Politecnico di Torino

Dipartimento di Automatica e Informatica

Corso di Laurea in Computer Engineering

Internship final report

Chatbot development,
RPA to support enterprise processes

Author:
Edoardo Debenedetti

Supervisors:
Dott. Oscar Pistamiglio
Prof. Fabrizio Bonani

2018-19 Academic Year

Summary

This is the report of the internship I had at the IT consulting firm Blue Reply s.r.l.

Having been assigned an R&D task, I have been the first in the firm to work on Robotic Process Automation and Intelligent Process Automation, to automate routine processes. I worked on an automatic e-mail classifier and on a PowerPoint presentation generator. Another project I took part to has been the development of a chatbot that answers questions about GDPR. I also contributed to the creation of a project proposal for a firm that manufactures and distributes fashion products.

Contents

1 Reply

1.1 Reply

Reply is one of the leader firms in Italy in IT Consulting. It has more than 7,000 employees and in 2018 it had turnover bigger than 1 billion €. It was founded in 1996 by three IT managers, and since then it has been one of the fastest growing companies in Italy. Its structure is very peculiar, since Reply owns a series of companies, each one specialized in one specific set of technologies or topic. In fact, Reply groups more than 80 companies in Italy, and more than 110 companies in the whole world. For instance, the Reply company I worked in, Blue Reply, works with IBM technologies, while Go Reply works with Google ones. In some cases, it might even happen that they are competitors when they apply to get a project. Each Reply company offers, among the others, services such as system integration, machine learning, cloud, internet of things — IoT, and e-commerce solutions. It operates in many industry sectors, like banking & insurance, automotive, healthcare, public sector, energy & utilities, telecommunications, distribution, and transportation.

1.2 Blue Reply and the Cognitive BU

Blue Reply counts more than 400 employees, divided in 16 business units. It was founded 2000, with the aim of enabling company to reach digital transformation, through services such as consultancy, design and implementation of solutions based on IBM technologies. Indeed, it is IBM Gold Business Partner. Notwithstanding this, Blue (as it is shortly called by its employees) also uses open-source technologies such as JavaEE, TypeScript, Docker, Kubernetes, Angular and so on. It works with some of the biggest European firms, in sectors such as finance, manufacturing, telecommunications, media, and retail, defining new business models supported by the design and the implementation of systems based on machine learning, cloud computing and IoT. The main focuses of business units are:

- Design and implementation of strategic, multi-channel, mission-critical solutions based on modular architecture and process-based integration systems.
- Design and implementation of vertical solutions for banking, consumer finance, insurance, manufacturing and luxury sectors.
- Cross-industry business consulting solutions.
- Implementation of digital commerce and digital marketing solutions, taking care of social collaboration.
- Design and implementation of information management and business intelligence systems.

- Creation of solutions for application security, cloud and IoT infrastructures.

Among the business units in Blue, I was assigned to the *Cognitive Business Unit*, specialized in cognitive solutions for companies. For instance, it creates chatbots powered by computer vision capabilities, voice assistants and smart search engines. During my internship, my teammates worked on projects for companies such as an insurance firm, a supermarkets group leader in Italy and several multi-national luxury companies.

2 RPA and RPA tools

One of the main topics I covered during my internship is Robotic Process Automation. In this chapter I will briefly introduce Robotic Process Automation, Automation Anywhere, and some Python automation libraries.

2.1 Robotic Process Automation

Robotic process automation (or RPA) is the automation of business processes, based on software robots (or bots). Bots can be used to automate routine and repetitive processes, accelerating companies' work-flow and allowing their employees focus on more important and fulfilling tasks. According to Transparency Market Research, RPA is expected to reach a $4.98 billion market cap by 2022, from $183 million in 2013 [2]. RPA can often lead to a 30-40% improvement in the cost and time to perform a process [1].

2.1.1 RPA advantages

RPA unveils huge advantages, both to companies and their employees. In fact, it:

- Takes employees away from repetitive, boring and frustrating tasks, such as copy-paste activities.
- Tasks executed by bots are run in a faster, scheduled and non-stop way, but, most of all, are error-free (given that the bots were correctly programmed)
- Since bots interact directly with the UI exactly as if they were humans, it needs no integration with APIs. RPA can indeed work with legacy systems without the need of changing the preexisting infrastructure.

2.1.2 RPA Risks

According to some Gartner, Inc. research papers about RPA, it has also some risks. In fact, RPA should be reputed a sort of temporary patch, to be applied while waiting for a "true" better-integrated, faster and more reliable automated system. This is due to the fact that RPA can affect enterprise agility, because of the link that RPA scripts have with the UI's behavior that makes them rigid. By 2020, more than 20% of the organization that already used RPA, will have substituted it with faster solutions with lower running and complexity costs. Because of the lack of clarity about the technology, RPA can also create misunderstandings inside enterprises. Thus, firms should be clear about what it is, why and how it works. Moreover, it should be clear that not all processes can be automated and that trying to automate such processes could be a waste of resources and time.

2.1.3 RPA and AI

RPA can be successfully integrated with several Machine Learning — ML — solutions. Either Automation Anywhere, Blue Prism and UiPath use some ML solutions, both to improve the basic features and integrate some more complex ones. In the former case, computer vision can be used to better recognise the elements on the screen. This improves the reliability of the bots that heavily use mouse movements and whose behavior is dependent on the position of the elements on the screen. In the latter case, ML is used to add some features that can make bots smarter. For instance, Automation Anywhere, that recently partnered up with IBM, integrates an interface with IBM's business decision making suite, in order to let the bot make decisions based on the data it gathers and act consequently. ML is also used for sentiment analysis and images recognition, to enable the bot to accomplish more complex tasks that would normally require some human intervention.

2.2 RPA tools

The widest used tools for RPA include, but are not limited to Automation Anywhere Enterprise, UiPath, Blue Prism, suites used to create, run and orchestrate bots.

2.2.1 Automation Anywhere

Automation Anywhere Enterprise is a tool developed by Automation Anywhere, used to develop and orchestrate bots that directly interacting with the GUI running some tasks. A complete Automation Anywhere environment includes three machines:

- The *Bot Creator*: a machine where an IDE is run. It is used to create bots through a drag-and-drop suite, that, however, has a programming logic and lets the developer use control flow statements such as `if-else`, `while`, and `for`.
- The *Control Room*: a web server that serves both a website, used as administration panel, and authentication and synchronization services. It is used to orchestrate the bots.
- The *Bot Runner*: a machine where the bots are deployed and run after the activation by the Control Room.

The Bot Creator The drag-and-drop suite used in the Bot Creator offers many actions, such as *Open program*, *Wait for window*, *Keystrokes*. One of the most useful is *Clone object* (that is an ambiguous name). It emulates an action (that can be done both with the mouse and the keyboard) on an object inside a window (e.g. a button, or a text field). This object can be recognized by Automation Anywhere in several ways, such as the position in the DOM (mainly in web pages), its coordinates, its name, or its content (e.g. some text).

The Control Room The Control Room is a web server that requires to be run on Windows Server. It offers a control panel from which the administrator can manage bots, activating, scheduling and synchronizing them between devices.

