Universidad Zaragoza
1542

Trabajo Fin de Máster

Microservicio basado en FHIR: Bot eHealth para el seguimiento de la Medicación de Pacientes

FHIR Compliant Microservice: an eHealth bot for Patient-Caregiver Medication Management

Autor

John Lawrence Rogers V

Directores

Surya Roca Mainer, José García Moros

Escuela de Ingeniería y Arquitectura / Universidad de Zaragoza

2018

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. John Lawrence Rogers V,

con nº de DNI Y4954774J en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Máster, (Título del Trabajo) Microservicio basado en FHIR: Bot eHealth para el seguimiento de la Medicación de Pacientes,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 20 de Septiembre 2018

Fdo: John Lawrence Rogers V

Acknowledgements

I would like to thank first and foremost the co-director and my tutor Surya Roca Mainer, for whom without none of this would have been possible. Surya was a source of inspiration for me throughout the difficulties in the project and kept my sights focused when they were unclear. Together with all the guidance she provided in person and the countless emails sent back and forth she ceaselessly aided me where and whenever it was needed. For everything you have done, I thank you.

I also give thanks to my director José García Moros, for his support and guidance. Most of all his patience as this project spanned countries and continents in its writing.

To my colleagues and friends in the laboratory thank you for being as friendly, inviting and supportive as you all were. It has been a pleasure to meet all of you and thank you for your words of encouragement and aid.

Lastly to my family: My wife Rosa, for putting up with the hours spent in the lab and at home throughout this project. And to my mother and father for being there for me during my years abroad

Abstract

Debido al aumento en la población de personas mayores y de los riesgos médicos asociados a ello, junto con el incremento del uso de dispositivos móviles, surge una gran oportunidad de proporcionar ayuda instantánea de manera fácil y económica a esta comunidad vulnerable. Más allá de los ancianos, sucede la misma situación para las personas sin muchos recursos y para la población que vive en zonas remotas del mundo; los dispositivos móviles están en todas partes y pueden ser utilizados por todo el mundo. Con el fin de aprovechar el poder de este gran recurso, este proyecto tiene como objetivo construir una plataforma de mensajería eHealth para enviar información médica relacionada con medicamentos, notificaciones fiables y administración de medicamentos a estas personas y a sus cuidadores. Para llevar esto a cabo, se ha realizado una revisión exhaustiva y se ha migrado la especificación de medicación local a un estándar internacionalmente reconocido. Se ha diseñado un chatbot para interactuar y ayudar a los pacientes y, al mismo tiempo que se ha realizado la expansión de una librería para mejorar la implementación de este chatbot y futuros chatbots en Python, se ha investigado y desarrollado un novedoso protocolo para gestionar chatbots multilingües. Finalmente, se ha realizado un análisis inicial para la detección de medicamentos basado en el reconocimiento óptico de caracteres y se han sentado las bases para el desarrollo futuro.

Abstract

With the ever increasing elderly population and the medical risks associated with them, as well as the exponential rise in their smartphone connectivity, there is a great opportunity to easily and cheaply provide instant aid to this vulnerable community. Beyond elderly the same is true for the impoverished and those living in remote regions of the world; smartphones are everywhere and entering everyone's hands.

In order to harness the power of this great resource this project aims to build onto the eHealth smartphone messaging platform to provide medical information on medications, reliable notifications, and medication administration reporting to these people and their caregivers. To do this a comprehensive review and migration of the local medication specification to an internationally recognized standard was undertaken. A chatbot was designed to interact with and aid patients. While a library was expanded for the better implementation of this chatbot and future chatbots in Python. A novel protocol for handling multilingual chatbots was investigated and developed. Lastly initial looks into optical character reader based medication recognition was performed and the foundation for future development was laid.

Contents

List of Figures

Chapter 1

Introduction

Telemonitoring of patients plays an important role in aiding the elderly, people with limited mobility, and people from remote regions where primary care facilities are limited. Of all the keys to telemonitoring the control of medications stands out. In the elderly and people with limited mobility, mistakes in the administration of medications is one of the leading causes of hospitalization, accounting for over 6% of hospitalizations (*1*). This highly avoidable problem is associated with patients lacking proper information, memory related issues, and caregiver mistakes. Often a patient receives a medication and is not aware of the proper dosage or even the appearance of the medication.

In patients with five or more daily medications it can become quite confusing as to which medication is used when and how. Memory can affect whether or not the patient recalls having taken a medication. In the case that they forget to take it, a missed dose is the result. For patients receiving antibiotics for an infection or blood thinners for heart problems that can be very problematic. The opposite case is where a patient forgets that they have taken a medication and repeats the administration excessively. One of the most common medication related issues is a result of that case. With the common Non-steroidal anti-inflammatory drugs (NSAID) this can often lead to gastro-intestinal bleeding(*2*).

This project proposes and goes through the design of two core products: An eHealth messenger service for the recording, monitoring, and provision of medication information paired with an internationally recognized standard of medical data storage and organization to the Spanish medication and doctor database. This is to be accomplished by integrating into a system of tools that compose a *patient - caregiver - chatbot* platform. The tools known as microservices are each fundamental programs designed to perform one aspect of the solution. They can work independently yet interact with each other fully for combined needs.

1.1 Project Overview

The eHealth messenger that this project focuses on, from this point to be referred to as the Medication microservice, is a tool to be implemented into the University of Zaragoza's Signal based eHealth messenger solution. It is a tool to interact with healthcare providers, patients and caregivers. It is to facilitate the interaction between them by keeping a record of medications, ensuring that medications are received, taken, or administered properly. The Medication microservice does this through an intuitive chat, a powerful notifications manager, and a complete medication information storage system. A quick overview of the Medication microservice is shown below in Figure 1.1

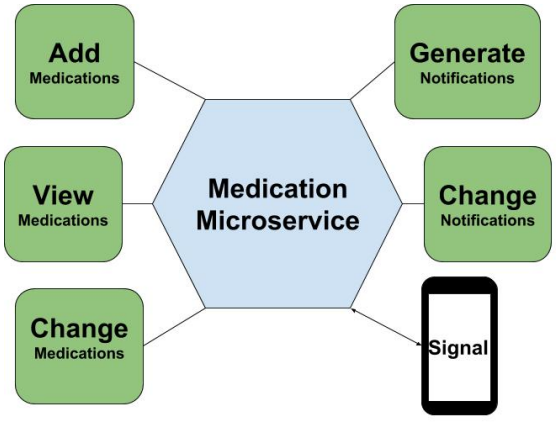

Figure 1.1: Overview of the microservice detailing the various functions.

Above it can be seen that when a patient receives their prescription sheets containing the codes from the *Agencia Española de Medicamentos y Productos Sanitarios Centro de Informacion de Medicamentos* (AEMPS CIMA) codes, they have only to input the medication codes to make a record of having received them. Once all of the codes are entered, and verified for correctness, the record is stored on our secure, internationally standardized server. The doctor or caregiver can at any time (if permitted by access rights) access the records to make sure that the correct medications are listed, what the dosages, schedule etc are. The microservice will automatically generate notifications if the user wishes. These can of course be customized or entered completely manually if desired. The notifications utility also makes a record of taken or missed medications and the time taken. It can also be customized for any other relevant steps.

The final piece of the puzzle is in the exchange and storage of the medication data. Specifications for the exchange of medical information are of utmost importance when one begins to think about how crucial they are to the safety, health and integrity of a medical solution. To do this the project will add an additional standard to the Spanish medication network. The *Fast Healthcare Interoperability Resources*, FHIR specification is one such method that prides itself in interoperability, its open nature, and ubiquity in the medical world and was chosen for this task.

1.2 Objectives

This project began with three main objectives, to develop a microservice to aid in the monitoring, administration, and record keeping of medications; to apply the FHIR specification to the Spanish AEMPS CIMA database, and to create a Python basis in which further AIML chatbot eHealth microservices can be implemented.

This microservice must be able to integrate into the existing microservice bot

framework. It must follow the same user experience format designed by the University of Zaragoza team. It should also provide the ability for expansion into a more *Natural Language Processing* structure than is currently implemented. It must be fully FHIR compliant and designed to be easily accessed and used by other microservices. The AIML Python library must be expanded to conform with the AIML 2.0 guidelines as far as is necessary for the chatbot and liberties may be taken for the expansion of added functionalities beyond the scope of the AIML 2.0 guidelines.

1.3 Materials and Technologies Used

In order to pursue this project many technologies, tools and resources were used.

- **Python 3.6:** This newer version (versus Python 2) was chosen because the latter version is being progressively out phased, thought the earlier one does have a more complete AIML library, the author found it prudent to use this newer version and expand its AIML library. This was the primary language used for the development of the chatbot and supporting functions.

- **AIML:** The basis for the AIML implementation in this project is pyAIML authored by Cort Stratton(*3*). This is the language used to construct the conversation flow for the chatbot. All the menus and operations are defined through AIML script.

- **Signal:** A secure messaging app that with end to end encryption. It is licensed under a free software license both the client and server (*4*)(*5*). A custom Signal software is installed on the users phone so that the user can send messages to the eHealth microservices.

- **Android OS:** Marshmallow Running on a Samsung Galaxy S5 Mini. One of the phone/OS combinations used in testing.
- **AppleOS:** Running on an Apple Iphone 7, another phone/OS combination used for testing on Apple devices.
- **Git:**Version control was maintained on a git server, GitLab via the University of Zaragoza.
- **Flask:** BSD licensed framework for small server applications. Flask provided the framework for the final server, and a sandbox for testing early implementations.
- **Gunicorn:** Licensed under MIT license, a pre-fork version of the Ruby Unicorn server. This is used to spawn workers to run the aforementioned Flask application and to serve the Flask application at a production level.
- **Docker:** Container platform for deployment of servers. Software used to aid in the deployment and orchestration of the microservices.
- **Ubuntu 16.4:** A Debian based UNIX operating system, in which the microservices were installed on.

1.4 Report Organization

This report is structured in the following manner:

- Chapter 1 consists of an introduction, where the background for the project, the objective, and materials are all detailed
- Chapter 2 gives a more detailed view of the state of the art in the field. Looking into the different standards for data organization, the question of security, and the task of migrating the Spanish medication information standard.

- Chapter 3 describes the development of the Bot, the expansion of the AIML library, and early development of further functions of the microservice.
- Chapter 4 goes deeper into the expansion of the chatbot, how multilingual support was handled and how the greater functionality of the chatbot was achieved.
- Chapter 5 steps through the results of the previous sections and gives examples of the microservice in action.
- Chapter 6 discusses the final implementation and future expansion of the project. It goes on to work out the successes and failures, and finishes with results and conclusions made.
- Appendix A is a comparison of different popular messaging clients used in the decision making process.
- Appendix B shows the distribution and allotment of time for the project along with a Gantt chart of the project timeline.
- Appendix C has the configuration file for the Docker build as well as a list of the package requirements.
- Appendix D lists the extensions added to the pyAIML library, along with short descriptions of each.

