



ugr

Universidad
de Granada

BACHELOR FINAL PROJECT

COMPUTER ENGINEERING

HOW-R-U?

Suite of e-coaches aimed to analyse human behaviour

Author

Carlos Sánchez Páez

Supervisor

Oresti Baños Legrán



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

GRANADA, ACADEMIC YEAR 2019-2020

HOW-R-U?: Suite of e-coaches aimed to analyse human behaviour

Carlos Sánchez Páez

Palabras clave: chatbot, telegram, salud, médico, asistente, coach, suite

Resumen

Hoy en día los trastornos mentales siguen siendo difíciles de tratar y diagnosticar. Además, debido al estigma que conllevan, algunos de los posibles pacientes no se sienten cómodos acudiendo a un profesional de la psicología. El objetivo de este proyecto es desarrollar una suite de e-coaches en forma de chatbots, específicamente, un chatbot de salud mental que ayude al diagnóstico prematuro de enfermedades mentales como la ansiedad y la depresión. Esta suite será modular, de forma que cada especialista pueda tener un agente conversacional asociado (psicólogo, nutricionista, etc.). Además, se le incorporará funcionalidad para que sea útil no solo para doctores, sino para analistas de datos, que contarán con la información recopilada de todos los pacientes.

Se plantea como trabajo futuro el uso de este sistema en un entorno en el que el asistente iniciará una conversación regularmente con el paciente y le hará una serie de preguntas definidas por su doctor. Tras ello, el especialista podrá acceder a una interfaz web en la que consultará y analizará las distintas respuestas proporcionadas por el paciente.

HOW-R-U?: Suite of e-coaches aimed to analyse human behaviour

Carlos Sánchez Páez

Keywords: chatbot, telegram, health, doctor, assistant, coach, suite

Abstract

Nowadays mental illnesses are still difficult to be treated and diagnosed. Moreover, they are associated to a stigma, so some possible patients do not feel comfortable when requesting profesional help. The aim of this project is to develop an e-coaches suite as chatbots, specifically a mental health chatbot that would help to the premature diagnostic of mental illnesses such as anxiety and depression. This suite will be modular so that every specialist can have an associated conversational agent (psychologist, nutritionist, etc.). Moreover, it will also support data analysts, special doctors that will be able to consult all the information from all the patients.

It is proposed the use of this system in an environment where the assistant will regulary start a conversation with the patients, making them questions defined by their doctors. After that, the specialists will be able to access a web interface where they can consult and analyze the answers given by patients.

Yo, **Carlos Sánchez Páez**, alumno de la titulación Graduado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 25613096C , autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Carlos Sánchez Páez

Granada a 01 de julio de 2020

D. **Oresti Baños Legrán**, Profesor del Departamento Arquitectura de Computadores de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***HOW-R-U?: Suite of e-coaches aimed to analyse human behaviour***, ha sido realizado bajo su supervisión por **Carlos Sánchez Páez**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expide y firma el presente informe en Granada a 01 de julio de 2020

El supervisor:

Oresti Baños Legrán

Agradecimientos

A mi familia, porque sin ellos nunca habría llegado tan lejos.

A Cristina, por su apoyo incondicional en todo momento.

A Casandra, por haber estado conmigo desde el principio.

A Oresti, por haberme apoyado en momentos difíciles y ser una gran fuente de motivación y conocimiento.

Al los buenos profesores que he encontrado durante mi carrera académica, por haber sabido enseñar y motivar.

Al mis amigos y amigas, por haberme ayudado siempre que lo he necesitado.

Contents

1	Introduction	7
1.1	Context	7
1.2	Motivation	7
1.3	Objectives	8
1.4	Structure	8
2	State of the art	9
2.1	Number of publications related to chatbots	9
2.2	Health application domains	10
2.3	Conversational agents types and communication formats	11
2.4	Technology	12
3	Methodology	13
3.1	Design	13
3.1.1	Requirements	13
3.1.2	Architecture	15
3.2	Implementation	18
3.3	Submodules: models and helpers	19
3.4	Conversational agent	22
3.5	Web interface	30
3.6	Docker	46
4	Environment setup	49
4.1	Development environment	49
4.2	Production environment	50
4.2.1	Pre-deployment: Telegram bot creation	50
4.2.2	System deployment	50
5	Discussion	54
6	Planification and budget	55
6.1	Planification	55
6.1.1	Budget	55
6.1.2	Scalable environment	56
6.1.3	Non scalable environment	56
6.1.4	One single instance	57
7	Conclusions	58
	References	59

1 Introduction

1.1 Context

Mental disorders are very common in our society. According to (Bandelow & Michaelis, 2015), almost 34% of the population suffer from anxiety at least once in their lives. They are difficult to be diagnosed and properly treated. Most intervention programs do not last as much as they should and doctors have a very high workload, so patients need to wait for a long time before being advised by a doctor. In addition, although having a mental disease is very common, it is still a taboo subject whose stigma makes them even more difficult to be diagnosed and treated, as (Davies, 2000) states.