2.2.2 Open source Python automation libraries

Due to some limitations in Automation Anywhere Enterprise we will see later, the use of some open sources Python libraries for some tasks turned out in being easier and more efficient. In fact, the PowerPoint presentation generator has been implemented with Automagica [14] and PyAutoGUI [16]. The former module requires `Python 3.7`, while the latter has no specific Python version requirements. Both modules can be installed via `pip`, with no other prerequisites on Windows (the OS I worked on). These libraries can be used to automate the GUI interaction, such as keystrokes, mouse moves, clicks, and drag-and-drops.

Automagica A part from mouse and keyboard tasks, Automagica offers an interface to achieve some complex web-based tasks, such as the monitoring of specific elements of web-pages and the filling of specific fields. These tasks are accomplished via a Google Chrome Selenium [6] instance. Selenium is a tool used to automate browser-based tasks that can be accomplished in different ways. One of them is referring to the `XPath` of the element we want to work on. The `XPath` of a web-page element is used to locate its node inside the DOM. It can be easily obtained using the "Inspect element" mode of a web browser. Identifying an element via its `XPath` is a very reliable method, since the `XPath` of an element doesn't depend on the position of the element on the screen (and thus on the screen size and resolution), nor on the content, or the size of the element. Still, it works even after a change in the page's look (unless the position of the element itself, or its node tree changes). In some cases, Automagica fails in doing some actions, such as dragging and dropping elements. This is the reason why I decided to use also **PyAutoGUI** to complete the actions Automagica can not do.

3 E-mail classifier

The first task I was assigned is an automatic e-mail classifier, using Microsoft Outlook and Automation Anywhere Enterprise, a robotic process automation suite.

3.1 Bot's functionalities

The business unit I am part of has a common mailbox, in which all the e-mails concerning the working group are received. However, many e-mails do not involve all the people that are part of the unit, thus, each of them looses time looking for the right e-mails to be read. A system that automatically categorizes messages in predetermined folders, one for each project, would help the BU very much. Since the company wanted to test some Robotic Process Automation applications, I was assigned the task of realizing such system, using Automation Anywhere on Microsoft Outlook. The bot, then, should, in order:

1. **Open** Microsoft Outlook and wait for the mailbox to load.
2. Take all the elements that **characterize** the first message in the mailbox, i.e. subject, sender, receiver, CCed people and the message body.
3. Send a request to a REST API to **classify** the message and receive the name of the folder to put the message into.
4. **Move** the message to that folder.
5. **Repeat** steps 2 to 4 for each message in the mailbox, until it is empty.
6. **Close** Microsoft Outlook.

I decided to create an API to classify the message, both to make the solution more scalable and because Automation Anywhere is not thought to be used to implement non-trivial algorithms. Thus the bot is composed of 2 parts: the Automation Anywhere task and the API, that I developed using **TypeScript** programming language, and that runs on **NodeJS** (more information about TypeScript and NodeJS can be found in 5.3.1). Thanks to Automation Anywhere, the task can be triggered or scheduled using the Control Room. Now let us see more in the detail how both the task and the API work.

3.2 The Automation Anywhere task

The first thing the task does is opening Outlook (using its path on `C:/` and using the function `Wait` to wait for the program to be completely loaded. Then, using the equivalent of a `while` loop, the task opens each message in the inbox. Once the message has been opened, we select each field of the mail (i.e. subject, sender, receiver and CCed addresses), using the *object cloning* function, and save it to a variable. This is done copying each piece of text to the clipboard

emulating `CTRL+C` keystrokes and saving the clipboard to a variable. The object cloning function has a misleading name: in fact, it does not copy, nor it does not clone anything. It actually lets the bot recognize and act on an object on the screen using its characteristics such as, for instance, the text contained in the object, its name and hierarchy in the DOM, or its position on the screen (even though the last one is not a reliable characteristic at all, as the position on the screen strongly depends on the screen resolution). Once all the information is gathered, the bot queries the API with a POST request formatted as a JSON that contains the email fields, and collects the answer, that is simply the name of the folder where the email should be moved to. In fact, once the answer is retrieved, using the object cloning function, the bot first opens the "Move to" menu, and then clicks on the menu entry with the name of the folder given by the API. Once all the emails have been categorized, the task closes Microsoft Outlook and ends.

3.3 The Node.js API

The API is built on top of Node, using TypeScript programming language, transpiled to JavaScript at the deploy stage, in order to be run. I used the *Express.js* library, a framework written in JavaScript and made for Node.js, that provides a set of features that truly simplify the development of REST APIs [8]. In fact, in order to create a server, we simply have to create an instance of `express`, and then use its method `app.post()` with the path at which the post request (in this case it is `'/classify'`) and a callback function that handles the request. In the callback we create an instance of the class `Classifier`, and use `Classifier`'s `classify` method. The server then checks whether the confidence of the classification is higher than the threshold declared in a settings file. If this is so, the name of the folder is sent back, otherwise *Not classified* is sent back (in case of missing classification, the message is put in the "Not classified" folder).

The `Classifier` class This class is used to classify the email using the data sent with the POST request. Data conform to the `MailData` interface, that contains the fields `from`, `to`, `cc` and `subject`. The class uses a JSON whose data, used to classify, were inserted manually by me. I did this to simulate a natural language classifier, which I could not create, due to a lack of data to train a true, fully functional model. The `classify` method, then, estimates what project the email is related to (that is the one with the highest confidence) and returns both the name of the project and the confidence of the estimation.

4 Technical report generator

Looking for other applications for RPA, I was given the task of developing a bot that automatically creates a technical report in form of a Microsoft PowerPoint presentation.

4.1 Bot's functionalities

One of the services offered by the business unit I worked in is the creation of cognitive search engines that enable customers to build helpful and easy-to-use platforms. A customer requires my team to send weekly (seldom daily) reports about their search engine performances. The report is a PowerPoint presentation with some screenshots of statistics and plots about the service performances (e.g. memory heap, CPU usage, number of requests, number of errors etc.) along the period requested by the customer. There are two servers that work as back-end from the same server, thus each slide of the report should include the plots of both servers, in order to be compared. The creation of the report requires the visit of more than 20 web pages. Reaching each page takes many steps, due to the complex design of the monitoring platform, and is highly time consuming. The average time to create manually one complete report is more than 50 minutes. Therefore, since this is a boring and repetitive task, we decided to automate it. As a result, the bot can complete the task in less than 10 minutes, and can work without any kind of supervision on a virtual machine, letting the user to do something more interesting and productive, while the bot generates the report. In order to create the report, the bot should:

1. Ask the user for the **time range** of the report.
2. Open the **template** file on PowerPoint.
3. **Write the time range** of the report on the presentation's cover.
4. **Open a web browser** and authenticate into the monitoring platform.
5. **Iterate** over a list of URLs to be visited and take a **screenshot** of each plot.
6. **Paste** each plot (two by two) on the presentation and resize it to fit into the presentation.
7. **Save** the presentation with a filename that includes the time range.
8. **Close** PowerPoint and the browser.

I started developing the bot using Automation Anywhere, but I came across some bugs. In fact, not all instructions were executed in order, and some keystrokes were simulated after some others that should have been simulated before. Thus, I decided to explore the market of RPA platforms, and I discovered two open source Python libraries that I decided to use to develop the

bot: Automagica and PyAutoGUI (it can be found more about these libraries in subsection 2.2.2). The bot, then, is a Python script to be run and used with the command line.