Chapter 2

Applications for Medical Data

The first stage in the designing the microservice was to define the capability in terms of medical data. A simple to use, well designed, and robust system for the data is discussed in the following sections.

2.1 Data organization Standard

The FHIR structure was designed in an iterative and review method based off of previous HL7 standards (*6*). This was to ensure that follies of previous standards were not implemented again. In the successors to FHIR one of the major problems was the massive amount of redundant and complex resources. FHIR's aim is to reduce the overall number of resources, while decreasing complexity and increasing interoperability. They accomplish this through the following guidelines, quoted from Bender et al. who helped develop the FHIR standard (*7*).

- *Resources should have a clear boundary, that matches one or more logical transaction scopes.*
- *Resources should differ from each other in meaning, not just in usage (e.g., different ways to use a lab report should not result in different resources).*

- *Resources need to have a natural identity.*
- *Resources should be very common and used in many different business transactions.*
- *Resources should not be specific or detailed enough to preclude support for a wide range of business transactions.*
- *Resources should be mutually exclusive.*
- *Resources should use other resources, but they should be more than just compositions of other resources; each resource should introduce a novel content.*
- *Resources should be organized into a logical framework based on the commonality of the resource and what it links to (see resource framework below).*
- *Resources should be large enough to provide meaningful context; resources that contain only a few attributes are likely too small to provide meaningful business value.*

All of the above has greatly reduced the complexity of the HL7 standard from the previous version *v3* to FHIR where now each resource has a broader scope, but because of the care in implementation is rigid enough to maintain intuitive consistency across uses. To illustrate this the principle example from this project will be used below. The situation is a series of medications from a prescription sheet is to be made.

"Doctor Smith prescribes his patient Jane Doe 10 different medications, he also wants reports of when they were ordered, received, and taken."

Based off of the third tenant of the standard *"each should have a natural identity"* it is clear from this case that the following resources at least will be needed:

- Person, to express the identity of each the patient and doctor.

- Patient, to give the patient Person its appropriate role.
- Practitioner, to give the doctor Person its appropriate role.
- Medication, to represent each medication as a Medication only.
- Medication Statement, to represent that a Medication has been prescribed / ordered etc.
- Medication Administered record of a medication taken or not taken.

This is convenient that a Person resource can be re-used as Doctor, nurse, caregiver, family member, patient or any other role. To define their role they are then referenced by a Patient instance of a resource. That way a Person can exist as a Patient as well as a Caregiver to another Person without having to be made a duplicate entry.

From there it is a matter of finding the utilitarian resources like List, fhirReference, fhirDateTime, Ingredient etc. that are used to make sure that every piece of data is stored in a clear and interoperable manner. These resources are not difficult to find as the definition for Medication details quite clearly that it is dependant on a fhirRefence to an Ingredient resource.

It may seem excessive to need so many separate resources referring to each other for a single prescription sheet, but if one were to consider the scenario where a patient gets a new prescription sheet from a doctor every month, with mostly the same medications the amount of additional resources needed greatly decreases.

2.2 Security

All successful medical applications lie behind a complete security solution. This platform is no different and traffic is sent over HTTPS through the use of JSON Web Tokens along with end to end encryption of messages to ensure there are no security breaches.

The security process begins with the distribution of key - secret pairs. The key and secret were generated using OpenSSL. This generates an RSA public and private key which is used to decrypt the token and JSON encoded message. This token is placed in the header of every interaction between the microservice and the microservice hub known as *Dispatcher*, this key microservice is described in detail in subsection 3.2.1. In quick summary it is the link between all microservices.

When the microservice receives a POST message the first thing it does is to extract the token from the header and pass it to the JSON web token authentication. This then compares it against a validation configuration. The token is successfully decoded then the message is allowed through to be parsed for use in the microservice.

2.3 Migration of CIMA database and adding FHIR Compliance

This platform has chosen FHIR as the optimum method for organizing, and relating data items in its servers. This resource is very powerful and strives towards the interoperability of all medical records, however as of yet the local health service, Aragon Salud which is a part of the greater health network of Spain, does not use FHIR for its medication records. In use is the system CIMA, which is a fully capable method of coding and referencing medications, but it does not have the interoperability nor ubiquity that FHIR provides. It was then decided to develop a tool to automatically translate the CIMA records to FHIR compliance.

The primary task in designing the CIMA to FHIR translator began with recognizing which data was relevant to our study and how it should be referenced and stored within the FHIR system. Below in Figure 2.1, is the structure of a typical CIMA medication entry.

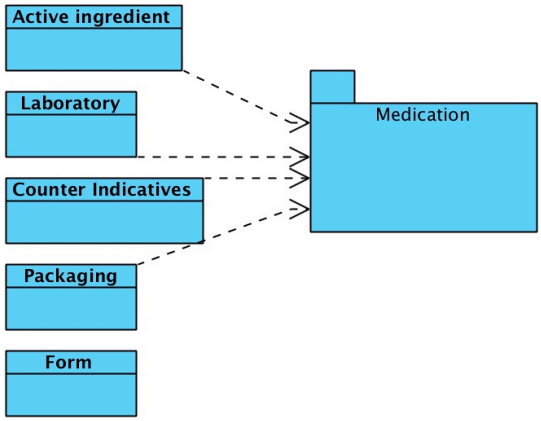

Figure 2.1: Representation of the components of a CIMA medication entry.

As shown above the database also provides several dictionaries of locations, substances, counter indicatives, the form of the pills as well as others not shown which are all linked to the medication entry through references in the main medication object. The FHIR databases model is similarly laid out except that it is much more reliant on a larger network of references. The advantage to this is that there is much lower chance of duplicate entries. The equivalent Medication object is shown below in Figure 2.2 by the red classes and modules, to show its higher dependence on relations.

A case where duplicate entries would occur in the CIMA method but not in the FHIR standard is with laboratories. A laboratory could very well be at the same time a laboratory, a clinical site, and a research facility. The FHIR method allows the laboratory to exist and then to be referenced into different roles.

Below is shown the translation from the equivalent FHIR compliant object. As can be seen much of the information is handled similarly but in the FHIR model there is higher granularity in the objects. That is instead of the medication having separate fields for active ingredients, the laboratory for that active ingredients and the dosage etc, it instead only wants an ingredients reference object. This contains a number of active ingredient objects, which each in turn have the two properties referenced: substance (ingredients) and organization (laboratory).

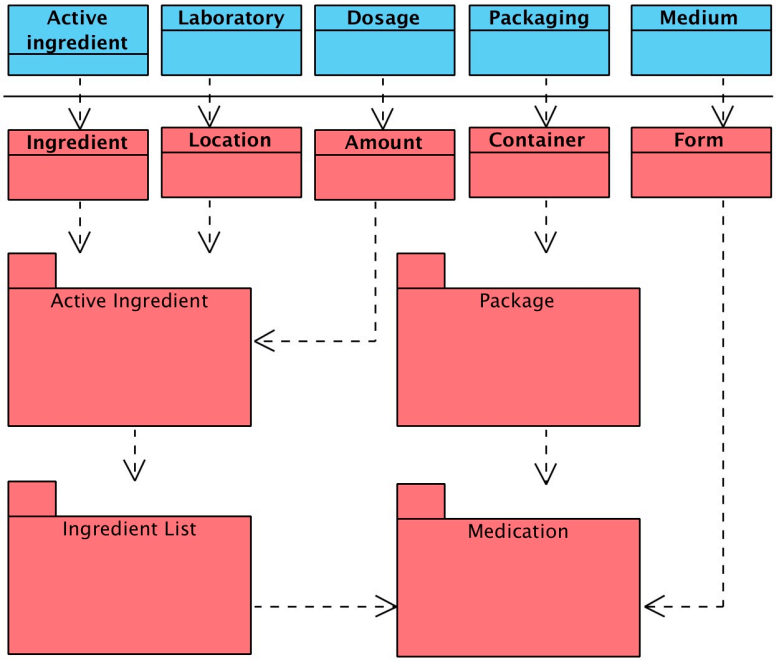

Figure 2.2: The translation from CIMA medication object in blue to the FHIR medication object and it's dependencies in red.

Finally below is shown the FHIR solution using FHIR objects. The right hand side of the map shows the medication. The Patient is shown in green and is referenced into the list of medication requests and the medication request itself. The request intern references the medication object. The cross referencing ensures that any relevant record can be called and an up to date record of the patient, or medication contents will be found.

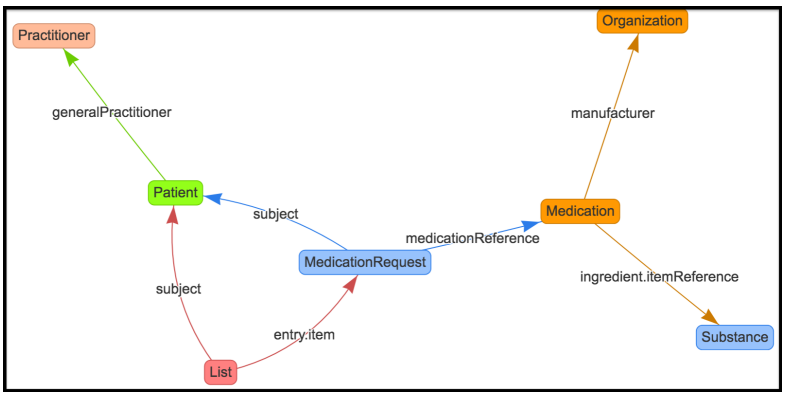

Figure 2.3: The full FHIR solution, from practitioner and patient to medication ingredient.

Chapter 3

Medication Microservice

The AIML chatbot is based on the ALICE chatbot developed by Richard Wallace (*8*). It is a XML style document where the tags define the structure of conversation. The bot is initiated by parsing the AIML document, the kernel then computes all of the recursion of the conversation. An example of this is shown below where the script for a simplified version of the starting menu is shown.

```
<category>
   <pattern>RECETAS</pattern>
   <template>
      <srai>MAIN MENU</srai>
   </template>
</category>

<category>
   <pattern>MAIN MENU</pattern>
   <template><br/>
      1. Añadir medicamentos a traves de su código <br/>
      2. Añadir medicamentos con una foto de la hoja de tratamiento<br/>
      3. Ver medicamentos actuales<br/>
      4. Ver historial de medicamentos<br/>
```

```
      5. Modificar un medicamento<br/>
      6. Añadir notificaciones para tomar medicamentos<br/>
      7. Modificar algunas notificaciones<br/>
      8. Ver notificaciones<br/>
   </template>
</category>
```

This code above shows the basic structure of an AIML script. It also exemplifies the standardized menu that all microservices use. The fundamental tag in the AIML document is *category*. This tag encompasses each unit of conversation. The particular *category* is called when the *pattern* is matched. In the above example the keyword to initiate the Spanish menu is sent "RECETAS" and the corresponding category is called. Within the *template* of the *category* is the *srai* tag which is explained in greater detail in subsection 4.1.1 called the *srai* tag. In summary it references another *category* which brings up the whole Spanish translated main menu. The apparent redundancy but strong benefits of this method will be discussed in section 4.1.

