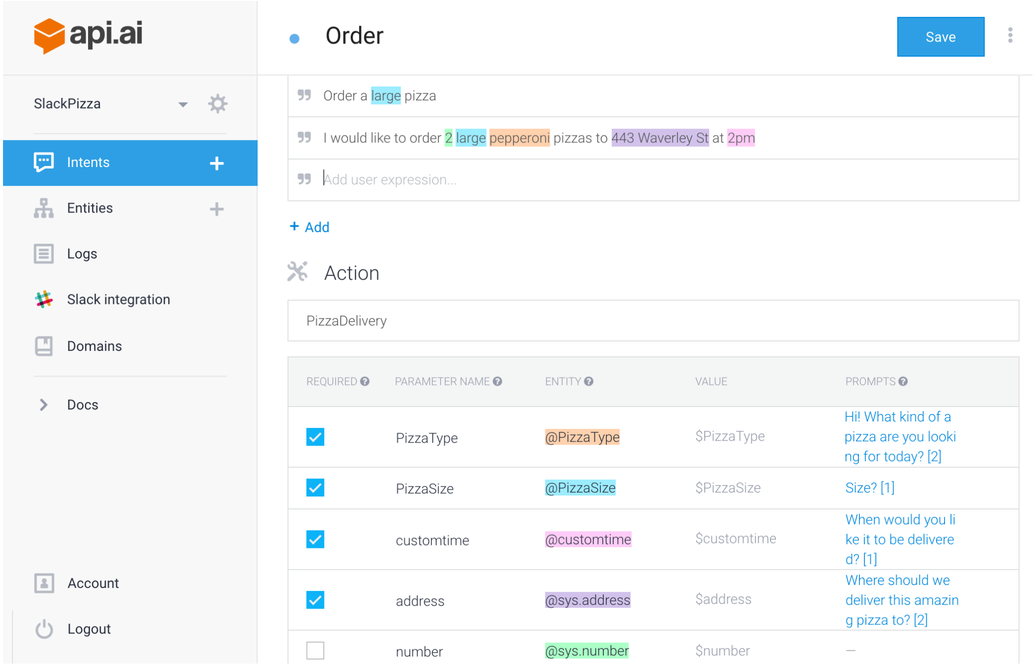
Your future's so bright. Gotta wear shades!

What about an umbrella?

Excuse me?

Do I need an umbrella?

Figure 2.5: An example conversation with a Facebook Messenger chatbot.



Order a large pizza

I would like to order 2 large pepperoni pizzas to 443 Waverley St at 2pm

Add user expression...

PizzaDelivery

REQUIRED	PARAMETER NAME	ENTITY	VALUE	PROMPTS
☑	PizzaType	@PizzaType	$PizzaType	Hi! What kind of a pizza are you looking for today? [2]
☑	PizzaSize	@PizzaSize	$PizzaSize	Size? [1]
☑	customtime	@customtime	$customtime	When would you like it to be delivered? [1]
☑	address	@sys.address	$address	Where should we deliver this amazing pizza to? [2]
☐	number	@sys.number	$number	—

Figure 2.6: API.ai's dashboard when creating a chatbot.

CHAPTER 3

IMPLEMENTATION

The chatbot's requirements make two components necessary for its implementation:

1. A **Named Entity Recognizer**, or NER [13]. This subsystem will be in charge of extracting entities from unstructured text. For example, in the sentence "I want one", the NER would identify the word one as the integer 1.

2. The **chatbot** logic. In order to carry out a natural conversation, this system will map entities to actions and their parameters, as well as identify conversational sentences without entities such as "Hello!", or "how are you?".

It is relevant to remark that the NER is necessary for the application to work properly, but its uses surpass those of the application. It is useful for different information extraction and information retrieval tasks, such as sentiment analysis [14] and question answering [15].

Due to this reason, and its interest as a standalone API, it has been designed to serve as an independent service to the rest of the application, which could also be used for other, more general-purpose applications where developers send sentences and the API returns structured information with the entities it identified within them, useful for digital libraries, email information extraction or profile information extraction[16].

Additionally, the application will require a static data store such as Amazon S3[1] for the configuration of each chatbot. The next figure shows the high-level organization of the software components.

The idea is that the system follows the microservices architecture [3], separating the concerns of the different system modules into self-contained services that interact with each other using RESTful APIs. This design principle allows for improved modularization and reusability.

In this sense, the NER could be improved without having to worry about collateral effects on the rest of the application, so long as the output of the service maintained

---

[1] https://aws.amazon.com/s3/

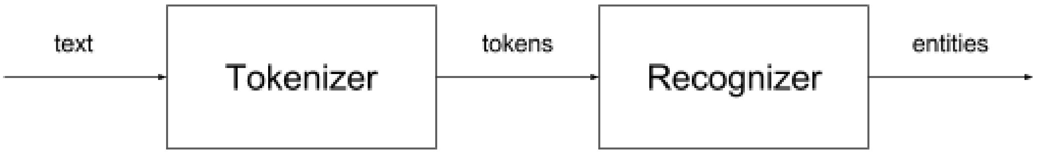


Figure 3.1: Three main components of the architecture: data store, NER, and Chabot logic.

its expected structure and format. In order for the chatbot logic to be a RESTful API, however, it must store no state.

That means the API alone would not be able to handle the conversation flow, as it must not keep a state to be RESTful. This functionality has been separated from the chatbot API, which itself simply maps user sentences to parameters according to the application specification.

All state handling will instead be a responsibility of either the client or a middleware server. In either case, the code will be almost exactly the same, in charge of:

1. **Handling chatbot state**: Keeping track of which action the user is trying to carry out, which parameters have been fulfilled so far and with which values, and other contextual information useful to generate appropriate responses.

2. **Handling user conversation**: Either responding to smalltalk such as "what is your name?" or asking for the missing parameters of an action. In both cases, the developer can customize the responses.

This middleware will be written in Javascript, and therefore will be usable either by the browser or in a middleware server. This setup covers different possible use cases:

1. For applications with very little compute power, such as **IoT** devices, the whole application can be in the cloud, and the IoT device simply takes the input, makes a call to the middleware endpoint which does all the work.

2. For applications built with technologies that cannot interoperate with Javascript, the middleware server can contain the chatbot logic, and accept text as input and return text or JSON as output, to be processable by any programming language.

3. Web applications can take full advantage of the RESTful API and manage the chatbot's state with an open-source library that will be provided. This way, developers can fully customize their chatbot having much of the boilerplating work done.

In fact, developers can use the services provided at different levels, depending on how much customization they need:

1. The most basic level is using exclusively the RESTful NER API to recognize entities in text. This allows for the most flexibility, but also requires the most work.

2. Using the chatbot API provides mapping from entities to parameters, basic disambiguation and conversational action detection.

3. With the state-handling library, used either as middleware or as a library inside the application, even more work and time is saved, at the expense of some flexibility.
4. Finally, using a template application which only requires specifying the application actions and connecting to the developer's API endpoints provides almost all the work needed for an actual application at a fraction of the cost.

Because all the code except the NER and the core chatbot logic will be open-source, developers will have flexibility at all levels to adapt the libraries to their specific needs. The advantage is that a person who has never programmed will be able to create a fully functional chatbot following simple steps to define the actions it can perform and what responses it should give to which questions, effectively creating an accessible application in hours without writing a single line of code.