4.1.1 Bot implementation

As stated above, the bot is written in Python, using Automagica and PyAutoGUI. In order to keep the script reliable and independent of the screen size, the interaction with the UI is entirely done with the keyboard and doesn't rely in any way on the mouse. The script is divided in three files: `uebaPPTGen.py`, where there are all the fundamental instructions, `utils.py`, where there are some functions wrapping some others from Automagica and PyAutoGUI (more about `utils.py` in 4.1.2), and `urls.py`, where all the URLs to be visited are saved. In fact, fortunately, each page to be visited has a specific URL, where the timestamps of the time range can be attached, enabling the bot to build a proper URL for each page. All the URLs are stored inside an array over which the script loops.

First of all, the script prompts the user for the time range to be used for the report generation, and the user is given three possibilities:

1. 8.30 - 17.30 of the current day.
2. The last 7 days.
3. A custom time range.

The monitoring platform accepts UNIX timestamps as GET parameters in the URLs, thus the time range should be generated accordingly. If the user chooses the first or the second option, proper start and end timestamps are generated, using some functions I wrote in the utils file. Otherwise, the user will have to input on the command line the UNIX timestamps (for the sake of simplicity, the script suggests a website where "human" timestamps can be converted to UNIX ones). I implemented a simple function to check whether the timestamp is a valid one, to avoid runtime errors once the script has started generating the report. The last action the user must do is writing the password to access the monitoring platform. I decided to prompt the user for the password each time for security and maintainability purposes. Once this last step is done, the bot starts creating the report.

First, the script opens the model PowerPoint presentation and writes on the cover page the time range the report is relative to (in a human-readable format). Second, it opens the browser on the login page and types username and password. In order to check that the part of the web page we're interested in is loaded, I wrote a function that checks whether the element exists on the page. The element is identified with its XPath (more about the XPath can be found in section 2.2.2). Once the page has fully loaded, the script starts a loop that consists in: opening the web page of the plot to be copied and waiting for it to be loaded, taking the screenshot, changing window to PowerPoint and paste

the image. Every two images, the script selects both the images on the slide (pasted images are by default put at the center of the slide), resizes them to be fit in the slide, moves the first image on the top and the second on the bottom, and finally moves to the next slide. This loop is repeated until the script reaches the end of the URLs array. The report file is finally saved and PowerPoint is closed.

4.1.2 `utils.py` file

In this file, I created some functions that wrap some others from Automagica and PyAutoGUI, to make the code cleaner and more readable:

- `waitForPage()` is used to wait for a specific web page to be loaded. It uses some Automagica's functions to read the title of the browser page.
- `waitForElement()` is used to wait for a specific element in the web page to be loaded. It tries within a loop to select the element using its XPath. If Automagica can't find the element, raises an exception. The exception handler sets a flag to false, that makes the function loop again. Once the element is loaded, Automagica doesn't rise an exception any more, and the function returns.
- `openProgram()` opens a specific program, using Windows' *Run* window.
- `openAndWaitForProgram()` opens a specific program and waits for it to be fully loaded.
  It uses the `openProgram()` function and Automagica's `ProcessRunning()` to check whether the program is running.
- `getDate()` returns a tuple containing start and stop timestamps that conform to the monitoring platform date-time format.
- `copy()` function emulates `CTRL+C` keystrokes.
- `paste()` function emulates `CTRL+V` keystrokes.
- `maximize()` maximizes the current window using `CTRL+WIN+UP`.
- `killCurrentWindow()` kills the current window using `ALT+F4`.
- `takeScreenshot()` takes a screenshot that starts from a chosen point and has a certain width. Both characteristics are passed as arguments. The function opens the snipping tool, emulates `CTRL+N` keystrokes (that is like pressing on the *New screenshot* button), and drags the mouse (keeping it clicked) from the start point to the end point. It finally copies the screenshot to the clipboard and closes the snipping tool.
- `changeWindow()` function changes the current window emulating `ALT+TAB`.

- `resizeAndDistribute()`, using a long combination of keystrokes, resizes the two pictures and puts the first one on the top of the slide and the second one on the bottom. This function exploits the fact that PowerPoint can be completely controlled using the keyboard. In fact, pressing `F10`, some letters and numbers label almost all the buttons in the window. Pressing each key is like clicking on the button labeled by the key.
- `checkTimestamp()` checks that the timestamp input by the user has a valid format.
- `epochToString()` converts a UNIX timestamp into a readable format timestamp.
- `epochToStringForFile()` works as the one above, with the difference that the format of time and date is thought to be included into the filename.

5 GDPR bot

After working on RPA, I asked for a different topic to work on. Thus my manager assigned me a new challenging task: the development of a chatbot that answers questions about GDPR.

Notwithstanding all the information that end users can look for on the web, there is still often little clarity about the main points introduced in GDPR, and it is difficult to find a user-friendly source that makes finding the right information easy. This is the reason why we decided to develop the aforementioned chatbot, with the aim to make GDPR more accessible to final users.

5.1 GDPR

GDPR—short for General Data Protection Regulation—is a relatively new law that created one set of data protection rules for all companies operating in the European Union, wherever they are based [4]. It applied from May 25 2018 [3] and, since then, it revolutionized the way the users' data should be used and is having a huge impact on companies' approach to data processing, protection and privacy. This required companies to have great flexibility and a non-negligible effort to become compliant to the new principles.

The main pillars introduced in GDPR are [5]:

- Privacy policies will have to be written in a clear, straightforward language.
- The user will need to give an affirmative consent before his/her data can be used by a business. Silence can not be considered consent anymore.
- Businesses will need to clearly inform the user about transfers outside the EU.
- Businesses will be able to collect and process data only for a well-defined purpose. They will have to inform the user about new purposes for processing.
- Businesses will have to inform the user whether a decision is automated by an algorithm and give him/her a possibility to contest it.

5.2 Functionalities

The chatbot is reachable on the instant messaging application Telegram, even though, as will be explained later, the back-end has actually been designed to be agnostic of the messaging service. As of now, the chatbot has some basic functionalities that still make it very useful. In fact, the user has the possibility to ask for:

- The bot's functionalities, getting a brief description of what kind of questions the bot can answer to.

- A specific article of GDPR, that will be sent to the user.
- A specific clause inside an article of GDPR, that will be sent to the user.
- Some specific information included in GDPR, such as, for instance, the countries GDPR is applied in, or how the end user should opt-in for the consent.
- An explanation about a term defined in GDPR.

Moreover, the chatbot is able to:

- Suggest the user the reading of some articles related to the one that has just been sent, using an inline keyboard.
- Prompt the user if he/she wants to know the exceptions to the clause or article that has just been sent, using a custom keyboard (with 'Yes' and 'No').

Due to a lack of law-domain knowledge, we didn't want to attempt an interpretation of the law creating answers avoiding legal jargon. Thus, all the answers given in case of generic questions are citations of the articles (or clauses) regarding the matter of the question asked by the user. However, the platform is ready to be used with some user-friendly answers, written by an expert.

5.3 Stack

The stack of the chatbot is:

- An **instant messaging app** that provides some API for bots (for instance, Telegram) or a web page that makes API calls to the platform
- **NodeJS**, as web server to receive, handle, format and send the messages.
- **TypeScript**, as programming language compiled to JavaScript code to be run on NodeJS.
- **IBM Watson Assistant**, to manage the NLP part of the chatbot.
- **Redis**, as an in-memory database management system, to save the users' session IDs.
- **MongoDB**, as a database management system, where the GDPR text, F.A.Q.s and definitions are saved.
- **Docker and `docker-compose`**, to orchestrate the local services (NodeJS, Redis and MongoDB).