3.1 Selection of Project Building Blocks

Throughout this project many tools, programs, and libraries were investigated. Some were found more suitable than others, the following is a brief overview of the selection process for some of the more fundamental pieces of the project, a complete list of the chosen materials is found in the Materials and Technologies used in section 1.3. These are the medical information standard, the messaging client, and the *microserver* serving utilities.

3.1.1 Messaging Client Language

This project has the great advantage in that a message based has already been prototyped by a group at the University of Zaragoza. This bot uses the platform Signal as its messenger client (*5*). Signal is a messenger platform similar to Telegram, Whatsapp, Skype etc. that was chosen by the University of Zaragoza team because of its GPL license(*9*), its high level of security and especially because its code is open and available for the distribution of private servers. A full list and comparison of Signal with similar messenger platforms is shown in Appendix A.

A hurdle for this project was that the team at Zaragoza wrote the bulk of the bot in Java but desired that further work be also implemented in Python. This is due to the recent increase in popularity of Python as a leading coding language. Introduced in 1991, by 2013 it was the third most popular language for OpenSource projects, only behind Java and C (*10*). And by now in 2018 it has the largest amount of programming related traffic on *Stackoverflow*, a popular question and answer board (*11*). Python's enormous following also means that it is quite well documented although its relative youth does mean that there are some holes in its available libraries. This became a particular issue for the project in programming of the bot.

The chatbot as mentioned earlier is based on the *Artificial Intelligence Markup Language*, AIML. The best available AIML library for Python was only written to comply with the AIML 1.0 standard not the current standard of AIML 2.0 this is because the 2.0 standard, though superior is still in BETA phase and has been for some time. Therefore, in order to be able to fully write the bot and all of its functions the task of expanding the existing Python AIML library to meet most of the AIML 2.0 guidelines was also pursued by this project.

3.1.2 Platform Serving

The platform is based on a series of microservices each dedicated to a specific function. Below, Figure 3.1 shows the existing structure with the interaction between microservices. The medication microservice is seen to fit in with an entire array of microservices. The key microservice for communication is *dispatcher* which handles all inter-communications and communication to the Signal app on the phone.

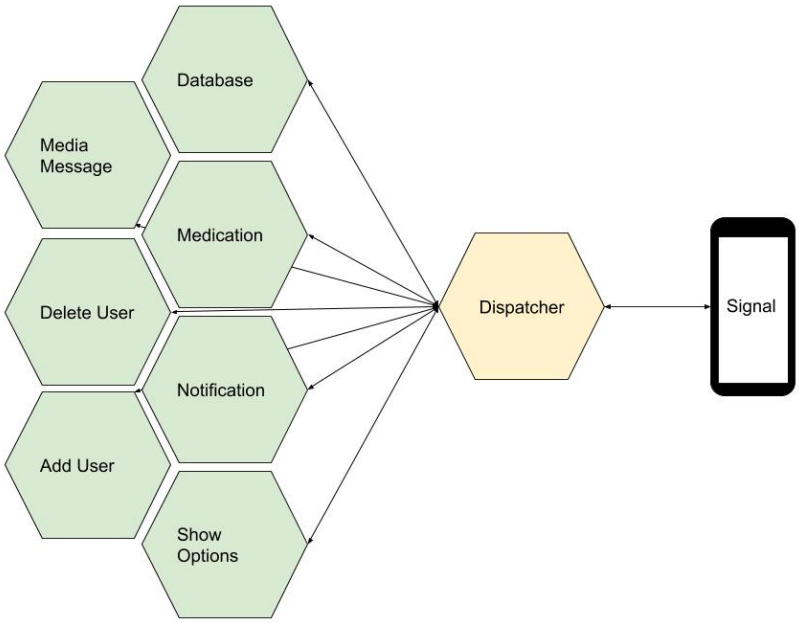

Figure 3.1: Detailed view of the full eHealth solution from phone to individual microservices.

A microservice is an independent program that runs inside of a virtual environment that is part of a bigger solution or service. These virtual environments provide a clean slate to make sure that all program dependencies are met without introducing compatibility errors with other microservices, or if one microservice encounters an error or hangs it does not disrupt the workings of the greater eHealth solution. It is always critical that an eHealth solution is maintained to be working smoothly and cannot be brought to a stop by run-time errors. This method also makes it adaptable to memory and processor allocation. These can be easily mon-

itored and controlled so that easy or passive microservices are not monopolizing valuable resources. This allows more memory or processor hungry tasks to always have those resources available. These needs can later be adjusted as seen fit so that as a microservice expands its scope it will never outgrow its virtual environment. A final reason that these virtual environments are so valuable is that they are portable. They ship as a self contained image that can be built and loaded into a new server in a matter of minutes.

Many solutions exist for these virtual environments: Apache Mesos (*12*) and Docker (*13*) being the industry leaders. This project uses the Docker solution as it was seen to be better suited for the modular sort of expansion that these microservices will experience. A comparison of the features is shown in Appendix A.

3.2 Other microservices

Various microservices have already been implemented by the University of Zaragoza team, the following are highlighted to describe so that their role in the design of the new microservice.

3.2.1 *Dispatcher* microservice

The anchor of the platform is the *dispatcher* microservice. This microservice is the brain that handles all intercommunication between other microservices, communicates with the database for user records and relates that information to the individual microservices. A typical message to *dispatcher* is sent over HTTPS and formatted as follows:

```
{"user":"+11222333345","message":
{"timestamp":1519907489750,"attachments":null,"body":"Lorem lipsum"}
```

Where the message is contained in the body section and if any pictures or

multimedia are sent in the attachments. The user identifier is the phone number including the country code.

Notifications microservice

Another useful microservice for monitoring the intake of medications that works well with the microservice here being developed, is the Notifications microservice. This was developed in parallel with the Medications microservice so their parameters were influenced and shaped by the needs of the other. The notification microservice receives the notification settings and is able to schedule messages to be sent to the patient, caregiver or medical professional.

The microservice also has the ability to initialize another microservice so that the other can be ready to receive the confirmation that the notification message was received and acted upon, or in the case of that it is not a record can be kept of a missed notification and appropriate action can be judged by the dependent microservice.

In the case of the Medications microservice that is to remind a patient to take the medication and request that the patient sends a confirmation that the correct medications have been taken. If not the Medication microservice can decide, based on user settings, if a new reminder should be made, a caregiver should be contacted or in certain cases that the single reminder was sufficient.

3.2.2 Database Requests

A final microservice that the Medications microservice communicates with is Database Request microservice. There are two main databases in the platform. One is the FHIR database which is detailed in section 2.3. The other database is used for more basic user information, such as the available microservices to a user, the user's role, name, etc. that are all important information for the microservices. This microservice is always listening for POST messages to request, update, and

create user information. The message to request a user is formatted as in the code below which is POST-ed to the server URL.

```
{"url":"users/_find?criteria={\\"id_signal\\":\\"' + sessionID +
    '\\"}","data":null}'
```

3.3 Function

Once the bot is defined in AIML it still needs to be instantiated, tell the main server that it is listening, kept running, maintain separate profiles for all users connected, and also respond to messages. The instantiation begins with the execution of the bot. To announce that it is running the bot sends *dispatcher* the following POST message:

```
{"client_id":"'Medicamentos'","message":"'recetas'",
"menu":true,"address":"'+self_url+'","canUse":["all"]}
```

Which tells *dispatcher* that it is running successfully, that it is available for use by users, and what command is needed to begin a conversation.

These portions are all handled in the Python portion of the bot. In the structure of this project this file was called *app.py*. The first task of the app is to contact the security server. After the token is created and all the security passes have occurred the AIML kernel is called to learn from the AIML document all the rules for the conversation. To do this an initial call is made for the kernel to learn the some basic commands and where to find the full AIML documents. These newly learned commands are then used to call the kernel to learn the full conversation.

In earlier microservices temporary user data had to be stored on a separate MongoDB database. The Python bot however does not need this. It is able to store, in local memory a conversation for each user currently connected with the

bot brain. Each user is an object containing information relating to the current conversation topic, the last *that* tag, and a list of the last several messages that the user sent. With these pieces of information the entire state of the conversation can be held in temporary memory without the need for a static database. A benefit of this is to avoid potential security breaches associated with storage.

The user identifier for each conversation is the users unique phone number that was used in the setup process of Signal. Every message that pertains to a user is headed with encryption, the token, and the phone number, so that it is always certain that the right conversation elements are sent to the right user, stored in the write database entry, and that to help prevent interception or man-in-the-middle type interference.

3.4 Flask application

The program on its own is able to function on its own for basic usage but for communication with the other microservices and simultaneous communication with multiple users it must be instantiated, served, and hosted properly. The server chosen for this task is FLASK a self proclaimed "micro server"(*14*). This framework on its own is well suited for simple server applications and is not weighed down by heavy development needs.

Next the Flask app that hosts request-response client is initiated. The primary task of this app is the request-response client which is always listening for POST signals to its server's address:Port. Once *dispatcher* is aware that the app is running and a user tells *dispatcher* that it wants to access it, *dispatcher* sends a POST to the app with a message containing the key word to start the conversation. This particular app was built with multi-language support, which will be further detailed in the Multi-language section of the report, but for now when the *dispatcher* sends the message "recetas" to the microservice, it initiates the Spanish

bot conversation attached to the user ID/ mobile phone number of the user.

The bot then sends the appropriate Spanish language start menu so that the user can perform the medication options. When the user makes a choice the message is sent via Signal to the *dispatcher*, the *dispatcher* attaches a token, and encrypts the message and sends it to the medications microservice, and the bot responds accordingly.

3.5 Serving the Flask App

The Flask app is a simple application that is known to be poorly suited for production servers but if placed inside a multithreading container it is more than capable of handling chatbot tasks. The web server gateway used to give multithreading capabilities to the Flask application is Gunicorn. Below in Figure 3.2 is a detailed view of the Gunicorn-Flask-Chatbot flow.

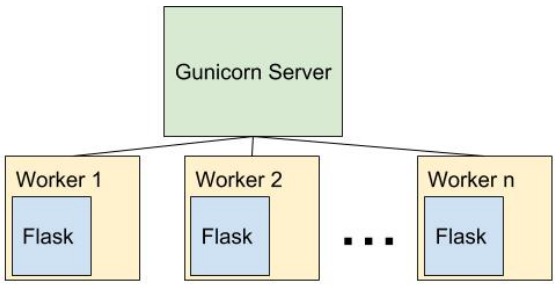

Figure 3.2: Structure of the Gunicorn - Flask - Application relationship.