CHAPTER 4

NAMED ENTITY RECOGNIZER

Some current and potential applications depend on information extraction processes. Its most relevant use is in question answering, a technique used by search engines such as Google and Bing to answer user queries such as “who invented the internet” by extracting information from documents. In this case, for example, Google automatically detects and extracts the names Robert E. Kahn and Vint Cerf as the answer.

Entity recognition is also useful for tasks such as recognizing the names of brands or companies in unstructured text along with sentiment analysis to know if a particular review or comment is hostile or favorable to them[17].

The named entity recognizer is in charge of identifying words that refer to relevant concepts or things. **Entity** refers to person names, dates, numbers, monetary values, locations, any custom entity type related to the application domain, as well as other useful concepts such as languages or telephone numbers [18]. The rest of the system depends on its output, so it is crucial to provide high levels of precision and accuracy to ensure that the rest of the application can depend on it.

Following the principles of the microservices architecture, the NER will have all the relevant information it needs contained within its own data store. Each of the NER’s client applications will have a set of developer-defined entities associated, so that a take-out ordering application which uses the NER could use the entity “beverage” with “coke, water, soda” as its possible values.

Therefore the NER system must be able to detect entities, the values of which may take different forms. Going back to the example, instead of “coke”, it could be interesting to find “Coca Cola, Coca-Cola, cocacola”, and any other variation, and identify it as a variant of the value “coke” for the entity “beverage”. Specifying this list of synonyms will be a responsibility of the developer building a chatbot application, as it needs information specific to the application that the chatbot cannot know.

The list of predefined entities that the NER will recognize is the following:

1. Dates and times
2. E-mails

3. Telephone numbers
4. Numbers
5. Prices and currencies
6. Languages
7. Locations
8. Custom entities

In order to identify them, hand-made rules based on regular expressions will be coded. The reasons for choosing this approach are the following:

1. There is no available labeled data that could be used to train a supervised machine learning system [19].
2. Hand-crafted rules are easily maintainable, extensible and comprehensible [20].
3. As the system grows and acquires more labeled training samples, some data mining technique will perform increasingly better against plain rules alone. However, these rules will still be useful to find **features** that serve as the input for the future machine learning algorithms.
4. Regular expressions, as opposed to machine learning, are deterministic in that the developer can be assured that a certain user sentence will be recognized a certain way. Using only machine-learning techniques that change throughout time makes the system's output change with the same input as the machine learns.

4.1 Requirements

- **R1**. The NER must be able to reliably recognize entities from unstructured text.
- **R1.1**. The NER must be able to recognize system entities such as times, dates, costs, and others that are found to be of general purpose.
- **R1.2**. The NER must be able to recognize user-defined entities and their synonyms or possible values specified by the user.
- **R2**. The entities identified must be normalized to a value usable by a computer, when relevant. This applies, for example, to dates and times, which must be transformed to another format representing the date using some standard.
- **R3**. Whenever possible, the values returned must comply with international standards, such as those created by the IEEE[1] and the ISO[2].

---

[1] https://www.ieee.org/index.html

[2] https://www.iso.org/home.html

4.2 Validation

R1, R2, R3. A comprehensive documentation of the recognized entities and their value formats, indicating the different formats that are accepted and which standards they follow.

4.3 Design

The NER will be a system that takes as input a sentence and produces as output a data structure that indicates what entities there are and which positions they occupy. To indicate the boundaries of each entity, the position of their starting character and the string that matched the entity will be used, in a similar fashion to what the SUTime [21] library does, which is used for time expression recognition and normalization.

The system will try to identify each type of entity in the provided sentence, iterating over all the identified tokens, first finding features for each of them and then predicting an entity type. This way, the algorithm resembles other sequence labeling techniques, and the extracted information from regular expressions can be used to feed a more complex classifier such as a CRF [22] or a neural network [23] changing the minimum amount of code: a single, self-contained module.

For each entity, the algorithm will proceed by first identifying the entity type and then normalizing its value. This way, a word such as “yesterday” should be first identified as a “date” entity, and then normalized to its corresponding value according to the ISO 8601[3], for example, “2017-03-04T17:23:43.307Z”.

In order to be as error-tolerant as possible, the entity identification process will be case-insensitive. On a related note, it is important to mention spell checking and potential error correction. The original implementation of the system will use no spell checkers or correctors due to several reasons:

1. Mobile phones and browsers already incorporate spell checking.
2. Implementing a spell checking system means two things: a time cost to develop the program, and increased storage requirements and execution time for each user sentence, which will require spell correction.
3. The spell checker will necessarily make errors, and even though a corrector with above 90% accuracy is currently easy to implement[24], it will still sometimes cause the correction to change the meaning of the user sentence, which will frustrate the user.

The high-level architecture is shown in the figure below. There are two main subsystems in the NER: the tokenizer and the recognizer.

4.3.1 Tokenizer

This module takes as input raw text and finds the substrings that match the regular expressions for time and date expressions, numbers, as well as non-entity types such as

---

[3] https://www.iso.org/iso-8601-date-and-time-format.html

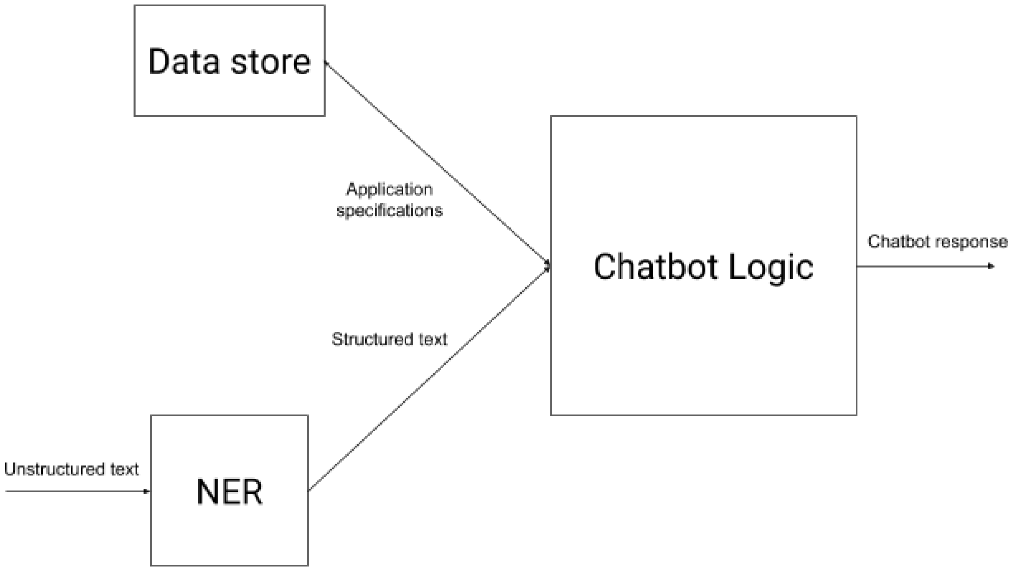


Figure 4.1: High-level architecture of the NER.

sentence separators (“.”, “,”, etc.), and assigns them their corresponding features, which are useful information to classify tokens as entities.

There can be features that are very strongly correlated to a single entity type, e.g. a token with the pattern 01-05-2017 is very likely a date, as well as features that are more general, such as capitalization, which could be useful to finding names, but is not a sufficient condition, as seen by the example "Can it be today?", where the word "can" is only capitalized because it is at the start of the sentence.