5.3.1 Brief stack description

Telegram is a free, open-source instant messaging app. It is multi-platform and offers multi-device synchronization. [17] One of the key features of Telegram is the support for **bots**. In fact, Telegram exposes some APIs that let developers automatically receive, process and send messages to users. We chose to use Telegram as messaging platform due to its ease of use, the community support, and to avoid the development of a true browser-based front-end. Moreover, Telegram bots offer some very useful features such as `reply_markup` [15], that allows for some special type of answers, such as *inline keyboards* and *special keyboards*:

- Inline keyboards are a set of buttons that are sent after a normal message, on which the user can click. Once the user clicked, the bot receives a *callback query* (that contains some data, such as the button the user pressed), and handles it.
- Special keyboards are keyboards that force the user to give only some default answers, such as "Yes" or "No".

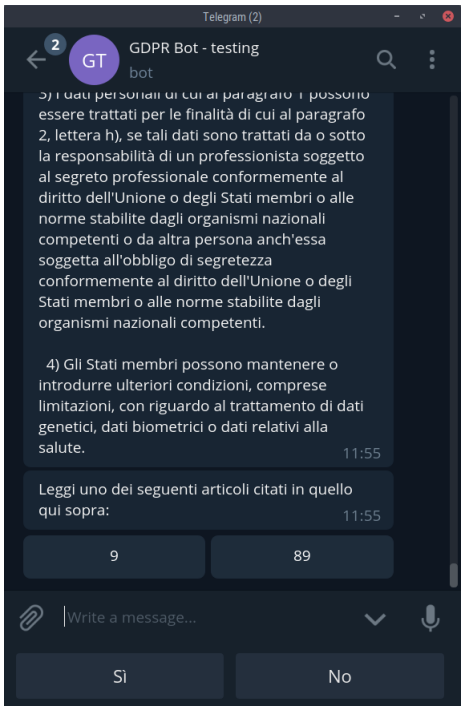


Figure 1: An example of Telegram's reply markup features.

An example of both an inline keyboard and a special keyboard can be seen in picture 1. In the lowest part there are the buttons "Sì" and "No" (respectively

"Yes" and "No" in Italian), that compose the special keyboard. Slightly above there are other buttons with some numbers. This is an inline keyboard with the articles related to the one just sent.

Node.js often simply called Node — is an open source [9] JavaScript runtime built on Chrome's V8 JavaScript engine [13]. It is, nowadays, one of the most used technologies to develop web-applications, both for front-end (with frameworks such as React or Angular) and back-end (with frameworks used to develop REST APIs, such as Express.js). It has a huge community support, that brought an impressive number of libraries, that can be installed with Node's default package manager `npm`. `npm` automatically takes care of the dependencies updates and installation. In fact, when the user installs a package into a Node project (using the `-s` or `--save` flags), the name of the package is inserted into the file `package.json`, that contains all the dependencies of the project, plus some information such as the author of the package, the version and the url of the repository. Thus, in order to run a Node application, a developer simply needs to run `npm install` (that installs all the dependencies) and `npm start` that runs the application from the entry point file. In `package.json`, a developer can also define some scripts that can be run using `npm`, to define how to compile and debug the application. As an example, here there is the `package.json` file of the chatbot project:

```
{
  "name": "gdpr-chatbot",
  "version": "0.1.0",
  "description": "A GDPR Chatbot",
  "main": "./js/index.js",
  "repository": {
    "type": "git",
    "url": ""
  },
  "scripts": {
    "tsc": "tsc",
    "start": "node ./js/index.js",
    "lint-fix": "tslint --fix 'ts/**/*.ts' 'ts/**/**/*.ts'",
    "docker-debug": "docker.compose build && docker.compose up",
    "debug": "nodemon --watch ./js --inspect=0.0.0.0:9222 --nolazy ./js/index.js",
    "postinstall": "tsc",
    "watch": "tsc -w"
  },
  "author": "Edoardo Debenedetti",
  "license": "ISC",
  "dependencies": {
    "@types/mongodb": "^3.1.19",
    "@types/node-telegram-bot-api": "^0.30.4",
    "@types/redis": "^2.8.10",
```

```
    "@types/watson-developer-cloud": "^2.40.0",
    "mongodb": "^3.1.13",
    "node-telegram-bot-api": "^0.30.0",
    "nodemon": "^1.11.0",
    "redis": "^2.8.0",
    "tslint": "^5.13.0",
    "typescript": "^3.3.3333",
    "watson-developer-cloud": "^3.18.1"
  },
  "config": {
    "unsafe-perm": true
  }
}
```

TypeScript often shortened as TS — is an open-source [12] programming language developed by Microsoft [11]. It is a super-set of JavaScript; this means that plain JavaScript is valid TypeScript code. The main difference between TypeScript and JavaScript, is that TypeScript is statically typed and offers some tools (such as IDEs plugins) that make development easier and safer. In fact, one of the biggest problems of JavaScript is that in complex projects development it is difficult to keep track of the type of variables and of the fields of the objects being used. TypeScript offers the possibility to declare the type of variables and also to define the prototypes of the objects (called *interfaces*). However, explicit type declaration is not always mandatory, as the compiler can infer the variable type from their first assignment. TypeScript is usually compiled to plain JavaScript, using the transpiler `tsc`, in order to run the code on Node or on browsers. The ECMAScript JS version TS code must be transpiled to must be declared in the `tsconfig.json` file, where the developer can also set some settings, such as the JS output folder, or the level of static typing strictness. Here there is an example of some TypeScript code, extracted from the chatbot:

```
/**
 * Splits a message in chunks to fulfil
 * Telegram maximum size requirements.
 * @param message the message to be split up
 * @param array ignore it, the function should
 * be called without this as an argument.
 * It is used as an accumulator to make the
 * function tail recursive.
 */
export function splitMessage(message: string,
    array: string[] = []): string[] {
    const telegramMaxChunk = 4096

    // Find the index of the last element of the first chunk
```

```
    const indexWhereToSplit =
        findWhereToSplit(message, telegramMaxChunk) + 1

    // Create the new, updated array
    const updatedArray = array
        .concat(message.substr(0, indexWhereToSplit))

    // Take the rest of the string, that
    // will still need to be splitted
    const stillToBeSplitted = message.substr(indexWhereToSplit)

    // If there is still some text to be splitted,
    // stop recursing and return the array as
    // it is, otherwise continue recursing
    return stillToBeSplitted === ''
        ? updatedArray
        : splitMessage(stillToBeSplitted, updatedArray)
}
```

IBM Watson Assistant is a service to create chatbots offered as part of IBM Cloud's AI suite. Via some REST APIs, Assistant can receive a message and process it. It can understand what is the user willing (called *intent*) and the variables of the user willing (called *entities*). For instance, if the user writes "I want to read the article 23 of GDPR", the intent will be the fact that the user wants to **read an article** of GDPR, while the entity is the **article 23**. Moreover, Assistant is not just a classifier. Instead, it offers the possibility to manage the conversation flow. It can directly give an answer to the user if some conditions are satisfied, or ask for more information that is missing. For example, if the user asks "I want to read an article", without specifying which one, Assistant will ask "Which one do you want to read?" waiting for the user answer. Since, for more complex answers, the bot needs to access the database, it needs to process the answer in a way too complex to be implemented directly on Assistant's platform. In these cases, Assistant is only used to recognise the intent and the entity (or the entities). Both will be sent to the Node web-server to process a proper answer.