Gunicorn is a HTTP server utility based on WSGI for Python. It is capable of spawning a multitude of workers on separate threads for handling simultaneous server access. It is also the piece that links the localhost port that receives external messages and hands them to the Flask application. This is done in the code below, where Gunicorn is initiated, bound to the localhost at port 84. The next parameter *wsgi:app* tells Gunicorn where to find the Flask app. The last two parameters are for loading the security key and certificate files.

```
"gunicorn", "--bind", "0.0.0.0:84", "wsgi:app",
"--certfile","/etc/ssl/PrivateCerts/userver.crt",
"--keyfile","/etc/ssl/PrivateCerts/userver.key"]
```

3.6 The Docker Container

For reasons of portability, expandability and maintenance as detailed previously in the Introduction, Docker was chosen as the shipping container of the microservice. The Docker image is a Debian based Unix distribution called Ubuntu version 16.4.

With the microservice running successfully in the Flask app, which is served by Gunicorn the command *pip freeze* was run to get a list of all dependencies. This list was saved as the requirements.txt file which is included in the Appendix C. The Docker image runs a complete install of Python 3 as well as an installation of all of the dependencies to be contained neatly inside its Docker container. This is handled by the Docker makefile. By running the command *pip install requirements.txt* all of the libraries could be installed to the Docker image.

A couple of the libraries had to be created or modified, such as the pyAIML library which was greatly expanded on, as detailed in chapter 4. In order to install in the Docker build-file they first had to be turned into complete Python packages. They could then be added to the Docker makefile as single line instructions to install. This is shown in the script at the end of Appendix C.

Once the Docker image is made it is trivial to upload it to the server. A utility is provided by Docker that allows for pushing completed images. Once the image is uploaded it is given a version number. The microservice administrator can then decide to exchange the current active Docker image for the new one. This allows for hosting both staging and production servers and easily exporting from one to

the other depending on need.

The incentives to use Docker are that it is self contained, very stable, and has no real access to the computer that is hosting it. These pieces make it extremely secure, which is a boon for medical applications, but this does make changing common parameters a bit of a hassle. One could change the source code for the program, take care to make sure that any modified files are the last items in the Docker build file, then rebuild the Docker image. But if the user changes even a byte of an earlier file in the Docker build file then all of the following instructions in the build have to be performed. A better method is to include a configuration file in the Docker image that can be accessed through the Docker manager. This task is not trivial as it is all dealing with memory addresses, but once the configuration file has been found it is quite straight forward to change it. In this project the configuration file contains the URL's for the security server, messaging, *dispatcher*, database, the message structure and many other important items.

3.7 HTML hosted web services

The secondary task of the Flask app is to host the HTML based utilities. These were all written in HTML5 with CSS and JavaScript. The first utility written was the message debug tool. This tool emulates the messages sent from *dispatcher* to the Medications Microservice. This was important to help isolate errors to certain functions of the microservice and to more easily follow the messages through the platform. The web interface is shown below in Figure 3.3, and it is accessed by making a secure HTTPS connection to the localhost.

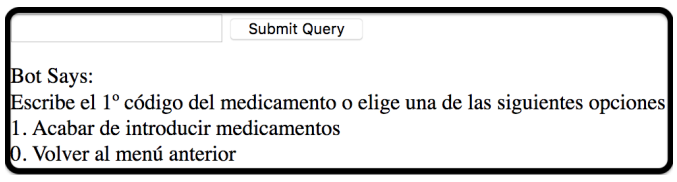

Figure 3.3: Example of the web interface used for testing the chatbot functionality.

Also in the secondary task of the Flask app are some of the experimental features that are implemented in the code but not activated in the final release because they are incomplete or buggy. One of these is a template generator for the OCR application in the bot. This application will be discussed further in section 3.9, in a quick summary it was an partially successful attempt to extract medication information from a picture of a prescription. The web utility for this function is designed to be able to load in a redacted or template prescription sheet, and then allow the app maintainer to manually configure the bounding boxes for each field. Shown below in Figure 3.4 is an example of the template generator for an Aragon Salud prescription sheet. The dialogue box to the right of the image shows the coordinates of the vertices of the box, which can be adjusted using the arrows for fine tuning. When the user presses submit, the coordinates are sent in a POST message back to the application and saved in a JSON file for later use in the OCR.

Figure 3.4: Web based utility for creating Prescription Sheet templates used in the OCR.

3.8 Bot Functions and Conversation

In the previous few sections the basic functionality of the bot and microservice were detailed. These all mesh together to form a complete medication solution which is shown in Figure 3.5. This graphic ties together all of the roles of each. From right to left a message is sent by the user through the Signal application. The message is received by the *dispatcher* microservice, which then decides which microservice to send it to. *dispatcher* then reformats the header and sends a POST message to the Medication microservice which then performs the security check detailed in section 2.2 to decide if the message is secure and valid. It is then decrypted and internally sent to the chatbot function. The chatbot in turn sends an encrypted message to the *dispatcher* microservice which then communicates with the Signal app on the user's phone.

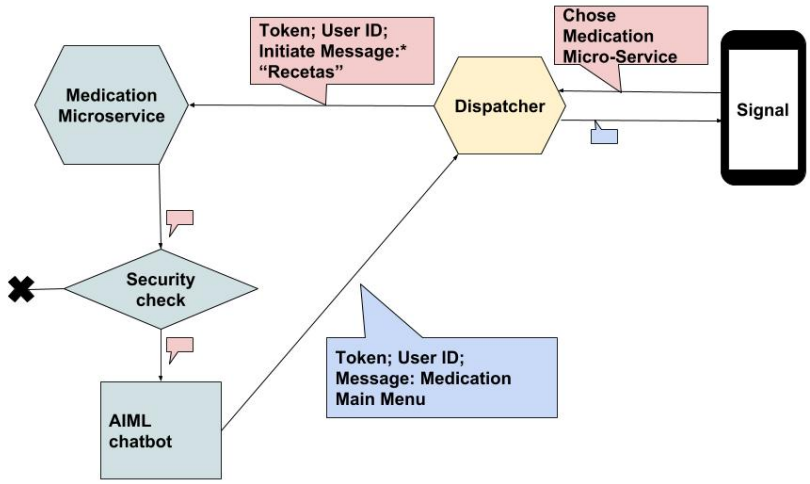

Figure 3.5: Detailed flowchart showing the flow of a message from phone to the microservice and back.

Before the actual chatbot can be written it is important to have all of the logic and flow of the dialogue decided on and figured out. For each of the main functions of the chatbot a decision tree was made. The program was divided into the following main decision trees:

- **Main menu:** The principle menu that the user sees when entering the

microservice. Displays the items listed below, as well as an option to exit.

- **Add Medication:** Function to manually add medications. Proceeds through a series of prompts for all the necessary medication information.
- **View Medication:** tool to view current medications that the user has active. This can be viewed by doctors, caregivers etc.
- **Change Medication:** Accommodation to allow for the correction of incorrectly entered medications.
- **Generate Notifications:** Function that reads in the active medications and prompts the user to generate notifications. Proceeds through a series of questions to configure the notifications properly.
- **Change Notifications:** Tool to change and view medication notifications. Prompts for which notifications are erroneous and then proceeds through the same prompts as the *Generate Notification* function.

Because it is the most representative function of the rest, the *add medication* extension has been chosen to illustrate a typical AIML extension decision tree in Figure 3.6. This tree is similar to that of other extensions. The complete list of which and their roles is shown in Appendix D.

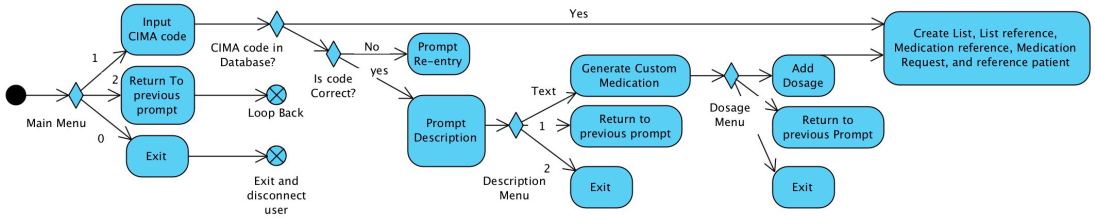

Figure 3.6: Decision tree of a segment of chatbot conversation. Shown is the conversation for manually adding a new medication.

The decision trees gave a clear understanding to the pathways, dialogue and relations needed to make the chatbot. Transcribing to AIML is quite straightfor-

ward from their but adding all of the needed functionality for this project was not trivial. This is all explained in further detail in the next section.

3.9 OCR

The final piece to this microservice was an experimental application using an Optical Character Reader (OCR) to parse a prescription sheet and save the medication information much in the same way as the manual entry. This portion met with limited success but has lain the foundation for implementation once some of the issues are resolved. Many powerful OCR tools exist but the *Tesseract OCR* was chosen due to its high level of documentation, integration into Python and the GPL license (*15*).

OCR's are very good at parsing characters and accurately creating strings that reflect the written characters. Where they have room for improvement is in recognizing page formats. In prescription sheets this is very important because in many prescription sheet formats the columns indicate specific properties of the medication. Initially histograms of a binary image were used to find specific lines and columns in the text, but when multi-line prescription descriptions were used this confused the engine. Another method was looked at to separate the columns more accurately and that was using a predefined template.

The template is generated using the web utility shown in Figure 3.4. This saves a JSON encoded template of coordinates for each field. This all worked very well but still found errors attempting to consistently get all rows within different columns matched. That is to say matching a medication code from the first column consistently with the description in the second column. This feature requires only a little more work to be fully functional but as such it was left out of the production microservice.

Chapter 4

User - Chatbot Communications

The foundation for communications in this microservice is in the AIML chatbot. AIML as previously noted in subsection 3.1.1 and detailed a little in chapter 3 is a markup language used to define a conversation. The principle unit of a conversation in AIML is the *category* tag. This tag contains the *pattern* which is used to match the *category* to an input, and the *template* tag which defines the response. The *template* is a special tag that can contains almost every other tag that is in the AIML system.

Within the template tag the script can define and read variables with the *get* and *set* tags. The *think* tag can be used inside the *template* but surrounding any text and tags to suppress output. This is useful if a function is used that outputs a value to another function and it is not desired that the output is returned in the chatbot response. Below is a small sample of AIML script using the above tags:

```
<category>
  <pattern>*</pattern>
  <that>* MEDICATION<\THAT>
    <template>
    <think>
      <set name = topic>Medication</set>
```

```
        <set name = "currentMed" > <showmed><get name =
            "currentRequestID"/></showmed></set>
      </think>
      <condition name = "currentMed">
        <li value = " none ">Could not find <input/> Are you sure the
            code is correct?</li>
        <li>found: <get name = "currentMed"/>. Is that correct?</li>
      </condition>
    </template>
</category>
```

The above script also introduces some more advanced functions of AIML. The first of which is the * in the pattern tag. This character tells the pattern that any input is acceptable to access this *category*. This is useful as a fall back *category* if no other *pattern* is satisfied. The next tag, *that* provides a limiting factor to the *pattern*, in order for the *pattern* to be satisfied the last message that the chatbot sent has to match the contents of the *that* tag. This tag can be omitted if the *category* is not dependant on previous messages but *that* is quite useful for conversation continuity. Within the *think* tags is another piece that is useful for continuity. The *topic* tag. This will be described below in further detail along with the *srai* tag which is not shown above. Finally above is the *condition* tag. This is a sort of switch/case block where a variable is loaded with *<condition name = ...>* and then compared against values to return the desired output. This part is outside the *think* which means its output will be included in the chatbot response.