The features are assigned by the feature identifier module, which contains the functions that given text and its already-found tokens finds features and assigns them to existing tokens or creates new tokens if they have not yet been found. Once this module finishes, featureless tokens are created with the words in the sentence that have not been assigned a feature. These tokens will be used by the merger module later.

Once all features have been found, the merger joins together tokens that form part of the same entity. For time periods such as “between today and tomorrow”, the merger takes the tokens “today” and “tomorrow”, and joins them in a composite token that indicates the semantic structure: today is the starting date and tomorrow, the ending, due to the words that precede each of them.

The whole process is explained with an example:

1. The sentence “between today and tomorrow” is passed to the tokenizer.
2. Four tokens are identified: “between”, with potentially several features, among them the starting date keyword feature, “today”, with the relative date feature, “and”, with the ending date keyword feature among others, and “tomorrow”, again with the relative date feature.
3. The merger finds two successive dates preceded by starting and ending keywords, and joins them together forming a composite token where “today” is the start and “tomorrow”, the end.

4.3.2 Recognizer

The recognizer takes the tokens extracted in the tokenizer as its input and is in charge of classifying and normalizing the value of the entities. Classification of tokens is performed on the basis of the features they have. The classifier is therefore the aforementioned module that could be modified to introduce machine learning techniques.

Currently, it is mostly a straightforward mapping function from features to entity types: if, for example, a token has the relative date feature (tomorrow, yesterday), its entity type is set to date, regardless of other conditions (tomorrow could refer to the name of a book or a music band).

Although this approach works for many entity types, it does not for names of people, brands and other more complex types that cannot be fit into a simple regular expression. It is for these techniques that machine learning provides better results.

The recognizer uses the entity type found by the classifier to call the normalizer. This module contains a large portion of the NER's logic, as it is in charge of converting arbitrarily complex tokens such as "between today and tomorrow" into values processable by a computer, in this case a JSON object with two fields, start and end, the first with the ISO 8601 string value corresponding to "today" and the second with the value for "tomorrow", relative to the current date.

There are certain entity types (location, email) that do not need to be normalized, and instead their string value suffices. The normalizer is also in charge of detecting ambiguities: parts of the sentence that could refer to several potential entities. For example, the string "2017" could refer either to the year or the number. It is a responsibility of higher levels to determine how to disambiguate depending on other factors such as conversation context.

4.4 Entity types

The table in the following figure shows the currently identified entity types, as well as the format of their output.

It is important to notice that even though continents and countries have standard ISO codes, they are returned as a string because that way a developer using the API can process all location entities in the same way.

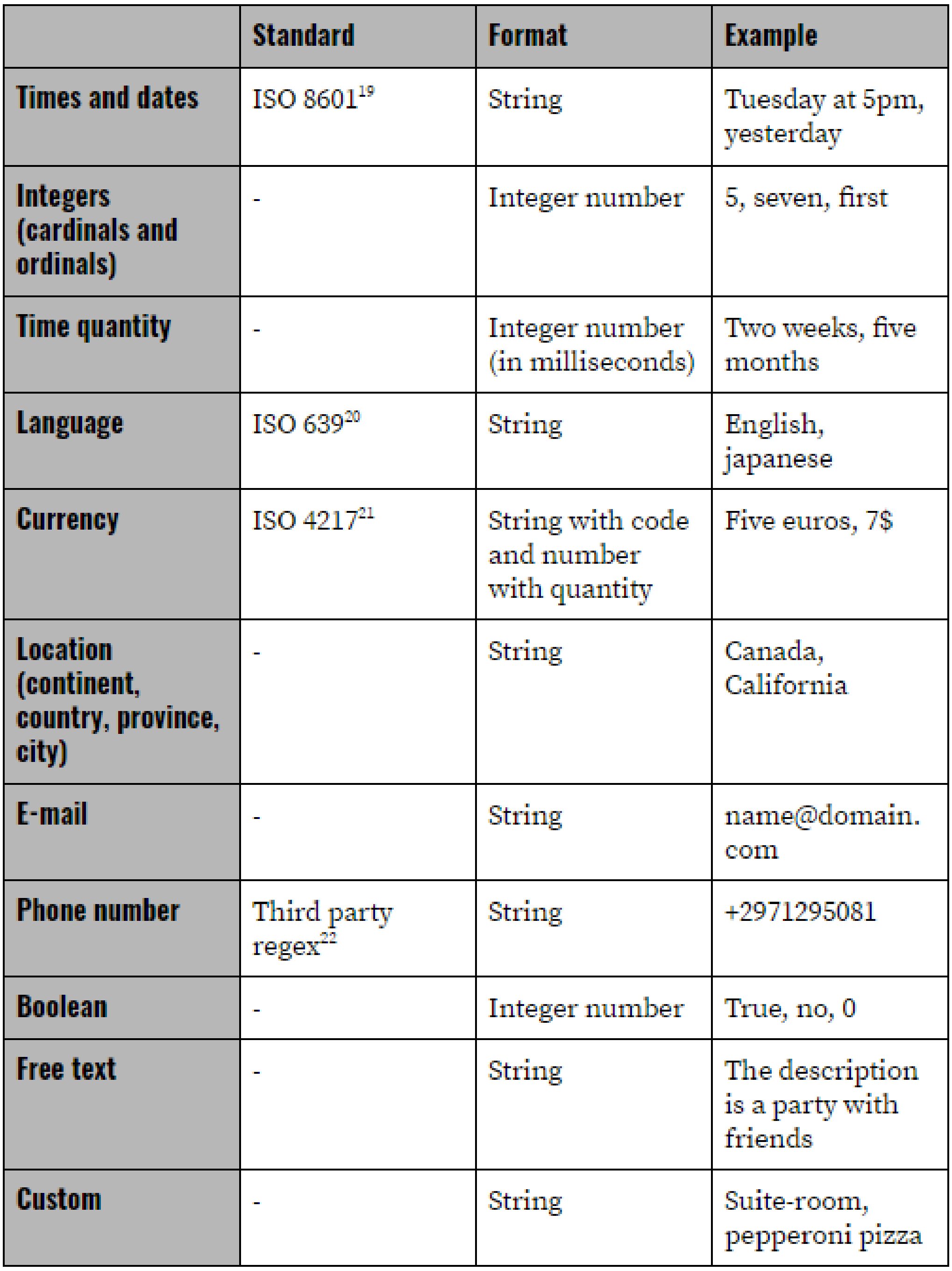
Also, the boolean type is very useful with parameters such as "has views to the ocean" in a hotel booking application, or "includes beverage" in a food takeout application.

Next, additional information is provided about some of the entity types.

4.4.1 Numbers

Currently, two types of numbers are recognized: cardinal and ordinal integers. More particularly, the following numerical expressions are recognized:

1. Any **digit** or combination of digits, e.g. 0, 1020, 12345.
2. **Cardinals** written as words up to quintillions, e.g. four, five hundred, six thousand, four hundred ninety nine, and so on.
3. **Ordinals** written as digits ending with "st", "nd", "rd", "th", e.g. 1st, 45th, etc



	Standard	Format	Example
Times and dates	ISO 8601[19]	String	Tuesday at 5pm, yesterday
Integers (cardinals and ordinals)	-	Integer number	5, seven, first
Time quantity	-	Integer number (in milliseconds)	Two weeks, five months
Language	ISO 639[20]	String	English, japanese
Currency	ISO 4217[21]	String with code and number with quantity	Five euros, 7$
Location (continent, country, province, city)	-	String	Canada, California
E-mail	-	String	name@domain.com
Phone number	Third party regex[22]	String	+2971295081
Boolean	-	Integer number	True, no, 0
Free text	-	String	The description is a party with friends
Custom	-	String	Suite-room, pepperoni pizza

Figure 4.2: Entity types, their format and output.