Redis is an open source, in-memory data structure store, used as a database and cache. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams. It can run atomic operations on these types, in order to avoid synchronization issues. Redis is written in ANSI C and works in most POSIX systems like Linux, *BSD, OS X without external dependencies. Linux and OS X are the two operating systems where Redis is developed and more tested, even though it is recommended to use Linux for deploying [10]. The chatbot uses Redis as key-value store, where the key is an ID of the user, got

by the chat, and the value is the ID of the conversation on Watson Assistant. Then, the only operations used are `GET`, that gets the value, given a key, `SET`, that sets a value with a key, `DEL`, that deletes the key-value pair, given the key. It is also possible to set a time-to-live (TTL) for a data structure saved on Redis.

MongoDB is an open-source NoSQL database management system. It stores JSON-like documents, that means that different documents can have different fields, and the fact that documents can have exactly the same structure as the objects used in the bot [19]. This means that documents in Mongo can be stored according to the models defined in TypeScript's interfaces, making the communication between the chatbot and the DB extremely easy. Documents are divided into collections, that allow for an easy organization of the DB. Each document has an ID that is unique in the collection, and can be used to retrieve the document itself. It is also obviously possible to retrieve documents based on their fields, using `find` queries.

Docker and `docker-compose` Docker is an open platform for developing, shipping, and running applications. Docker enables developers to separate their applications from their infrastructure so they can deliver software quickly [7]. Docker architecture and mechanisms can be seen from figures 2 and 3. The core of Docker are *images* and *containers*. Images are models of self-contained applications and containers are instances of the images. Each container is isolated from the others and has its own dependencies, solving compatibility issues and making containers independent from each other. Nonetheless, docker containers are very different from virtual machines. In fact, a Docker container does not need its own kernel with its own operating system. Instead, it uses the host OS and its resources. The differences between containers and virtual machines can be seen in figures 4 and 5.

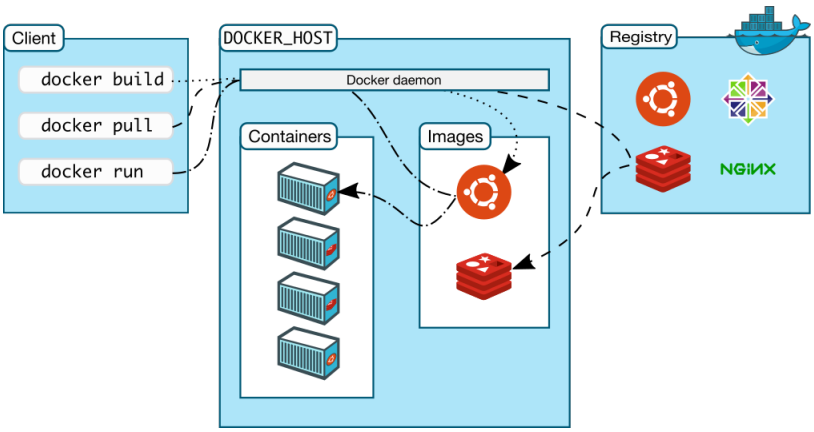


Figure 2: A diagram that represents Docker architecture, source [7]

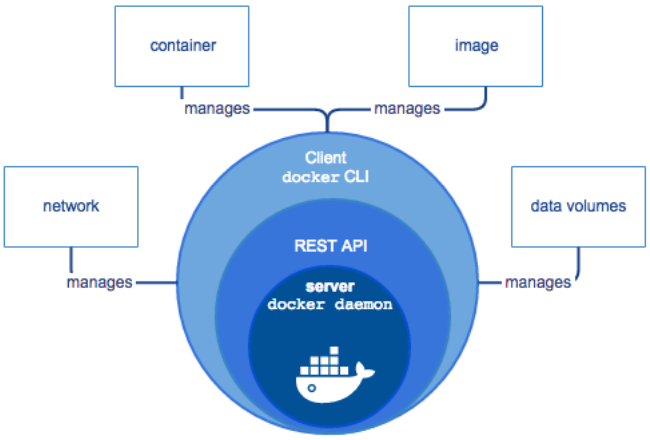


Figure 3: A diagram that represents how Docker components work together, source [7]

The main features of Docker are:

- The possibility to create an image using a file, named `Dockerfile`, in which all the features of the image are listed. In the `Dockerfile` the developer can tell from which image the new image must be created (for instance an image of a Node, a MongoDB or a Redis server), and list some characteristics such as exposed ports, storage folders to be used (and how they are mapped to the ones in the host) and what commands should be run in order to setup and start the service of the image (such as `npm install` and `npm run`).
- The possibility to create some local virtual networks in order not to publicly expose some critical services. For instance, this allows to isolate the database that, in this way is reachable only from containers running on the virtual network and not from the outside.
- The possibility to use some pre-built images (such as, for instance, that of a DBMS) that can be downloaded from a repository and directly run locally with one command.
- The possibility to define how a group of containers must be used together and to activate them with one command, using `docker-compose`.

`docker-compose` uses a file, `docker-compose.yml`, to define how a set of containers should be deployed, and how they should work together. All the containers are deployed inside a virtual network and are listed as `services`, and each container is activated with some specific characteristics:

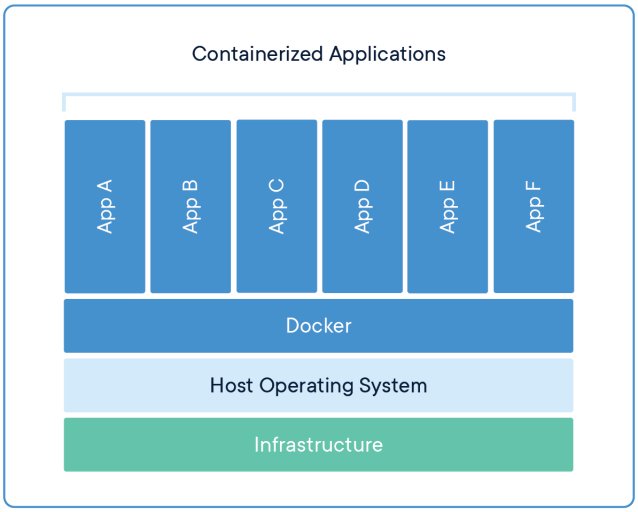


Figure 4: A diagram that shows how containers work, source [18]

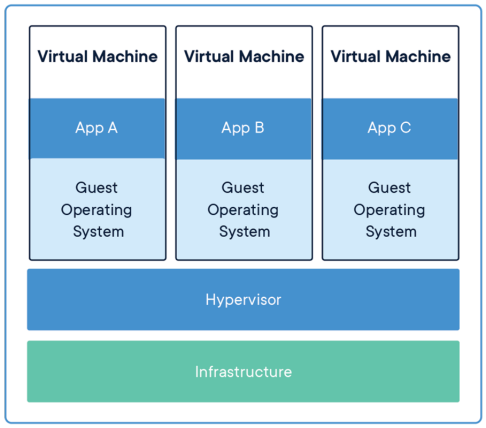


Figure 5: A diagram that shows how VMs work, in comparison to containers in figure 4, source [18]

- The image to be used (i.e. `node`, or `redis`).
- The name of the container.
- The information about the folders to be used by the container.
- The commands to be run inside the container once it is activated.
- The environment variables to be set.
- The ports to be exposed both inside the local networks and their (possible) mapping to the host machine.