4.1 Multi-lingual support

The author of this project is a native English speaker, but the program is intended for a Spanish audience. Rather than attempt figure out all of the logic in

Spanish, as is typically the route in writing AIML code, it was decided to design a novel method for inter-lingual ready AIML. To understand how this was done it is first important to understand a few key aspects and behaviours of AIML that were instrumental in designing the translation capabilities.

4.1.1 The *Srai* tag

This tag doesn't have an official acronym, but is known as symbolic reduction or symbolic recursion. Four common terms for functions that can be done with the *srai* tage are:(*16*)

- Symbolic Reduction
- Divide and Conquer
- Synonyms resolution
- Keywords detection

The third term, synonym resolution is perhaps the most relevant to the translation case. Translations are in some sense just fancy synonyms. In this case an entire dictionary is kept where every single line is *srai* linked to its other language counterpart. An example of this is shown below:

```
<category>
  <pattern>1</pattern>
    <template>
      <think>
        <set name = "topic">SPANISH</set>
      </think>
    <srai>DONE ADDING MEDICATIONS</srai>
  </template>
</category>
```

```
<category>
  <pattern>DONE ADDING MEDICATIONS</pattern>
    <template>
      Muy bien, todos los medicamentos están en nuestro sistema. Por
          favor indica una de las siguientes opciones
      <br/>
      1. Ver todos los medicamentos
       <srai>ZERO OPTION</srai>
    <think>
      <set name = "topic">FULL MANUAL ENTRY</set>
    </think>
  </template>
</category>
```

Above the first block of code shows the *srai* tag used to get the response from the second block of code. Furthermore a third level of recursion is in the second block which calls a third block, not shown, with a repeated menu element. The final response from calling the first block is:

Muy bien, todos los medicamentos están en nuestro sistema. Por favor indica una de las siguientes opciones

1. *Ver todos los medicamientos*
0. *Volver al menu anterior*

4.1.2 The *Topic* tag

The *srai* tag is useful for linking in words and translations, but it can be very confusing to maintain lists of translations for various languages, or to keep track of what part of the conversation it is being used in. To solve this the *topic* tag

was used. The *topic* tag is essentially an excluder. Any categories not listed inside the *topic* tag of the current *topic* are completely excluded from the search. This is advantageous as the AIML kernel has a top down parser and this tells it to skip all other tags except those of interest. In the case of foreign language support a language topic is made for every supported language. The computer can then, based on the user ID which is also the phone number, determine the default language to chose.

Once the dictionary topic is chosen and the correct translation has been used the parser returns from the *srai* tag and parses any other tags inside the *template* tags. At this point it is crucial to remember what is the current *topic*. It is still the dictionary of the given language, therefore after every language related *srai* a *topic* change must be done to make sure that the kernel stays on the current logical topic, not the lexical one. This translation facility does add the complexity of having to always remember what the current logical and lexical topics are but if the diagram below is used to standardize the method of translation and logic then it is quite straight forward.

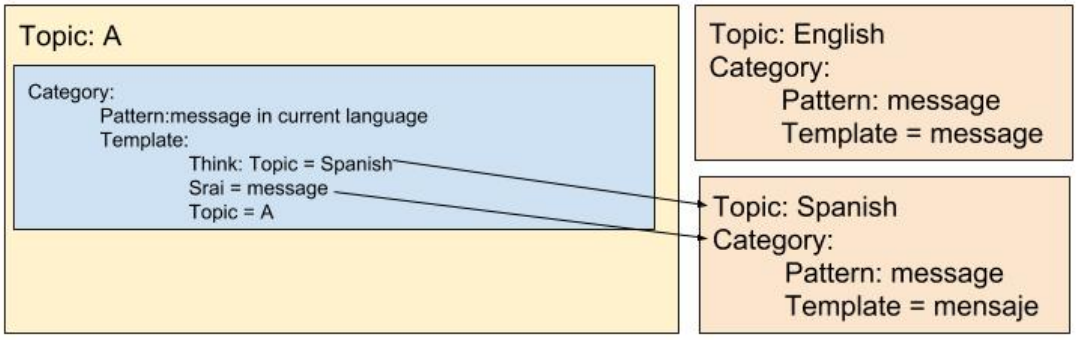

Figure 4.1: Representation of the logical flow of the translation capabilities of the chatbot.

In Figure 4.1 there are two *topic* colours, yellow and orange. The yellow topic is the actual conversation. This contains the logical flow of conversation but none of the actual words that are sent in the response. Instead the *topic* is changed internally in each *category* of the conversation *topic* to the target language. In the

above case that is Spanish. With the *topic* set to Spanish the symbolic reduction performed by *srai* finds the correct *category* within the Spanish *topic* whose *pattern* matches the *srai*. This response "mensaje" is then bright back into the original *category* in *topic* A and is added to the response. Finally the *topic* is reset to "A" in order to continue the flow of conversation.

4.2 Expansion

The basic AIML library for Python 3 is restricted to the capabilities from the AIML standard 1.0 developed by Cort Stratton and Paulo Villegas. This standard is very capable for most conversational means, even including quite complex conversations. It does not, however, have facility to access Out Of Bounds functions. This ability is key to a bot that has to interact with data. The AIML 2.0 standard implemented many new abilities but this was by far the one most required in order to develop the Medication microservice. Though eventually the entire AIML 2.0 specification is the goal, this function was the best place to start.

It was desired that the base pyAIML library was left as unchanged as possible, so that as the main library updated it would be easy to implement a couple of small changes to integrate the extensions. The first attempt at doing this was overly complex and tried to add rules and exceptions to all of the different existing tags, to show their relationships with extension tags. This was fraught with errors and quickly abandoned. Instead the below method was chosen. This is where the key piece to adding functionality arises. It was decided that the optimal method for adding functionality would be to create a new kernel that would be called whenever a tag failed to pass the initial check against acceptable tags. An external list of the newly developed AIML 2.0 tags and other extension tags would be invoked. The "self._processExtension(elem, sessionID)" function was of key importance to get correct. The *elem* variable is a Python list of all the tags and their contents

recursively outside the current tag but in the current *category*. The first level of recursion in the element is stored at position zero, but for the external processor it was important to hand off the entire remaining recursion tree. This is because the external functions have a much more limited access to the bot brain. For native AIML tags the brain remembers the element tree, sessionID of a given conversation The session ID is needed by external processes so that any communication with the database or *dispatcher* originating from the extension processes have the user related.

```
if elem[0] in _ExtValidation:
   handlerResp = self._processExtension(elem, sessionID)
    return handlerResp
else:
   handlerFunc = self._elementProcessors[elem[0]]
   return handlerFunc(elem, sessionID)
```

These two snippets of code are very helpful at explaining the overall working of the pyAIML library, the above shows that the“Kernel” class is called by “self” to start the _processExtension() function. That function is shown below and is a modified version of the _elementProcessors() function from the pyAIML library. The function receives the element tree, and checks its length, if less than three it means that the tree is only an extension call. If it is of size 2x>1 then it is an extension call contained within another unknown process. This process is then sent back to the original process element function to decide whether it is an extension or a native function. Beyond the desire to maintain the original functionality of the library, this last piece was important to allow for the proper processing of the *think* tags. This tag in AIML restricts the output of a function so that it is not added to the response string. Therefore when an extension process is contained within a *think* tag the processExtension function receives it, sees the think tag and

makes certain that the extension is processed silently.

```
def _processExtension(self, elem, sessionID):
   response = ""
    value = ""
    if len(elem)<3:
      fx = getattr(KernelExtension(), "ExtensionProcessors")
        response += fx[elem[0]](None, sessionID, self)
    elif elem[2][0] in self._elementProcessors:
      for e in elem[2:]:
         value += self._processElement(e, sessionID)
           name1 = elem[0]
        fx = getattr(KernelExtension(), "ExtensionProcessors")
        response += fx[name1](value, sessionID, self)
      return response
```

The complete list of extensions is provided in Appendix D, some are of particular note, because they illustrate some of the greater functionality of the program quite well. While others, such as *br* are simply formatting or mathematical functions. The next few paragraphs will describe the function in detail of the *newlist* and *gennotification* tags.

4.2.1 New list Extension

This custom AIML extension function serves to generate a new list associated with the current patient. It is called when a user initiates a new medication input. The lists are a collection of all medications input in that session, but can also be edited so that previous lists can have medications added. This extension was chosen to be explained because it does a good job representing the FHIR implementation of a prescription sheet as well as a sort of generic framework for all other FHIR interactions that the bot does.

The first thing done by this function is to generate a new list. That list is then assigned the status of current, meaning that it is an active current list of medications. This is important because as the prescriptions expire it is needed that this status be switched to "retired". The mode shows that this list is not finished and that it is a working copy. At this point the function is able to call in the patient FHIR identification, based on the user ID/phone number. That patient is then referenced into the list and the list is submitted to the server.

```
new_list = l.List()
new_list.status = 'current'
new_list.mode = 'working'
search = pat.Patient.where(struct={"telecom": 'phone|' + sessionID})
subject = search.perform_resources(smart.server)[0]
sub_ref = ref.FHIRReference()
sub_ref.reference = subject.relativePath()
new_list.subject = sub_ref
new_list.source = sub_ref
post_ret = smart.server.post_json(new_list.relativeBase(),
    new_list.as_json())
req_ret =
    smart.server.request_json(post_ret.headers._store['location'][1])
r = l.List(jsondict=req_ret).id
```

The *new list* function is key to initiate the list but it still must be filled in with a medication. This handled in a similar function where the medication code input by the user is compared against the local database, and if found the list is loaded, a reference is made to the medication via a Medication Request object and the reference is saved into the list as a List Item.

4.2.2 Generate Notifications Extensions

The *gennotification* function is well suited to describe the notification protocol, as well as the math and logic used to calculate the notification schedule from a given input. A particular issue with any sort of input that is quite non-standard like dates and frequency, is trying to frame the input in a standard way. To do this some special structuring of the AIML had to be done, this is shown below.

```
<gennotification>{
   "yearDates":["<get name = "yearFreq"/>"],
    "MonthDates":["<get name = "monthFreq"/>"],
    "WeekDays":["<get name = "weekFreq"/>"],
    "DayHours":["<get name = "dayFreq"/>"],
    "HourMins":["<get name = "hourFreq"/>"],
    "RepNum":"<get name = "repeatNum"/>",
    "StartDate":"<input/>",
    "FhirID":
    "<oneFhirID>
        <get name = "currMedList"/>,
          <get name = "medCounter"/>
    </oneFhirID>"}
</gennotification>
```

This snippet of AIML contains two standard tags, the *get* and *input* tags. The *get* tag is used to recall variables that have been saved in other sections of the AIML. While *input* is used to get the last text input by the user. For this section the last question that the user is asked is *"When is the Start Date"*, for which the *input* tag extracts that string. The formatting of this section is very important, the *gennotifications* custom tag requires that its input be in XML format. As seen by the { } and key value pairs the XML format is hard coded in the AIML instead of relying on users correctly typing out the arguments. The other custom tag in this

section is the *oneFhirID* tag. This tag sends the entire current active medication list, and a counter number, to the oneFhirID processor function. That function then selects the appropriate FHIR id from the list based on which iteration through the list the code is on. The use of FHIR is helpful here in that only the FHIR id needs to be passed around, instead of the entire medication object.