4. **Ordinals** written as words up to quintillions, optionally followed by the word "one", e.g. first one, twenty second, nine thousand ninetieth.

4.4.2 Times and dates

The list of recognized date expressions is the following:

1. **Week days** and their abbreviations, e.g. tuesday, thurs.
2. **Month names** and their abbreviations, e.g. march, aug.
3. **Years** expressed as digits or their names, e.g. 2017, two thousand seventeen.
4. **Time adverbs**, e.g. today, tonight, yesterday.
5. **Date idioms**, e.g. the day after tomorrow.
6. Dates **relative to time quantifiers**, e.g. in two weeks, seven months from now.
7. **Absolute dates**[25] with and without the preceding week day name. The following formats are identified and normalized:
8. MM_DD_YYYY
9. DD_MM_YYYY
10. YYYY_MM_DD
11. YYYY_DD_MM
12. MM_YYYY
13. YYYY_MM
14. **Combinations** of the previous dates to create date periods, e.g. between today and tomorrow.

As it can be deduced, absolute dates with formats 7.a and 7.b as well as formats 7.c and 7.d can be ambiguous, as dates such as 1/2/2017 can refer to either the first of February or the second of January. These entities are detected as ambiguities, and it is up to the caller to decide how to interpret the date.

There are other possible date references that have not been implemented, such as known holidays (e.g. Christmas, or Hanukkah) and relative references to months and years (e.g. last year, or next month). These are left as future improvements.

The following are the recognized time expressions:

1. **24-hour clock** formatted digits with and without seconds, e.g. 18:15, 10:25:00.
2. **12-hour clock** with either a.m or p.m, e.g. 12 pm, six a.m.
3. 12-hour **clock expressions**, e.g. quarter past five, seven to ten, nine o'clock.
4. **Parts of the day**, e.g. noon, morning, evening.

Furthermore, any combinations of the accepted times and dates are properly mapped to either dates or periods, depending on how they are formulated. For example, “tuesday evening” will be mapped to the next tuesday from the current date at 6 pm. On the other hand, “between now and this evening” will become a date period, even though its members are times.

4.4.3 Custom types

Custom types are defined entirely by the user using one of two ways: lists and regular expressions. Either one can be specified using JSON schemas, so that developers can directly import their current custom types. An example JSON specification would be the following:

```
{
   "name": "tour",
   "systemType": "enum",
   "validationSchemaFile": "url_of_the_schema_file.json"
}
```

The schema file would be loaded once, and the resulting entity specification would then be:

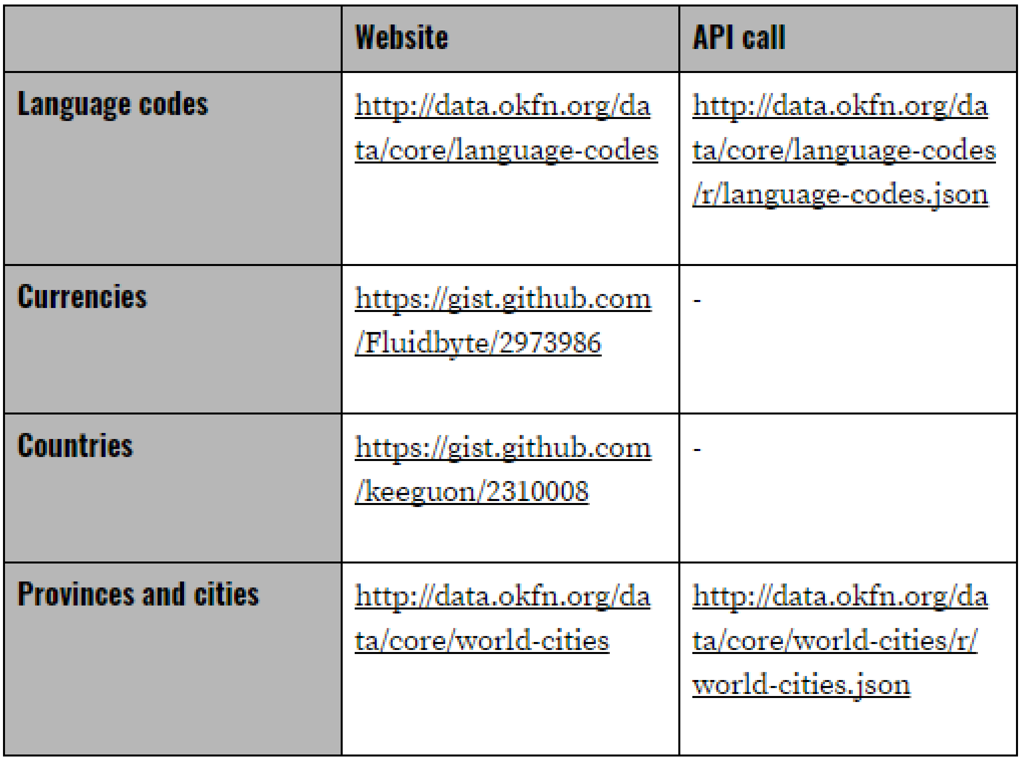
```
{
   "name": "tour",
   "systemType": "enum",
   "validationSchemaFile": {
     "enum": [
       "tour",
       "visit"
     ]
   }
 }
```

4.4.4 Free text

The free text parameter is used to capture free form text, such as the text used to describe an event. In order for it to work the developer must specify triggers such as “description”, so that from the text “the description is a party with friends”, the extracted free text is “is a party with friends”, which can be easily processed to remove the leading verb "is". This work will for now be carried out by the API caller.

An example of an entity with the free text type is in the following specification:

```
{
   "name": "description",
   "systemType": "freeText",
   "triggers": [
     "description"
   ]
 }
```



	Website	**API call**
Language codes	http://data.okfn.org/data/core/language-codes	http://data.okfn.org/data/core/language-codes/r/language-codes.json
Currencies	https://gist.github.com/Fluidbyte/2973986	-
Countries	https://gist.github.com/keeguon/2310008	-
Provinces and cities	http://data.okfn.org/data/core/world-cities	http://data.okfn.org/data/core/world-cities/r/world-cities.json

Figure 4.3: Entity types, their format and output.

Here the entity has been defined using a schema file, but it could also have been specified directly.

4.5 Data sources

The NER needs lists to map entities such as countries, capitals or currencies to their corresponding standard value, and also lists of known names, such as province and city names. For some of these entities, a static list may suffice, such as area unit names, which are a fixed standard (e.g. metres, pounds, litres, etc.). However, for other entities such as the most populated provinces and cities, which are more volatile and prone to change, an API that is periodically called to retrieve such information is better suited.