This is the `docker-compose.yml` file that lets us activate the chatbot, by simply using the command `docker-compose up` (API keys have been censored for security purposes):

```
version: '3'

services:
  redis:
    image: "redis:alpine"
    container_name: "redis-container"
    volumes:
      - redis-data:/etc/redisdata/
    command: ["redis-server", "--appendonly", "yes"]

  db:
    image: "mongo"
    container_name: "mongodb-container"
    volumes:
      - mongo-data:/etc/mongodata
    ports:
      - "27017:27017"

  chatbot:
    build: .
    command: "npm run debug"
    container_name: "gdprbot"
    ports:
      - "3000:3000"
      - "9222:9222"
    volumes:
      -  /usr/src/app/js:/home/edoardo/Documenti/reply/gdpr-chatbot-sl/js
    environment:
      - BOT_TOKEN=XXXXX:XXXXXXXXXX
      - ASSISTANT_ID=XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXX
      - ASSISTANT_IAM_APIKEY=XXXXXXXXXXXXXXXXXXXXXX-XXXXXXXXXXXXXXX
```

```
      - ASSISTANT_URL=https://gateway-lon.watsonplatform.net/assistant/api

volumes:
  redis-data:
  mongo-data:
```

5.4 Architecture

It can be roughly seen in figure 6 how the chatbot infrastructure has been designed in a layered way, with the following layers:

- An interface that receives and sends the user messages, via the IM APIs (or via some exposed REST APIs, in case the chatbot user interface is implemented in a stand-alone mobile or web-application).
- A set of functions that handle the received message and format the message to be sent.
- Three independent sets of functions used to connect the platform the the three services in use: Redis, MongoDB, and Watson Assistant.

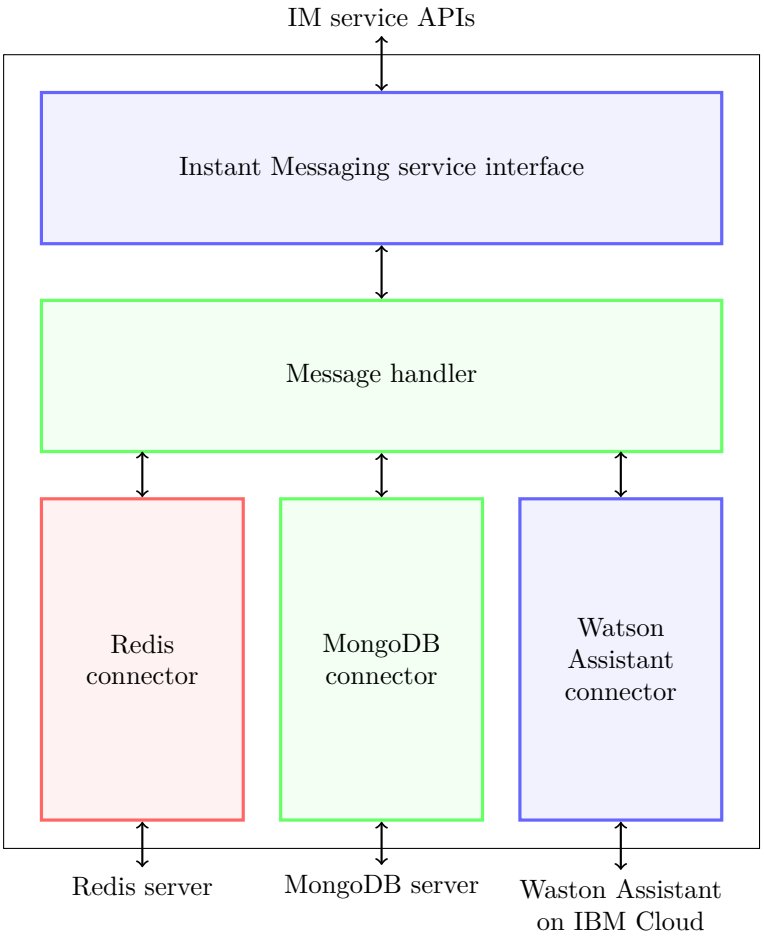
I tried to comply with functional programming principles, by preserving immutability of variables (i.e. all the variables are in fact constants, they are never reassigned) and by using typically functional iterating methods such as `map`, `reduce` and `filter`. Moreover I made massive use of lambda and higher-order functions.

5.4.1 IM Interface layer

The highest layer is the entry point into the platform, the one that connects the chatbot with the user interface, may it be an instant messaging application or a stand-alone web or mobile application. In the former case, the REST APIs of the IM application must be used. In the latter one, some REST APIs must be exposed and made reachable from a front-end.

Since Telegram is being used, we needed use to Telegram APIs. For this purpose we decided to leverage the open source package `node-telegram-bot-api` [20]. In order to use it, we can simply use Node's package manager **npm**, running `npm install --save node-telegram-bot-api`. We need to receive messages, commands, and to handle what are called *callback queries*, that are queries made when the user taps an element of an inline keyboard. Telegram APIs require an API key that identifies the bot itself. The key is passed to the program as environment variable, read via `process.env`, Node's object that lets the runtime reach environment variables. This is done in this way, in order not to hard-code the API key, that would be insecure and less practical in case of different runtime environments (such as testing and production). In fact, different deployment environments require different bot API keys for the different bots to be used.

Figure 6: A simple diagram that represents the architecture of the chatbot



The first thing we do is the bot initialization, that consists in the creation of a new `TelegramBot` object called `bot`. Thus, two methods of this object are required: `onText` and `on`. The former (used twice) is asynchronously called when receives some text that corresponds to regular expression specified as first argument, while the latter is invoked when a message of the kind passed as argument arrives. In particular, in this case, we use the method with the `'callback_query'` argument, that implies that the message is intercepted whenever the bot receives a callback query. Both the methods require a callback function that receives as argument the message received. The message can be then processed inside the callback function.

`bot.onText` usage As stated above, we use this method twice, the first time with the `^\/.+` regular expression, to take all the commands (that always begin with '`/`'), while the second time with the regular expression `^(?!(\/)).+` to filter out commands (i.e. messages that do not begin with a '`/`').

Commands are processed by the `handleCommand()` function, that returns an *hello* introductory message, in case of the '/start' command, otherwise returns an answer that warns the user that the command was not recognised. Once the answer has been created, it is sent with `TelegramBot`'s `sendMessage()` method, that takes as argument the id of the chat, retrieved from the incoming message, and the answer to be sent.