Once the XML is properly formatted and the correct medication is associated with the notification, the XML is passed to the genNotification processor. There the XML is parsed and the below logic is used to separate irregularly repeating notifications into regular simple notifications that the notifications microservice can understand. This logic works by a checklist. The parser determines what is the largest unit of time referenced in the notification. That is if a notification is three days per week, the largest unit is a week. If it is 3 pills in on the 1st and 15th days of every month then month is the largest unit. Once that is found the start date is calculated by determining the unit and seeing if the event can happen before the next unit starts or not. For example if a patient is to take a pill on Monday, Wednesday and Thursday every week at 10:00 in the morning but it is Wednesday the 10th at 17:00 when the notification is generated the generator must know to put a notification for Thursday that starts on Thursday the 11th, but also it needs to generate two separate notification initiators for Monday the 15th and Wednesday the 17th. This is critical because if a prescription such as an antibiotic must be taken for 10 iterations and the program were to count Wednesday the 10th as already having happened but it didn't then the patient would stop the Wednesday administration a week before the doctor had planned.

Finally the notifications are generated by a POST message generated with the following code. There the body is the XML style notification information.

```
r = session.post(notifications_url,data=body,
  headers={'Authorization': token,'content-type':
      'application/json'},verify=False)
```

Once all of these additional features were written the conversational logic had to be established. The nature of the microservice means that some very smart conversational rules have to be set. This is because as a user enters a list of medications, goes to modify existing values, or wishes to enter in notification information, the bot has to be able to loop through all of the relevant entries. This is a problem for AIML because although it does have memory for the last several conversation points, it has no built in functions for looping, counting, knowing what previous topics were used. To overcome this some inventive systems of bot based variables, extension functions and topic switching was employed.

4.3 Notifications Logic and Language Processing

With the bot functions all working effort was then taken to add to the capabilities of the bot and to lay the foundation for future projects. One of these that is a desired in later iterations of the eHealth Bot is that the user can communicate fully in natural language to interact with the bot. During the development of the Notifications Generation conversation sections especially it was logical to include the ability to enable natural language processing. This is because reminders and notifications have such a varying degree of types and a simple menu based selector would either not have the ability to deal with complex reminders or would rely on too large of a series of menus to accurately represent the desired notification. Some example notification schedules that had to be accommodated

- 1 each day
- 3 every day
- 5 pills, 2 times a day, each day
- 3 pills a week on Monday, Tuesday and Saturday

As can be seen these have a lot of variation in wording and need. In order to interpret this a Python Library called *dateparser* which is quite robust at parsing date text and returning the correct Python *datetime* object. A key piece to this was for the Python bot to perform an initial parsing so that the library functions would receive the words as they were desired. For the above examples the word EACH was chosen as a separate the number value from the period that it is to repeat. This is shown previously in the code in the section detailing the programming of the notification functions. There those the number value is known as the time unit, and the period is known as the *period unit*. This breakdown allows the bot to interpret a variety of notification date structures with high accuracy. In the implemented version of the bot this was not enabled because it was not consistent with the menu based system that was part of the current user experience specifications of the University of Zaragoza team. They do however desire that future implementations will be able to use this and so it was left in a state that can be easily activated.

Some other pieces of natural language processing that had to be addressed were multiple language support, misspelling and ensuring intuitiveness. The first two of those were addressed in early implementations in this project as well. As explained above in the AIML section it details that the bot is able to know the language of the user and send the appropriate menus. That function only allows for the bot to send multi-language menus, it has no utility for interpreting received multi-language messages. In order to do this a preliminary tool was experimented with to introduce a lookup table of common expressions that would be appropriate given a certain situation. The lookup table is a function that runs on the Python side of the bot. It intercepts the message before it reaches the AIML kernel. The bot is already aware of the local language of the message so it is trivial to separate the string into words and translate to English. The English messages are then sent to the bot so that the bot can find the correct response. On the bot response end however the translation happens in the AIML. This is because it is easier to write

translations for formatted text in AIML than it is to train rely on a computer based translator for correct translations. The bot performs the same function as detailed above in the AIML section. The topic is changed to the value of *CurrentLanguage*, which in the current case would be Spanish, and a *srai* tag is used to substitute in the correctly translated text.

Chapter 5

Results

The results of this project aim to show how it has accomplished the goals and demonstrate its performance in a case-study type scenario. Due to time constraints the real-world use results for the project have not been included. The project as of writing is leaving its Beta phase and will very soon be placed into real world use cases with a group of nurses and patients, under the custodianship of the team at the University of Zaragoza. Instead, to supplement these real-world results, a more technical summary of the FHIR migration and the achieved chat functionality along with where they were successful or lacking. Following that, is a simulated interaction will be presented.

5.1 FHIR Specification Application

The program written to automatically parse the CIMA database, convert their structures and references to the FHIR specification was key to the implementation of the Medication microservice. This self contained app parses through over 21,000 medication records, 3,700 ingredients, 3,700 laboratories and generates the correct FHIR objects for all of them in around 8 hours on a 2014 era laptop with 8 gigabytes of RAM. This does seem a little long but the existing database is entirely text based

and a custom parser had to be written and each reference and object is uploaded via individual POST messages to the database.

The newly uploaded medications though do follow perfectly the FHIR specification and are accessible to all other microservices on the platform. The picture below shows the total medication records, laboratories and other FHIR specification objects that have are in use by the platform. Medication objects make up the vast majority of records, and this aims to highlight how important it is that they are up to a standard that is recognized easily across microservices and by any other new solution that may take over.

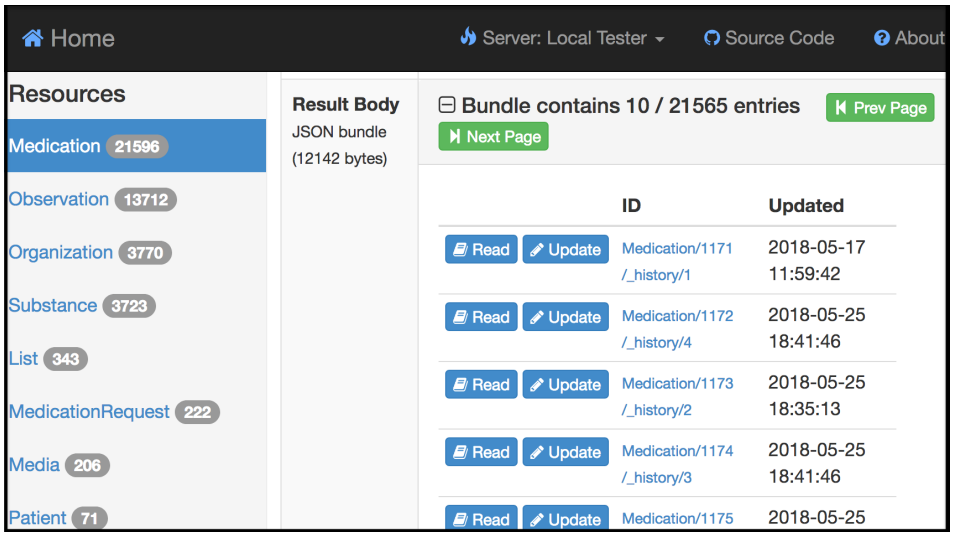

Figure 5.1: The web interface for the HAPI FHIR server, showing a sample of the *Medication* entries.

5.2 Functionality

The individual bot functionalities were laid out in the objectives, but throughout the development process certain aspects of them differed. In some cases they have more functionalities and flexibility, in others certain unforeseen technical difficulties required that aspects be adjusted. Below are listed four of the core functionalities, with the desired outcome, some explanation of how they in fact behave and a discussion of the success, compromise, or difficulties that each had.

5.2.1 Manual Entry of Medications

The application works fully for inputting most medications. A few medications were found however that used outdated medication codes or non-standard AEMPS codes. These codes were of course not on the database that was imported from AEMPS CIMA. To accommodate this while the databases that these few medications came from are tracked down a utility to override the code if not found was made.

A decision was made here to only have the user who created the medication have access to the custom medication. This was because it would be very unsafe for a non-medical professional to be trusted in transcribing medical information that would then be relied upon by other patients. Therefore when a custom medication is made it is linked via a FHIR object reference to the identifier of the patient or caregiver who created it.

5.2.2 Generate Notifications

The notification generating utility was written with the ability for near full Natural Language processing of scheduling information, but was then modified to conform with the other menus. This was decided to avoid confusing the user. The functionality was left intact in the code so that it can be activated in the future if needed.

The automatic generation of notifications was also experimented with and was accomplished with high success. The generator is able to process the recommended dosage period and extrapolate a suggested notification schedule. This feature is not implemented in the production version but is also available for when it has been tested more and optimized.

5.3 A Full Interaction

With the core functionalities fully explained, the following is a simulated conversation with concise explanations of each step in the interaction process. Shown the following four figures are highlights of two of the main functions of the microservice. One is the addition of medications using the CIMA medication codes. The other is applying notifications for the administration of those medications.

The two figures below, Figure 5.2 and Figure 5.3, show the conversation for the addition of a CIMA medication. First the *add medication* option is selected by the user. The bot then prompts the user for the CIMA medication code. If the code is not found the bot would ask for confirmation that the code is correct. In the case below the medication code is correct, 68025, corresponding with *Liofara*. Although the medication was found the bot still needs to confirm that the found medication is the intended medication by the user.

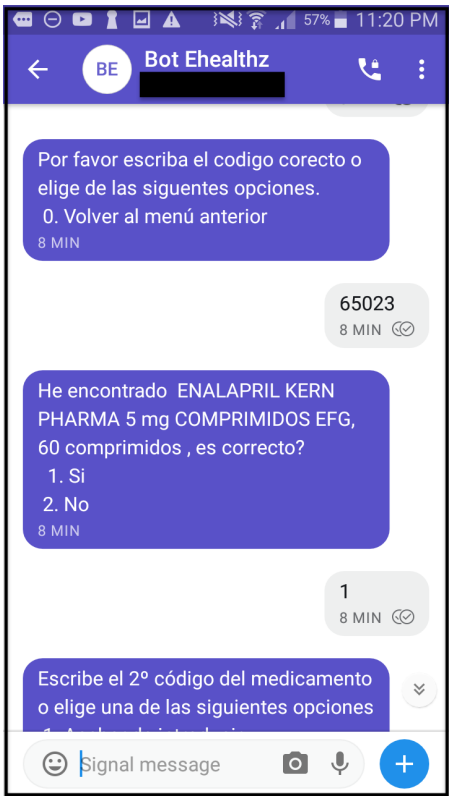

Figure 5.2: Shows the selection of the "add medication" option followed by the input of a code and confirmation that the found medication is correct.