The next table presents a list of such sources of information:

CHAPTER 5

CHATBOT

The chatbot is in charge of handling user conversations with the purpose of performing actions in the context of a certain application. In contrast to the NER, this system has different separated components to carry out its function. These components can be used from either server or client, thanks to Javascript's isomorphism [26].

The two main modules have been outlined before, and now will be explored in more detail:

1. The **RESTful** chatbot **API**: A program in charge of finding entities in user sentences and mapping them to the parameters defined in the application specification.
2. The **state-handling** library: A bundle of functions, mainly, keeping the context of the user conversation, asking the user for the missing parameters necessary to complete an action, and generating responses to user sentences taking into account the conversational context.

5.1 Chatbot API

This service will provide developers with the ability to find the parameters and actions they define inside unstructured text. It will require an application specification, that is, a configuration file with all the actions the application has and the parameters each accepts.

The API will be RESTful because that way requests and their responses are cacheable both client and server-side, thus reducing the overall cost and execution time. The idea is that when a request arrives at the API, it is determined whether the sentence is a **conversational** action, e.g. "What is your name?", which have fixed, predefined answers, or an **application** action.

If it is a conversational action, then it is not necessary to call the NER and the conversational action name is returned. Instead, if it is an application action, then entity extraction and mapping to parameters is necessary, which are then returned along with the name of the action the user is trying to execute.

So, for example, if the user says "I want to see hotels in Madrid", the chatbot would not recognize the sentence as a conversational action and would thus try to map it to an application action with its parameters. In this case, it could be mapped to the action "show hotels" with parameter "location = Madrid".

But the chatbot cannot possibly have the contexts for every single domain an application may belong to. Therefore, as mentioned before, a specification of the application's actions is necessary for it to work. In this simple case, the specification could be as follows:

```
{
  "name": "Hotelbot",
  "actions": [
    {
      "name": "show hotels",
      "parameters": [
        {
          "name": "location",
          "systemType": "location"
        }
      ],
      "conditions": [
        {
          "systemType": "enum",
          "validationSchemaFile": "somePath/show.json"
        },
        {
          "systemType": "enum",
          "validationSchemaFile": "somePath/hotel.json"
        }
      ]
    }
  ]
}
```

Now, show.json and hotel.json would contain a simple enum following the JSON schema "enum" type containing the alternatives, which could be the following:

show.json:

```
{
 "enum": ["show", "see", "list", "display", "view"]
}
```

hotel.json:

```
{
 "enum": ["hotel", "hostel", "resort"]
}
```

If the user introduced a sentence including both one of the verbs in the show.json file and one of the hotel synonyms in the hotel.json file, then the action “show hotels” would be returned, and if the sentence also contained some parameter, in this case the location, it would be returned as well, along with the value.

As mentioned before, sometimes the NER detects ambiguities it cannot solve, which are passed to the caller for disambiguation. At this level some disambiguation can take place. When a user is trying to perform an action that has one of the alternatives but not the others, then the ambiguity is solved in favor of that alternative.

For example, in the previous action “show hotels”, if the user could set the maximum number of results saying “show me up to 2017 hotels in Madrid”, the number 2017 would be identified as an ambiguity by the NER, as it could refer to either a number or a date.

However, as the “show hotels” action takes no date as a parameter, it would be interpreted as a number by the API. Disambiguation at this level cannot be complete, as there are certain cases that cannot be solved automatically. For example, if the “show hotels” application has a maximum and a minimum price and the user says “show me hotels for five hundred euros a night”, the chatbot cannot distinguish whether “five hundred euros” refers to minimum or maximum price.

These ambiguities must be solved by the caller of the API, either automatically using some heuristic, or by asking the user which alternative they meant. It would be interesting, however, to give some hints to the API about the semantics of having a “minimum price” and a “maximum price”, which is why the API lets developers specify triggers for parameters that help during disambiguation.

For example, the parameters for minimum and maximum price could be specified as follows:

```
{
 "name": "minimum price",
 "systemType": "currency",
 "triggers": [
   "minimum",
   "above",
   "starting from",
   "from"
 ]
},
{
 "name": "maximum price",
 "systemType": "currency",
 "triggers": [
   "maximum",
   "up to",
   "below",
   "under"
 ]
}
```

This way, a sentence such as “show me hotels above five euros” would be interpreted as having the “minimum price” parameter with value “five euros”. Finally, regarding the output format, the goal was to have a format that could be directly usable by most web applications. Because the output represents actions and their parameters, the most natural and widely used format is Flux Standard Action[1], which is directly consumable by applications using the Flux[2] architecture.

The output format, ambiguities included, for the sentence “show me hotels for five hundred euros in Madrid”, would be the following:

```
{
 "type": "show hotels",
 "payload": {
   "location": "Madrid"
 },
 "meta": {
   "payload": {
     "location": {
       "at": 42,
       "text": "Madrid"
     }
   },
   "ambiguities": [
     {
       "text": "five hundred euros",
       "at": 20,
       "alternatives": [
         {
           "minimum price": {
             "value": {
               "amount": 500,
               "type": "EUR"
             }
           },
           "maximum price": {
                     "value": {
               "amount": 500,
               "type": "EUR"
             }
           }
         }
       ]
     }
   ]
 }
}
```

---

[1] https://github.com/acdlite/flux-standard-action

[2] https://github.com/facebook/flux

5.2 Chatbot middleware

In order to solve the remaining ambiguities, generate responses to user sentences and ask for missing parameters, some logic is necessary. Each of these three functions can be implemented as separate, composable modules, so that a developer can specify which he wants to use. For now, however, all three are included in the current middleware stateful API.

In order to handle conversational context, some state will have to be stored in the middleware, namely variables to indicate if the user answered the question the chatbot asked, if the user filled all the required parameters of an action, and other similar useful variables. Taking into account this contextual information can make the interaction more natural.

The ideal option when handling this state would be to use a standardized, composable pattern, so that the chatbot's state can be incorporated easily into the rest of the state. This follows the Flux pattern, where the state is centralized and managed from a single source of truth.

There is one particular state-handling library, redux.js[3], which implements the Flux state handling pattern. It has been chosen for being the most popular Flux implementation and for its composability: a developer already using redux.js can easily incorporate the chatbot code into his existing codebase with ease.

Additionally, it is simple enough that a user looking to create an application from scratch will be able to take the chatbot middleware as is and add to it the application's logic. The middleware takes as input the actions and parameters returned by the chatbot API, modifies its internal state, then creates a text response for the user and if the user has finished an action, it also returns the name of the action and the parameters with their values.

5.3 Other chatbot-building APIs

Many Artificial Intelligence as a service platforms, or AIaaS, have sprung up during the last few years. Among them, the most prominent are API.ai[4], acquired by Google, Wit.ai[5], acquired by Facebook, and LUIS.ai[6], created by Microsoft [27].

All of these products are based on machine learning: the developers specify sentences and how to interpret them into "intents" and "entities". Wit.ai and API.ai also offer the developer the possibility of connecting webhooks to generate the response dynamically, or call some API when a user triggers a certain intent.

Other platforms, such as pandorabots.io[7] are based on AIML[8], the Artificial Intelligence Markup Language, and require hand-crafting the conversation flow using a descriptive language. But the main, distinguishing trait of all these services is that they are based on learning by example, usually with machine learning algorithms.