Non-command messages need to be processed at the lower level. This implies the usage of Watson Assistant, Redis and Mongo. In the lower layer, we need to retrieve (and/or save) Assistant's conversation ID from Redis. The key to be associated to the value is generated by simply appending the ID of the chat to the string 'telegram'. I did this, in this way, for scalability purposes, thinking of a way to differentiate, in the future, among the different chat providers. We then call the asynchronous function `handleMessage()`, whose behavior and implementation we will analyzed later, that returns an array of `Message` objects, that contain the answer(s) to be sent to the user plus some information about the need for a reply markup (e.g. custom or inline keyboards). Once the message is created, it is sent using the function `sendMessages()`, that is not `TelegramBot`'s `sendMessage()` method, but a function written by me to allow the usage of reply markups and the split of too long messages. `sendMessage()` implementation will be discussed later.

`bot.on` usage This method is used to handle callback queries, then it is called with `callback_query` as first argument. In a a Telegram callback query is called whenever the user clicks on an element of an inline keyboard. In particular, in this bot, the inline keyboard is used to suggest to the user the reading of the article referenced in the one just sent. Then, when the user clicks on on button, the bot must send back the required article, using the lower layer function `handleArtCallBackQuery()`.

In the callback function we first check that the query has some data. If it has not, the query is ignored. Otherwise we parse the object containing the

data, saving it in an object of type `CallBackQueryData`, that is an interface with only one field, the type of query. However, I also created an interface `ArticleCallBackQuery`, that extends `CallBackQueryData` adding a field that contains the number of the article required by the user. Then, if the callback query is of type `ArticleCallBackQuery` (which, as of now, will be always be the case), the program retrieves the article number, leveraging the lower layers, and send the required article with the `sendMessages()` function. Finally, once the answer is sent, we sent also an answer to the callback query, using `bot`'s method `answerCallbackQuery`. This will show a pop-up on the chat, telling the user that the article has been sent.

`sendMessages` function The `sendMessages` function is used to send an array of messages. It requires the array of messages and the ID of the chat messages must be sent to. It requires an array of messages, because the bot might want to send more than one message at a time. For instance it might send an article, and then ask, in a separate message, whether the user wants to know the related articles. The messages are in form of `Message` objects. The `Message` interface contains the text of the message to be sent and the (optional) array that contains the strings to be included in the (optional) custom keyboard. Briefly described, the function loops through the array and creates an `options` object in which the information about the reply markup is saved. The message is then sent with the `finalizeMessage()` function. We will analyze its behavior later. In case there are some references to the just-sent message, we take the numbers of the referenced articles and include them in an inline keyboard. Finally we send the message that recommends the user those articles.

`finalizeMessage()` function Telegram requires the messages to be less than 4096 characters long. The bot needs then to split the articles that are longer than the limit in a smart way, in order not to split the message in the middle of a sentence. This is accomplished with a function that truncates the message taking the last `'\n'` character before the 4096th one. This function returns an array of strings to be subsequently sent with `bot`'s `sendMessage()` method.

5.4.2 Message handler layer

This layer, composed by the `handleMessage()` function and a few other support functions, is the core of the application, the one that links the receiver/sender with all the services used by the bot. In this layer the message written by the user is taken and an array of messages ready to be sent is returned to the upper layer.

`handleMessage()` function This function first takes Watson's ID from Redis, using the key passed as argument to the `handleMessage()` function and queries Assistant for an analysis of the message. Once the response has been received, we have several possible cases:

- Watson's response includes an answer ready to be sent back (i.e. Watson recognises in intent whose answer is available in the conversation flow set on IBM Cloud). If this is the case, the function returns the answer.
- Watson recognises an intent in the user message.
- There is an intent coming from a previous message that still needs to be fulfilled: i.e. the user in the previous message asked for an article, without declaring which one; the bot then saves the intent into the context and then asks for the number of the article; the user answers and the bot sends the article back.

In both the last two cases the answer is handled by the `checkIntent()` function, that will be analyzed in the following paragraph. Finally, once the answer has been formatted and made ready to be sent, the function returns. If there was a pending intent to be fulfilled, or there was a 'Si' or 'No' intent (those used to let the user give affirmative or negative answers to the bot's questions), we need to clear Assistant's conversation and delete the key-value pair from Redis.

`checkIntent()` function This function is used to discern the different intents that can be returned by Watson Assistant. We can have different possible intents:

- **`ArticleContent`**, that means that the user wants to read a specific article.
- **`CommaContent`**, that means that the user wants to read a specific clause of a given article.
- **`Definition`**, that means that the user wants to read the definition of a term introduced or used in GDPR.
- **`Si` or `No`** , that means that the user wants to say yes or no to a question asked by the bot. It is in Italian, because the bot speaks and understands Italian only.
- The intent is one of the **FAQ** intents, that means that the user asked for a specific concept (i.e. "Where is GDPR applied?"). However, this has two possible sub-cases:
  - The concept in the article that contains the answer to the user has **some exceptions**.
  - The concept in the article that contains the answer to the user has **no exceptions**.
- The intent is **not valid**, there was an error somewhere, and a message of error is returned.

I will now explain how each intent is handled.

`ArticleContent` In this case, the user wants to read a given article. We then retrieve the text of the article from Mongo and return the well-formatted article, with references (if present) attached to the `Message` object to be returned, ready to be sent.

`CommaContent` In this case, the user wants to read a specific clause of a given article. We then retrieve the complete article from Mongo, extract the specific clause and return it well-formatted, with references (if present) attached to the `Message` object to be returned, ready to be sent.

`Definition` In this case, the user wants to know the definition of a term included in GDPR. The function first extracts from Mongo in which article (and clause) the definition is contained (using as key in the DB the term to be defined, that is extracted by Assistant), then it takes (always from Mongo) the text of the article (or only of the clause, in case the definition is entirely contained in the clause). Finally the text is formatted and returned inside a `Message` object, ready to be sent.

`Si or No` In this case, the user affirms or negates his will about something asked by the bot. If the user says no, the bot simply returns a message containing a positive feedback. Otherwise, the bot returns what suggested to the user in the previous message (i.e. an exception to the answer to a F.A.Q.).

F.A.Q. intents As will be seen later more in detail, each F.A.Q. has a corresponding intent. The intent is used as key to get from Mongo the article that contains the answer to the asked question. Then, the article is retrieved. If the answer is contained in just some clauses of the article, only those clauses will be formatted and returned. Otherwise, the entire article will be returned.

5.4.3 Connectors layer

This layer is composed by three connectors, that are functions used to connect to the two databases (Redis and Mongo) and to IBM Watson Assistant on IBM Cloud. Each connector is independent of the other, in a way such that the whole application in agnostic of the external service used. Each connector is a wrapper of some official drivers.

Redis connector The bot uses Redis to store the Watson Assistant conversation IDs, that are the UIDs of the conversations between Watson and the different users. We decided to use Redis due to two main reasons:

- We need to access the conversation ID very fast. The fact that Redis is an in-memory database provides the bot with a fast access to the data.
- Each couple of key-value is just composed by two short strings, that take very few memory. This won't be too heavy for the memory.

- Redis allows us to set a time to live (TTL) for the key-value pairs saved. This is very useful, because Watson Assistant will delete the conversation (and thus the IDs) 5 minutes after the last message sent by the user.

Redis server is on a local container. The NodeJS server communicates with the Redis one via the Redis driver for NodeJS, that is the npm package `redis`, and can be installed with `npm install --save redis`.