Behind the scenes in the above code the microservice has already generated all of the FHIR objects necessary to assemble the prescription sheet. All of the objects are marked with the tag ”working” so that they are not considered ”active” but are available for editing and viewing. Once a medication is fully added and confirmed the corresponding objects become ”active”. The front end for this is shown below in Figure 5.3.

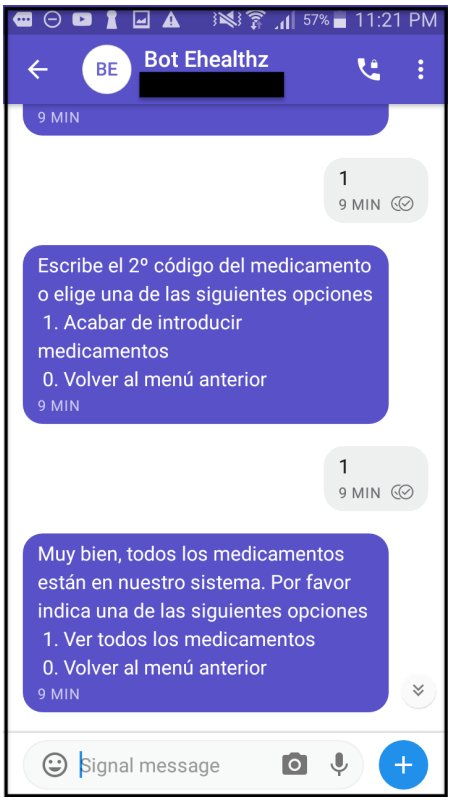

Figure 5.3: Sample of a successfully entered medication.

Finally a medication is confirmed and the user has the option to continue entering as many medications as needed or to return to the main menu to perform other actions. Not shown in these two figures is the manual entry of medication, but it performs much the same way, with an option to return to the previous question to fix any erroneous entries.

The other major function of the chatbot is the entry of notifications. This begins with the user entering the generate notifications menu option and then

receiving a list of all active medications. The user must then choose the medications they want to add notifications for. This is shown below in Figure 5.4, where number 4 *Liofora* is chosen for adding a notification.

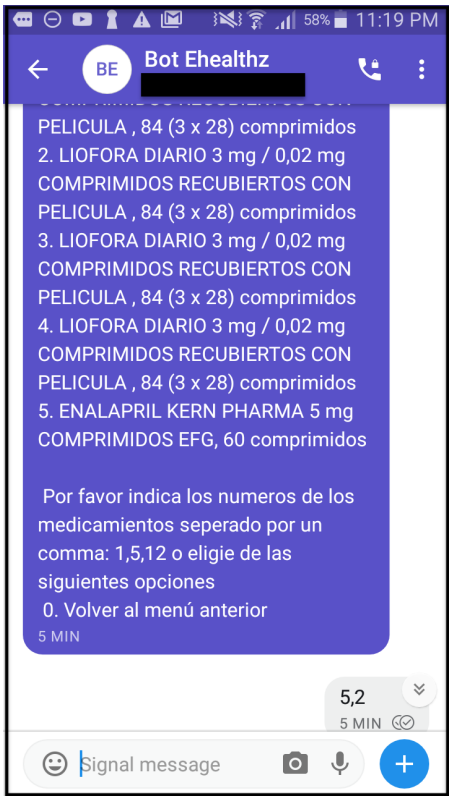

Figure 5.4: Conversation between bot and user during the generate notification function of the bot.

The bot then asks for the administration frequency in various steps, starting from most general, to most specific. Here the user has entered twice a week, then on certain days etc. Below in Figure 5.5 the final questions are shown and upon answering the start date question the notification is generated. If there were further medications to generate notifications for the bot would then prompt the information for the next desired medication. In this case, because there are no further medications the bot returns to the main menu.

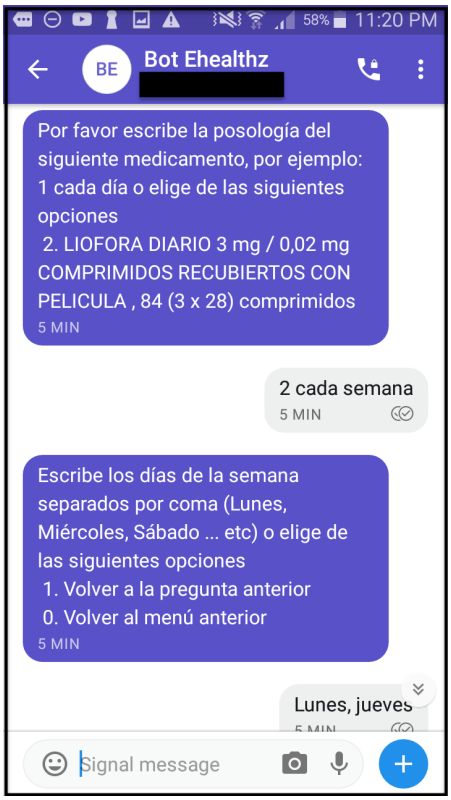

Figure 5.5: Dialogue for filling the notification by asking frequency and which days.

Chapter 6

Conclusions and Future work

6.1 Conclusions

This project was able to attain all of the objectives set forth as well as perform preliminary investigation into further work that would better the usability of the microservice. That is it has designed an eHealth solution for the monitoring and administering of medications, it has also thoroughly investigated the Spanish AEMPS CIMA medication specification and translated it to the internationally recognized FHIR specification. It has furthered these aims by also developing the foundation for an OCR tool to be used for automatically generating medication information for the patients. To accomplish these main goals the following subtasks were also completed:

- A study of the AEMPS CIMA specification in which there is no available documentation.
- A review of medication specifications, followed by an in depth study of FHIR.
- Expand the pyAIML library to allow for AIML 2.0 tags as well as adding all the extensions necessary for this microservice.
- Develop Multi-lingual chatbot logic in AIML.

- Develop a working secure chatbot that can be successfully hosted as a microservice allowing for multiple connections at once.

- Implementation of online browser based tools for administrative tasks on the microservice.

6.2 Future work

As with any project there were goals that could not be accomplished and ideas that were developed later on that would beneficial to add to the project but were not included in the original scope. It is important in a full deployment project to recognize when to stop adding features and to finish the features desired by the project outline. Some of the features and ideas that did not make the final version of the microservice are included below for future implementation.

6.2.1 With FHIR

The CIMA database provides more information than is currently needed by any existing microservice, so it was not deemed important to fully transfer each record in its entirety. A perfect example of this is that on the CIMA database pictures of the packaging and the medication itself are recorded. These pictures would make a great addition to the FHIR database and pave the way for some other new functionality that will be discussed in the Medication microservice's future work.

Concurrently the stand-alone application that parses the CIMA database is entirely command line driven, requires an instance of Python on the local machine, and is a little cumbersome to use. Future work for this would involve making an HTML web interface to maintain the database transfer and an automatic bot that manages the day to day checking of the database to make sure that all records

are up to date. These two functionalities would greatly improve the usability and reliability of the FHIR translator.

6.2.2 The Microservice

The microservice is far from complete in that there are many medication based functionalities that will be desired. These are often found through the trial period, but during the development some of these became apparent but were chosen to not be pursued in the immediate future for the desire of releasing a complete solution instead of a mix and matched sampling of all the capabilities of the platform.

As mentioned previously there is an untapped resource in the pictures provided by the Spanish CIMA database. These pictures can be easily entered into the FHIR database model that is currently being used with the platform. Once on the FHIR database it is merely a matter of configuring a mircroservice to request the images in order to use them in a variety of applications. Some of the most exciting paths for this are for care of elderly and sufferers of dementia. Using template matching a microservice could be designed to validate that a patient is taking the correct medications, automatically confirm that a medication has been taken. As well it could confirm with patients, doctors and caregivers the correct dosage, appearance and form of medications.

As mentioned previously, the OCR functionality was never completed but it was started and advanced to a near BETA level. Future work in this would involve improving the consistency of the row separator of a prescription sheet. Once that has been achieved this function will be complete.

Another utility of the platform that was largely left unused by this microservice is the separation of roles. The platform allows for the assignation of caregiver, doctor and patient roles. It is important to implement these so that the permitted actions of each role can be defined within the microservice. The current microservice only allows for the patient and doctor style role. This is because

there is a lot more overlap between these two. A doctor should have access to the medications and the ability to change them if entered incorrectly. Where as the implementation of a caregiver type role would be more of a passive role that can view only the things that the patient has deemed necessary for them to see. This might be for one patient a simple binary response about whether the medications for that day have been taken, up to a complete summary of all medications currently, and historically taken along with charts about the usage and how often the patient remembers to take them.

Bibliography

1. K. Patel *et al.*, *BMC Clinical Pharmacology* **7**, DOI: 10.1186/1472-6904-7-8 (2007) (cit. on p. 1).
2. M. Pirmohamed *et al.*, *BMJ* **329**, 15–19 (2004) (cit. on p. 1).
3. C. Stratton, *pyAIML*, 5th Aug. 2017, (http://pyaiml.sourceforge.net/) (cit. on p. 4).
4. K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, D. Stebila, presented at the, pp. 451–466, DOI: 10.1109/EuroSP.2017.27 (cit. on p. 4).
5. O. W. Systems, contributors, *Signal* (cit. on pp. 4, 15).
6. *FHIR, HL7.org*, 2011 (cit. on p. 7).
7. D. Bender, K. Sartipi, *HL7 FHIR: An agile and RESTful approach to healthcare information exchange*, June 2013 (cit. on p. 7).
8. R. Wallace, *A.L.I.C.E*, ALICE A.I. Foundation, 1995 (cit. on p. 13).
9. *GNU General Public License*, version 3, Free Software Foundation, 29th June 2007, (http://www.gnu.org/licenses/gpl.html) (cit. on p. 15).
10. T. F. Bissyande, F. Thung, D. Lo, L. Jiang, L. Reveillere, presented at the 2013 IEEE 37th Annual Computer Software and Applications Conference, pp. 303–312, ISBN: 978-0-7695-4986-6, DOI: 10.1109/COMPSAC.2013.55, (2018; http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6649842) (cit. on p. 15).

11. D. Robinson, *The Incredible Growth of Python | Stack Overflow*, en-US, Sept. 2017, (2018; `https://stackoverflow.blog/2017/09/06/incredible-growth-python/`) (cit. on p. 15).

12. T. A. S. Foundation, *Apache: MESOS* (cit. on p. 17).

13. R. Docker, B. Knight, W. Davies, D. Presley, *DOCKER: Orchestral Works* (cit. on p. 17).

14. *Flask User's Guide*, Pallot's Team, 2010 (cit. on p. 20).