---

[3] http://redux.js.org/

[4] https://api.ai/

[5] https://wit.ai/

[6] https://www.luis.ai/

[7] https://developer.pandorabots.com/

[8] http://www.alicebot.org/aiml.html

Developers have to specify examples of sentences and what their “meaning” for the application is. For example, they have to label sentences like “I want to see hotels in Madrid” with the **intent** “see hotels”, and use any entities the algorithm finds as parameters, in this case, Madrid.

In order to add new functions (usually called intents) to the bot, new examples specific to that intent are necessary. And although some services, specially API.ai, provide good generalization with only a few examples, to understand the user consistently, the developer needs to add examples representative of how end users will actually use their chatbot, which is very hard to do without user feedback and a large number of examples for certain complex questions.

Furthermore, these services have very low maintainability. Each intent has completely separate and independent examples that have been added by hand, and it is the developer’s task to make sure these examples are not contradictory, that is, that there are not multiple similar or equal sentences the developer indicated should be handled differently. This can become a daunting task if an intent has two hundred and even more examples and there are multiple collaborators adding examples.

Using example training also makes the chatbots rigid and not reusable: whenever a developer wants to create a new action, he cannot reuse the examples from other intents. If the original requirements only impose that there is a “show hotels” intent, but then the new intent “show rooms” is added, new examples have to be added from scratch, no matter how similar the two are syntactically.

Could this inefficiency have been prevented? Yes, for example if the developer had created a generic “show X” intent and then specified that he had rooms and hotels. But not all chatbot APIs allow developers to do this, and it is still not maintainable, because now if through usage analytics it is discovered that most users try to see the available rooms saying “give me the rooms”, or a synonym of the verb give, then only the “show X” intent can be modified, affecting also the “show hotels” intent.

This example was merely used to illustrate the underlying problem: that the developer is forced to handle **entities** and **intents** (foreign to the realm of applications) instead of **parameters** and **actions**. The user must first understand what entities and intents represent, which is not that well documented, then manually map intents to actions and entities to parameters. In some cases, there are ambiguities which the platform creates, due to its inability to provide context.

Take the sentence “show hotels between fifty and two hundred euros a night”. If there is no way to specify that “between” makes fifty the minimum price, there is an ambiguity the user has already solved, and asking “is fifty the minimum price?” will make the user see the chatbot as having no intelligence, shattering the veil of intelligence it may have had.

This project takes a different approach to the existing tools in the way applications with it are built and how the conversation is handled. Instead of having a nondeterministic black box that needs to be trained with examples, the user must define conditions for actions to trigger. These conditions are independent and composable to ensure maximum reusability.

Returning to the “show hotels/rooms” example. The developer could use two conditions: the presence of a “show” verb and the presence of a “hotel” synonym in the sentence. Because conditions are independent from each other, they can be created and maintained separately. If requirements change and the developer needs to make the “show rooms” action also look for synonyms of “give”, then it suffices to create a composite condition made up of either a synonym of give or a synonym of show, without changing the “show hotels” action in the least.

Writing and labeling examples of sentences by hand turns out to be a time-consuming task, which many developers do not have the time to do. As mentioned before, it creates a pile of data tied to the platform used to develop the chatbot that cannot adapt to changes in the requirements. And with a chatbot, there is almost a guarantee of changes in the requirements, as a group of developers can hardly correctly model user interaction *without* any user feedback, which is the entire reason for user centered design[28].

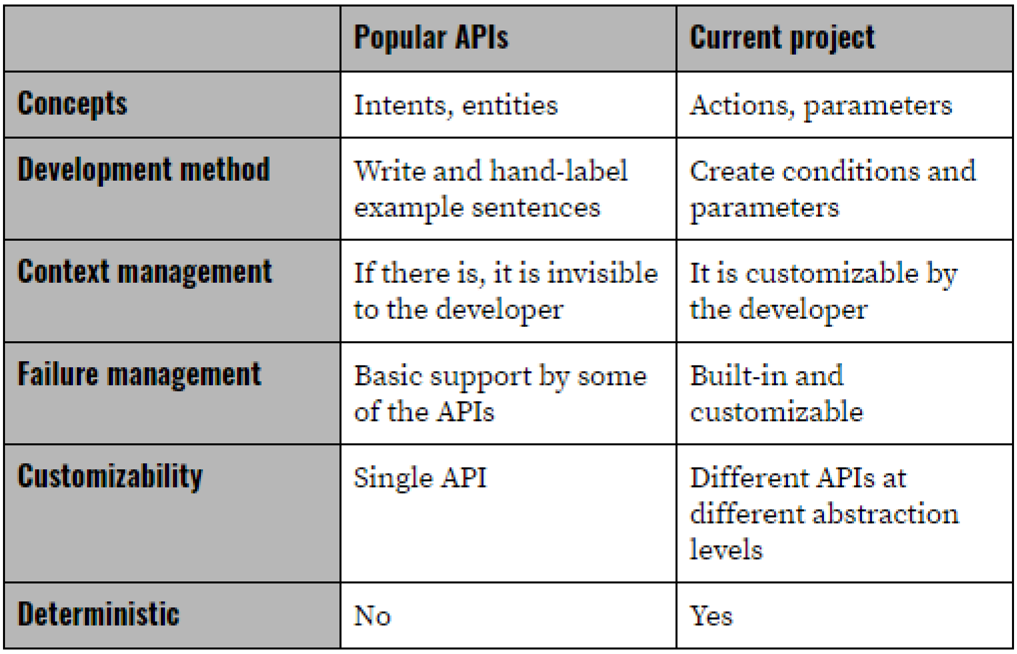
Instead, it will be thanks to the user interaction collected by usage analytics tools that the chatbot will be able to improve and adapt to how users actually use the application. Developers of a hotel-booking chatbot may think that most users will say “show me the hotels in Madrid”, or a similar sentence to see the hotels, but a large fraction of users may actually say “what are some nice hotels in Madrid?”, which is radically different and will hardly be captured as the same if the chatbot was trained from examples.

Apart from the problems with the way popular framework’s chatbots are currently trained, one more problem has already been pointed to by others: that “The APIs learn only from example and do not provide ways to take advantage of additional domain knowledge”, quoted from an article by a chatbot-building startup named Conversate[27].

The same article also points out to a problem this project has tried to solve: that NLP and chatbot APIs provide only basic text-to-entity or text-to-intent mapping. The rest of the development work: building the UI, handling the state, keeping a conversational context is left entirely up to the developer, and they provide no optional libraries or services to carry that burden.

And it is precisely those things which take up a lot of developer time and prevent users without coding knowledge from creating chatbots. Other important matters that are seldom handled or even considered by currently used APIs is “Failure management”, as the Conversate article says, which is how the chatbot deals with failure to understand the user.

As a summary, these are the main differences in approach or execution between the present program and the currently popular AIaaS platforms:



	Popular APIs	**Current project**
Concepts	Intents, entities	Actions, parameters
Development method	Write and hand-label example sentences	Create conditions and parameters
Context management	If there is, it is invisible to the developer	It is customizable by the developer
Failure management	Basic support by some of the APIs	Built-in and customizable
Customizability	Single API	Different APIs at different abstraction levels
Deterministic	No	Yes

Figure 5.1: Entity types, their format and output.