The first thing we do in the connector is the connection to the server. The connection is done with the driver function `createClient()`. The port is the well-known one for redis (`6379`) and the hostname is the one of the Redis Docker container. Then, we need to communicate with the server for three reasons:

- Get the conversation UID: this is done with the `GET` method of the client. I transformed it in a promise with `promisify()` to make asynchronous handling easier.
- Set the conversation UID: this is done with the method `SET` of the client. We set the TTL to 300s using the `'EX', 300` arguments.
- Delete the conversation UID: this is done using the `DEL` method of the client.

MongoDB connector MongoDB is used to store all the information about GDPR, such as FAQs and GDPR itself. The MongoDB server is on a local Docker container. On Mongo there are 3 collections:

- **`articles`**, that contains all the articles of GDPR. Each article is a JSON that contains its title, its number, and an array of objects that contain the clauses. Each clause contains an array with the sub-clauses and with the articles referenced in the clause. The id of each document is the number of the article.
- **`definitions`**, that contains all the data about the definitions of words in GDPR. Each document contains the number of the article and the specific (if any) clause(s) where the word is defined. The id of each document was given by me and is the same as the name of the entity defined on Assistant for the specific definition.
- **`faq`**, that contains all the data about FAQs. Each document contains the fields contained in the definitions, plus the name of the FAQ, a Boolean to know whether the answer has some exceptions or not, and (if this is the case) the id of the Mongo document that contains the data about the exception.

The documents are retrieved with the `getArticle()`, `getDefinition()` and `getFaq()` functions. Each kind of document is saved according to the interfaces defined by me.

Watson Assistant connector Watson Assistant is used for the NLP part of the bot. It is hosted and served by IBM Cloud. We need to communicate with Watson for three reasons:

- Query Assistant for the intents and entities that are in the user's message.
- Delete a conversation.
- Create a conversation and get its ID.

We first obviously need to initialize the bot with an API key (taken from the environment variables, as for the Telegram API key), and the date for the last time the API were used (for compatibility purposes). Everything is done with the official IBM Watson Developer Cloud driver, that can be installed with `npm install --save watson-developer-cloud`.

Assistant query In order to query Assistant we use the `askAssistant()` function. It sets all the required parameters in the payload that will be sent (e.g. the message to be processed). Then it sends the request with the function `messagePromise()`. The `messagePromise()` function wraps the driver's `message()` method, creating a `Promise` from it. It finally returns Assistant's response along with the `sessionId` (that might have just been created). Assistant's response conforms to the `WatsonResponse` interface.

Context deletion After the bot fulfills a user request, it has to clear the conversation context, to avoid any kind of collateral effect due to previous context variables. It is done via the `clearContext()` function, that wraps the driver's `deleteSession()` method.

Conversation creation In order to query Watson with a message to be processed, we first need to have started a conversation, and we need the ID for the conversation. The conversation is created with the `getId()` function, that wraps driver's `createSession()` creating a `Promise` from it, to make handling easier.

6 Conclusion

This internship has been a unique opportunity to improve the skills I already had and to learn several new ones. In fact, I learned how to develop a chatbot using IBM Watson Assistant and Telegram, how to automate business processes using RPA, what Docker is and how to use it. I also learned some system administration skills (mainly on Linux OSs, such as CentOS), managing virtual machines on the hyper-visor server we had in the office. I improved my Node.js, Python and TypeScript skills, as well as my team working and architecture design abilities, also thanks to the autonomy I was given in all my activities. Working on real projects and proofs of concept that one day could be sold to Blue Reply's customers, or expanded by some other Blue Reply's employees, made me feel responsible for what I was doing and was truly educational. In fact, I was forced to design a smart architecture and write clean, meaningful and documented code, having it reviewed by my colleagues. Going to the office daily, following my colleagues' routines also let me have a grasp of what working life is.

Acknowledgments

I would like to thank: Blue Reply, that gave me the opportunity to intern there; my company supervisor, Blue Reply Cognitive business unit's manager Oscar Pistamiglio, who decided to take me in his BU, assigned me interesting projects to work on and gave me autonomy; last, but not least, all my business unit's colleagues, in particular Marco Schiapparelli, who supported me giving me advises and guiding me in the projects' implementation.

References

[1] Thomas H. Davenport. "Process Automation and the Rebirth of Reengineering". In: *The Wall Street Journal* (July 2015). [Online; Last visited April 10, 2019]. URL: https://blogs.wsj.com/cio/2015/07/08/process-automation-and-the-rebirth-of-reengineering/.

[2] Rachael King. "AT&T Employees Automate Repetitive Tasks with Software Bots". In: *The Wall Street Journal* (May 2016). [Online; Last visited April 10, 2019]. URL: https://blogs.wsj.com/cio/2016/05/15/att-employees-automate-repetitive-tasks-with-software-bots/.

[3] European Parliament and Council of the European Union. "General Data Protection Regulation". In: *L119, 4 May 2016, p. 1–88* (Apr. 2016). URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1528874672298&uri=CELEX%5C%3A32016R0679.

[4] European Commission. *2018 reform of EU data protection rules*. [Online; Last visited April 10, 2019]. URL: https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules_en.

[5] European Commission. *A new era for data protection in the EU — What changes after May 2018*. [Online; Last visited April 10, 2019]. URL: https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf.

[6] Selenium Contributors. *Selenium, browser automation*. [Online; Last visited April 10, 2019]. URL: https://www.seleniumhq.org/.

[7] *Docker overview*. [Online; Last visited April 10, 2019]. URL: https://docs.docker.com/engine/docker-overview/.

[8] Node Foundation. *Express.js homepage*. [Online; Last visited April 10, 2019]. URL: http://expressjs.com/.

[9] Node Foundation. *Node.js JavaScript runtime*. [Online; Last visited April 10, 2019]. URL: https://github.com/nodejs/node.

[10] *Introduction to Redis*. [Online; Last visited April 10, 2019]. URL: https://redis.io/topics/introduction.

[11] Microsoft. *TypeScript homepage*. [Online; Last visited April 10, 2019]. URL: https://www.typescriptlang.org/index.html.

[12] Microsoft. *TypeScript - JavaScript that scales*. [Online; Last visited April 10, 2019]. URL: https://github.com/Microsoft/TypeScript.

[13] *NodeJS about page*. [Online; Last visited April 10, 2019]. URL: https://nodejs.org/en/about/.

[14] OakwoodAI. *Automagica, Open Source (Smart) Robotic Process Automation*. [Online; Last visited April 10, 2019]. URL: https://github.com/OakwoodAI/Automagica.

[15] *ReplyKeyboardMarkup API reference*. [Online; Last visited April 10, 2019]. URL: https://core.telegram.org/bots/api/#replykeyboardmarkup.

[16] Al Sweigart. *PyAutoGUI, a cross-platform GUI automation Python module for human beings*. [Online; Last visited April 10, 2019]. URL: https://github.com/asweigart/pyautogui.

[17] *Telegram F.A.Q.* [Online; Last visited April 10, 2019]. URL: https://telegram.org/faq.

[18] *What is a container?* [Online; Last visited April 10, 2019]. URL: https://www.docker.com/resources/what-container.

[19] *What is MongoDB?* [Online; Last visited April 10, 2019]. URL: https://www.mongodb.com/what-is-mongodb.

[20] yagop. *Node.js Telegram Bot API*. [Online; Last visited April 10, 2019]. URL: https://github.com/yagop/node-telegram-bot-api.