15. R. Smith, presented at the Ninth International Conference on Document Analysis and Recognition (ICDAR 2007), vol. 2, pp. 629–633, DOI: `10.1109/ICDAR.2007.4376991` (cit. on p. 27).

16. R. Wallace, *The Elements of AIML Style* (cit. on p. 30).

Appendix A

Comparing Messaging clients

Below is a table detailing the differences in features of various messenger platforms. Below it can be seen that in features that were important to this project Signal was the clear choice at the time of beginning.

	Whatsapp	Telegram	Signal	Skype	Viber
Encryption by default	☑	☐	☑	☑	☑
Source code available	☐	☑	☑	☐	☑
generate & keep a private	☐	☐	☑	☐	☐
Encrypted Metadata	☐	☐	☑	☐	☐
Secondary Factor Identification	☑	☑	☐	☐	☐

Figure A.1: Table detailing the different features of various messaging clients

These requirements are of particular import because of the nature of this project. The first, third, fourth and fifth requirements are all security based. Medical data and the security of patient information is of the utmost importance when designing an eHealth solution. Secondary factor identification was desired but it was not a deal breaker when deciding on a platform. The second criteria, that the source code is available, is incredibly important because that gives the developer the ability to truly customize the application without having to worry about com-

plex corporate licensing and a more robust development community to aid in the development process.

Appendix B

Time Distribution of the Project

The project spanned a year from its inception in December 2017 to its presentation in December 2018. This time can be separated into the main segments as detailed below in figure Figure B.1. The figure shows the projected timeline in a Gantt chart. Much time was allotted at the beginning for learning the system. Then predictions on the time frame for programming each of the functions of the microservices were used to a lot the development stages. The final stages involved the compilation of the finding of the project, and writing the report.

After the initial planning was done and the project was begun it was found however that there were large portions of the project that were not accounted for. Chief among these was the implementation of the AIML 2.0 library into python and the difficulties encountered with the desired OCR functionality. These task true time burden are shown at the bottom of the Gantt chart.

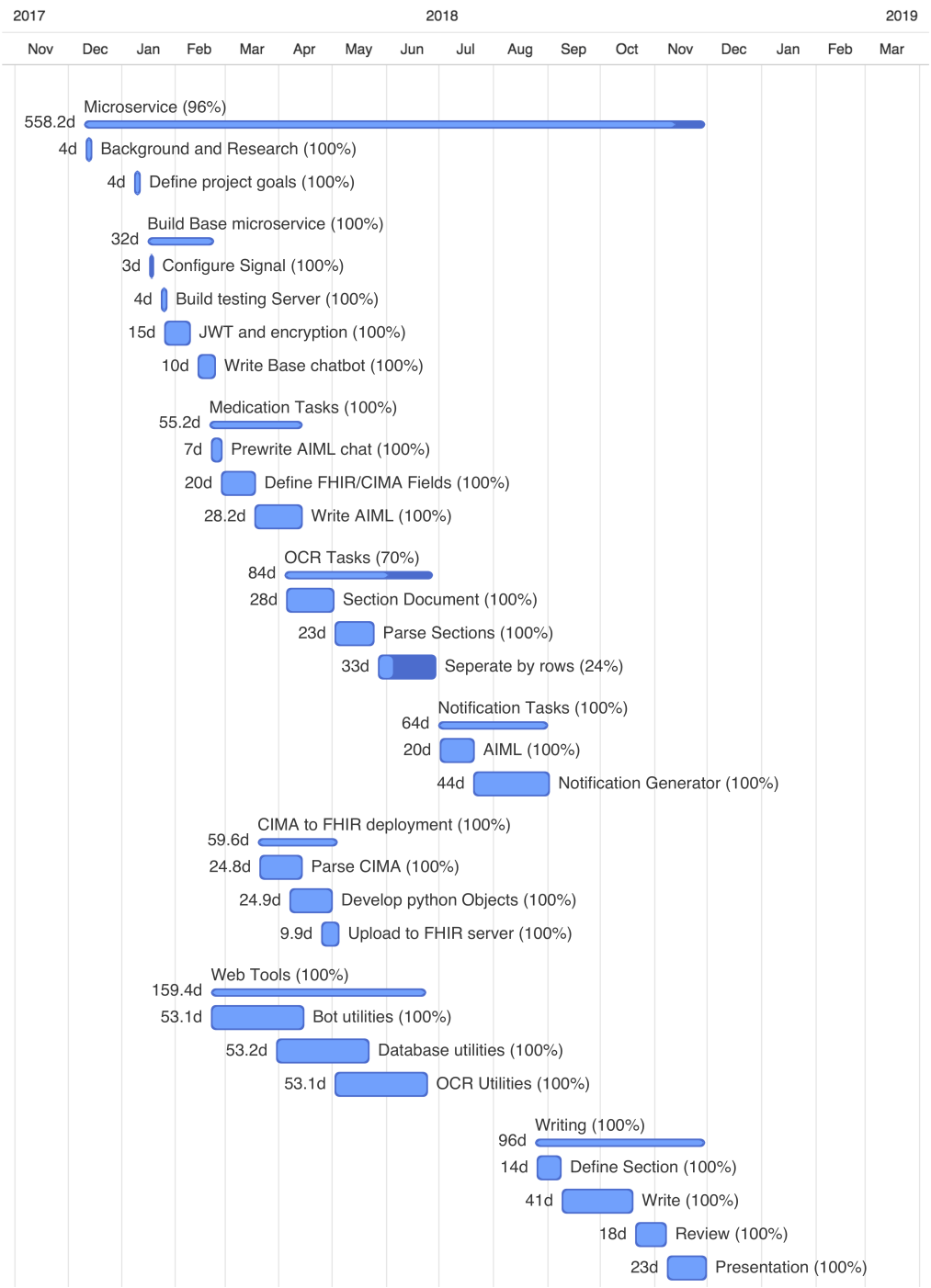

Figure B.1: Gantt Chart of the project from proposal to final presentation.

Appendix C

The final build

The following is simply a list of the dependencies that the microservice uses inside of docker. Below that is the docker build script used to build the microservice. These are provided for posterity, as a sort of installation and running guide.

```
asn1crypto==0.24.0
certifi==2018.4.16
cffi==1.11.5
chardet==3.0.4
click==6.7
cloudpickle==0.5.3
cryptography==2.2.2
cycler==0.10.0
dask==0.18.0
dateparser==0.7.0
decorator==4.3.0
fhirclient==3.2.0
Flask==1.0.2
Flask-JWT==0.3.2
gunicorn==19.8.1
idna==2.6
imutils==0.4.6
isodate==0.6.0
itsdangerous==0.24
Jinja2==2.10
kiwisolver==1.0.1
lxml==4.2.1
MarkupSafe==1.0
matplotlib==2.2.2
netifaces==0.10.6
networkx==2.1
numpy==1.14.3
oauthlib==2.1.0
Pillow==5.1.0
pycparser==2.18
```

```
PyJWT==1.4.2
pyOpenSSL==18.0.0
pyparsing==2.2.0
python-dateutil==2.7.3
pytz==2018.4
PyWavelets==0.5.2
regex==2018.6.9
requests==2.18.4
scikit-image==0.14.0
scipy==1.1.0
six==1.11.0
socketio==0.1.3
toolz==0.9.0
tzlocal==1.5.1
Unidecode==1.0.22
urllib3==1.22
uWSGI==2.0.17
Werkzeug==0.14.1
```

Dockerfile

This is the installation file for the microservice in Docker. In the *Install Python Requirements* section the custom and modified libraries are shown to be installed with the *-e* suffix to the *pip3 install* command. The final section preceded with *Entry point* shows the command that Docker issues to start the microservice inside the *Gunicorn*.

```
#Download base image ubuntu 16.04
FROM ubuntu:16.04
RUN apt-get update
RUN apt-get install -y uwsgi
RUN apt-get install -y python3
RUN apt-get install -y python3-pip
RUN apt-get install -y libgtk2.0-dev
RUN pip3 install --upgrade pip
RUN apt-get install -y python3-tk
# Set port
EXPOSE 5084

#Install Python requirements:
```

```
WORKDIR /root

RUN ls

ADD requirements.txt requirements.txt

RUN pip3 install -r requirements.txt

ADD /security /security

RUN pip3 install -e ../security

ADD /develop /develop

RUN pip3 install -e ../develop/OCRimg

RUN pip3 install -e ../develop/uservicetools

RUN pip3 install -e ../develop/fhirtools

RUN pip3 install -e ../develop/requests-oauthlib

RUN pip3 install -e ../develop/extaiml



ADD userver.crt /etc/ssl/PrivateCerts/userver.crt

ADD userver.key /etc/ssl/PrivateCerts/userver.key


ADD wsgi.py wsgi.py

ADD app.py app.py

ADD /WebApp /WebApp

ADD /static /static

ADD userver.conf userver.conf

ADD hapi.conf hapi.conf



ENTRYPOINT [ "gunicorn", "--bind", "0.0.0.0:84", "wsgi:app",\

             "--certfile","/etc/ssl/PrivateCerts/userver.crt",\

             "--keyfile","/etc/ssl/PrivateCerts/userver.key"]
```

Appendix D

Extensions

- showmed: Depending on arguments to this extension it displays either a single medication as referenced by the FHIR identification sent to it, or it returns the entire list of active medications associated with the user.
- oob: AIML 2.0 tag for adding out of band functions.
- microservice: tag that runs a script that announces the microservice as running./
- posologia: Able to parse the dosage that is input into computer friendly information.
- description: Used in the manual entry of a medication, it is the tag that tells the microservice to save the input description to the custom FHIR medication object
- newlist: Generates a new FHIR List object, that is referenced to the user. The list is given the status of *working* until it is confirmed by the user to be filled completely.
- newrequest: Initiate the FHIR medication request object for the input of a new prescription.

- addone
- EntInErr: Provides a tool to flag a medication as *entered in error*. All medication entries are permanently saved on the data base
- addtolist: Adds the medication request reference to the list of medications.
- mancode: Begin the process of manually entering medication. Takes medication code as string of numbers.
- manname: Manually enter the name of medication.
- mandose: Manually enter the dose of a medication.
- readcurrentMed: Get the up to date information about the current focused medication. This is an internal function used when cycling through a list of medications.
- picture: Calls the function that processes base64 encoded image files.
- br: workaround to allow for multi-line messages. Depending on platform the newline is handled differently and this function corrects compatibility issues.
- allcurrmeds: Returns reference codes for all current active medications for the patient.
- setperiodunit: Parses the users notification frequency and returns the largest unit of time given. Example: 4 per week, the period unit is week.
- showcurrnoti: Get current notification that is being modified:
- count: Counter to keep track of medication in list.
- counteradv: When medication in list is finished advances counter to next medication.

- onemed: Focus on single medication from medication list.
- setcurrfreq: Set the frequency of a notification.
- gennotification: Generate a notification by sending the notification parameters to the notifications microservice.
- oneFhirID: Takes a medication CIMA code and finds the correct unique FHIR id to reference the medication correctly.