CHAPTER

6

USE-CASE: HOTELBOT

The present chapter will show a possible use-case for the chatbot API that has been used to better introduce some of the concepts so far: HotelBot. It will show how a developer can create a chatbot application that allows users to book hotel rooms using this chatbot API. The steps to create an application are the following:

1. Define the requirements
2. Create a valid application specification
3. Provide the specification
4. Create or adapt an interface that calls the API

The first thing the developer would do is to create a specification for the HotelBot that contains the actions his users will do. For simplicity purposes, it will be assumed there are only two actions: "show hotels" to see available hotels in a certain area or price range and "book hotels", which allows the user to make the reservation.

A possible application specification that defines both actions is the following:

```
{
  "name": "Hotelbot",
  "actions": [
    {
      "name": "show hotels",
      "parameters": [
        {
          "name": "location",
          "systemType": "location"
        },
        {
          "name": "minimum price",
          "systemType": "currency",
          "triggers": [
            "minimum",
            "above",
            "starting from",
            "from"
          ]
        },
        {
          "name": "maximum price",
          "systemType": "currency",
          "triggers": [
            "maximum",
            "up to",
            "below",
            "under"
          ]
        }
      ],
      "conditions": [
        {
          "systemType": "enum",
          "validationSchemaFile": "show.json"
        },
        {
          "systemType": "enum",
          "validationSchemaFile": "hotel.json"
        }
      ]
    },
    {
      "name": "book hotel",
      "parameters": [
        {
          "name": "room count",
          "systemType": "number"
        }
      ],
      "conditions": [
        {
          "systemType": "enum",
          "validationSchemaFile": "book.json"
        },
        {
          "systemType": "enum",
          "validationSchemaFile": "hotel.json"
        }
```

```
            ]
        }
    ]
}
```

Once the application is validated using the JSON schema provided in the documentation of the API, and it has been uploaded, the developer is provided with an id to uniquely identify calls to his application. Using a provided interface template (which would already contain the API call) or a custom solution, the users can now begin talking to the chatbot API, which would respond with structured text.

When the HTTP call hits the API, it loads the application specification created by the developer from either cache or the storage server. Once loaded, it extracts the entities from the sentence. Because the developer has only specified actions, conditions and parameters, these first need to be mapped to entities. Each parameter and condition is mapped to a custom entity type during the import process, so that they can be identified during extraction.

Entity extraction is a process made up of multiple steps. To illustrate each step, the processing of the sentence "book me one room there" will be used:

1. **Tokenization**: The sentence is divided into substrings or "tokens", typically corresponing to a single word. The sentence would be separated into "book", "me", "one", "room" and "there".

2. **Feature extraction**: Tokens are assigned the features corresponding to the regular expressions they fit. The tokens "book" and "room" would be assigned the feature *custom* also indicating which custom entity it has matches, "one" would be assigned the feature *cardinal*, and "me" and "there" would not be given any features.

3. **Classification**: Tokens are classified into entity types depending on their features. "room" and "book" are assigned the *custom* entity type, "one" is assigned the *cardinal* type, and finally "me" and "there" are given the "any" type, as they have no features.

4. **Normalization**: Tokens are given an appropriate value based on their type. The only token that is normalized in the sentence is "one", which is assigned the numerical value 1. The rest have as value the string they match, e.g. the token book has the value "book" and the token room has the value "room".

5. **Entity building**: Entities are created from the entity types and the values found for each token. At this point, tokens with features that are not enough to constitute an entity are discarded, such as "me" and "there". If tokens have multiple entity types, they are considered ambiguities, to be dealt with at a later stage.

After entities are extracted, the chatbot API tries to find an action in the application whose conditions can all be matched with the available entities. Then, using either the action found, or the action specified by the developer, the chatbot will find which parameters in the action can be fit by the entities. If there is no action in progress, the possible parameters are returned as an ambiguity, indicating the actions they can belong to.

Once the response reaches the caller, it can be dealt with in several ways. One simple pattern is to maintain some local state to keep track of the progress of each action. This way, when the user says "can you show me some hotels", the application receives a response indicating the user wants to "show hotels", and can consequently make a call to its backend, retrieve the hotel information, and show it to the user with any combination of text, voice, image and video.

Subsequent actions will modify the conversational state, so that the user can specify filters like "I only want to see those in Madrid". In this case, the chatbot API recognizes "Madrid" as a location parameter, and the application can use this information to show only those hotels which are in Madrid.

The more variables the developer decides to keep in his application, the richer the conversation context will be, offering an experience as akin to human conversation as possible.

6.1 Example conversation

This section will show an example interaction between a user and the HotelBot. The user is trying to find and book a hotel. For simplicity purposes, it will be assumed that the user already has a bank account linked to his HotelBot account.

User: "i want a hotel for today" HotelBot: "Where do you want the hotel?" User: "in madrid" HotelBot: "Ok, there are sixty-four hotels in Madrid available today" User: "show me only those above fifty euros" HotelBot: "Looks like there are twelve hotels in Madrid starting at fifty euros available today" User: "Tell me more about them" HotelBot: "The first one has four stars. The price is 74 euros a night. Do you want to know more?" User: "no, show me the next" HotelBot: "The second one has three stars. The price is fifty-six euros a night. Do you want to know more?" User: "yeah" HotelBot: "It's located in Plaza del Callao. The description reads 'The best hotel to spend the night in the heart of Madrid'. User ratings give it an average of four stars." User: "does it have a room with three bedrooms?" HotelBot: "Yes" User: "k, I want to book it" HotelBot: "How many rooms?" User: "only one" HotelBot: "Do you want to use your regular payment method?" User: "Yes" HotelBot: "Your room is ready. Register at the hotel to receive your key. Have a great stay!"

In this example, the application uses an additional action not defined here, "know more", to show the user information. This action would need code in the application to create the response based on the returned information from the hotel booking backend or third-party API used.

CHAPTER 7

CONCLUSIONS

The three project goals set at the start have been fulfilled: the chatbot API has been successfully developed and used in a contest, proving an application built using it works, and the documentation to use the API is currently publicly available[1]. Additionally, the API fulfills all the requirements.

This is an ambitious project that has just begun. There are yet many tasks to do before it is ready for production, as mentioned in the future improvements section.

These initial results show that it will soon be possible to build and deploy smart applications in a few hours, specifying only the minimum necessary. Building usable chatbots, however, is not so simple, and requires monitoring user feedback and making changes to adapt to how people truly interact with the application. Most likely, this or other project will provide the tools to make that possible.

7.1 Future improvements

There are many things to improve moving forward. Some of these have been mentioned throughout this document, others have been left out. The following is not a comprehensive list of future features and goals, but rather a summary of the most relevant:

1. Available in other languages. Currently the application works only with english. Writing a NER for other languages is an unsustainable endeavor. Instead, the more viable choice is to have a translation service convert those other languages to english. Because the responses are generated by an independent module, these can be generated directly in other language if the developer wishes to do so.

2. Improve the NER: From adding support for non-regular types such as names to adding new date entity patterns, the NER can be greatly improved to make sure the most entities are recognized and normalized correctly.

---

[1] https://github.com/fpluis/chatbot-docs

3. Create the public chatbot libraries: Making the developer's life easier is the entire goal of this project. Therefore, new and existing libraries will try to be as easy to use and interoperate as possible.

4. Develop the platform to sell this program as a service: So far the API is ready to accept incoming requests and let developers specify applications. However, there is no website to create an account, set up billing, and define the application without having to manipulate the JSON file.

5. Add machine learning to improve recall: Some entities such as names or locations can only be reliably recognized using machine learning techniques. However, in order to both take advantage of machine learning and have a deterministic system, the developer will be able to fix the way certain sentences are interpreted, and even to disable machine learning altogether.

6. Add machine learning to provide developers with **user feedback**. Knowing the sentences most often used, and the usual interaction routes for users can be very helpful to improve the chatbot. Information about usage can also be used to know if there are certain sentences that users commonly use but the application fails to recognize or recognizes them incorrectly.

7.2 Participation in the Whenhub hackathon

The application[2] submitted to the Hackathon contest organized by Whenhub[3] was awarded the "Best overall" prize. Whenhub is a company that provides services to create schedules and events that can be shared with others, being useful for events like conferences. These schedules are stored as data without style, and can be seen with any of the visualization they provide.

Because they are a new company, they wanted front end developers and designers to supply them with ideas for these visualizations and also backend developers to build products with their API. And those were the three categories they created: backend, frontend and designers, the latter two submitting visualizations.

The chat UI was in part developed by Artem Rudenko, a colleague and current student at the UIB. The rest, including the application's logic and state-handling, as well as part of the UI was developed by me. The entry was submitted to the backend category, although it used a new visualization Whenhub did not have: text-based, which is more accessible than others based on images.

The application allowed the user to create, list, update and delete schedules and events. Additionally, the functionality of "time-shift" was added, which allowed users to shift events forward or backward in time by simply saying a sentence such as "timeshift it forward two weeks". Me and my colleague found this useful for use-cases where a series of events are rescheduled but their relative position in time remains constant.

The entire application, as well as the deployment of the API and testing, took a total of 45:44h between the first and eighth of May, 2017. Artem Rudenko and I have already received the 5000$ prize.

---

[2] The contest entry can be found at https://devpost.com/software/whenbot

[3] https://www.whenhub.com/

7.3 Personal valoration

When I decided I would build a chatbot API, in mid-February, I did not even know what natural language processing was, or how long it would take me to build a system that would parse human language into actionable data, provided some context. Not even if the result would be acceptable.

But at the time of writing this documentation, four months and over 450h of work later, I have learned many things about language and how to process it, although there remain much more to learn. I have seen just how different it is to develop an application using only a chatbot versus creating a responsive UI, and the experience was very positive.

The very reason I started this project was so people who have not written a line of code, not just developers, could build useful applications without having to worry about how to make UI elements responsive, positioning them on the screen and making sure the application works on different browsers or operating systems. In part because I do not want to do all those tasks, but I still want to build web applications.

Developing the Whenhub application was much like developing any “backend” application. All I had to do was think about which functions I had, how state was handled, and how to connect what users said to the backend calls I had. When I wanted to add a new action, I simply modified the API configuration file and added a new handler to the chatbot API response.

I still intend to improve this experience and provide as much information to the developer as possible about how users really use their application, because that is key to making the app useful and usable. As I developed the Whenhub application, one of the people whom I asked to test it tried to change the description of an event using a combination of words I had not thought about, which was immediately added.

Just like user interfaces, chatbots need user feedback because developers can hardly imagine all the possible ways a user will interact with their application.

BIBLIOGRAPHY

[1] B. A. Shawar and E. Atwell, "Using dialogue corpora to train a chatbot," in *Proceedings of the Corpus Linguistics 2003 conference*, 2003, pp. 681–690. 1

[2] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures.* University of California, Irvine Doctoral dissertation, 2000. 1.1.2

[3] M. Fowler and J. Lewis, "Microservices," *ThoughtWorks. http://martinfowler.com/articles/microservices. html [last accessed on February 17, 2015]*, 2014. 1.1.2, 3

[4] J. Nielsen. Response times: The 3 important limits. [Online]. Available: https://www.nngroup.com/articles/response-times-3-important-limits/ 1.1.3

[5] M. Roberts. Serverless architectures. [Online]. Available: https://martinfowler.com/articles/serverless.html 1.2.1

[6] R. C. Martin, *Clean code: a handbook of agile software craftsmanship.* Pearson Education, 2009. 1.2.2

[7] E. Elliott. Master the javascript interview: What is a pure function? [Online]. Available: https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-pure-function-d1c076bec976 1.2.2

[8] Usage share of web browsers. [Online]. Available: https://en.wikipedia.org/wiki/Usage_share_of_web_browsers 5

[9] M. Negnevitsky, *Artificial intelligence: a guide to intelligent systems.* Pearson Education, 2005. 2

[10] J. Weizenbaum, "Eliza—a computer program for the study of natural language communication between man and machine," *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, 1966. 2

[11] Phrasee. Parry: The a.i. chatterbot from 1972. [Online]. Available: https://phrasee.co/parry-the-a-i-chatterbot-from-1972/ 2

[12] A. Rodrigues. A history of smarterchild. [Online]. Available: https://motherboard.vice.com/en_us/article/jpgpey/a-history-of-smarterchild 2

[13] D. Nadeau and S. Sekine, "A survey of named entity recognition and classification," *Lingvisticae Investigationes*, vol. 30, no. 1, pp. 3–26, 2007. 1

[14] S. Batra and D. Rao, "Entity based sentiment analysis on twitter," *Science*, vol. 9, no. 4, pp. 1–12, 2010. 3

[15] D. Mollá, M. Van Zaanen, and S. Cassidy, "Named entity recognition in question answering of speech data," in *Proceedings of the Australasian Language Technology Workshop*, 2007, pp. 57–65. 3

[16] J. Tang, M. Hong, D. Zhang, B. Liang, J. Li *et al.*, "Information extraction: Methodologies and applications," *Emerging Technologies of Text Mining: Techniques and Applications*, 2007. 3

[17] M. Souza and R. Vieira, "Entity-centric sentiment analysis on twitter data for the potuguese language," in *Proceedings of the 9th Brazilian Symposium in Information and Human Language Technology, Fortaleza, CE, Brazil*, 2013, pp. 21–23. 4

[18] G. Zhou and J. Su, "Named entity recognition using an hmm-based chunk tagger," in *proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2002, pp. 473–480. 4

[19] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," 2007. 1

[20] L. Chiticariu, Y. Li, and F. Reiss, "Rule-based information extraction is dead," *Long Live Rule-Based Information Extraction Systems*, 2013. 2

[21] A. X. Chang and C. D. Manning, "Sutime: A library for recognizing and normalizing time expressions." in *LREC*, vol. 2012, 2012, pp. 3735–3740. 4.3

[22] A. McCallum and W. Li, "Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons," in *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*. Association for Computational Linguistics, 2003, pp. 188–191. 4.3

[23] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167. 4.3

[24] T. Weiss. Deep spelling. [Online]. Available: https://medium.com/@majortal/deep-spelling-9ffef96a24f6 3

[25] Date format by country. [Online]. Available: https://en.wikipedia.org/wiki/Date_format_by_country 7

[26] J. NR. What is an isomorphic application? [Online]. Available: https://www.lullabot.com/articles/what-is-an-isomorphic-application 5

[27] Conversate. A review of natural language apis for bots. [Online]. Available: https://medium.com/@Conversate/natural-language-apis-for-bots-e791f090e32f 5.3

[28] C. Abras, D. Maloney-Krichmar, and J. Preece, "User-centered design," *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications*, vol. 37, no. 4, pp. 445–456, 2004. 5.3