In addition, people do not go to the doctor at the point of need, so there is not a continuous traceability of patients' health status. This causes a lot of relevant data not to be retrieved. This data could be suitable to perform more specific and personal medical analysis. Having a coach that regularly interacts with the person with a mental disorder as well as extracts data from their responses would be able to show the specialist their evolution over the time.

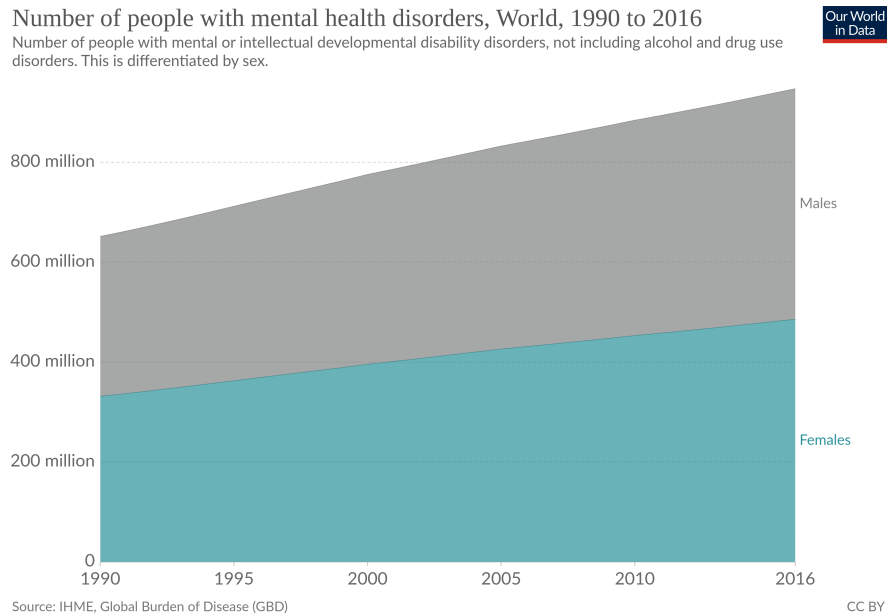


Figure 1: Number of people with mental health disorders
Reprinted from (Ritchie & Roser, 2018).

1.2 Motivation

Nowadays, technology is becoming increasingly integrated into our lives. Specifically, smartphones have become a daily basis used tool. People read the news, check the weather, chat with their relatives and friends, etc., using smartphones. Modern chat applications (such as (*Telegram*, 2013)) allow people to create bots that can interact with them like if there was a person on the other side. These chatbots are growingly becoming popular amongst people because they cover many functionalities, from tracking shipments to playing games and reminding tasks. Telegram bots do not require another app to be installed in the client's phone, so the client can save storage space. Chatbots can evolve

into conversational-agents-as-sensors are systems that consist on virtual agents that interact with users to retrieve their status in an ubiquitous, continuous, customizable and implicit or explicit way.

As we previously discussed, mental disorders are taboo, so having a bot you could talk with about how your day went, feelings, etc., could lead to an easier way of diagnosing them because chat conversations are seen as "natural" by society. The conversational agent will ask the patient a batch of questions (previously defined by their doctor) and show the data to the specialist so that the diagnosis can be more precise.

In addition, the mental specialist-patients ratio is quite low in Spain, about 2 psychologists per 100.000 citizens, (Vicente, 2019). In the future, this lack of specialist will be even more evident, as more people will suffer from mental health disorders. In fact, by 2030 there will be approximately 2 million more adults in the UK with a mental health problem. (of Health Authorities & Confederation, 2014).

1.3 Objectives

- **Main goal:** to develop a conversational-agent-as-a-sensor which will be able to interact with a person with a disorder and ask questions defined by specialists.
- **Secondary goals:**
 - To design a graphical web interface where doctors can consult their patient's responses.
 - To design a flexible and scalable architecture to add functionality to the system.
 - To design an architecture based on containers to host the different system modules.
 - To implement a system that covers the previous goals.
 - To test a beta version of the assistant in real people and analyse the retrieved data as well as target audience's feelings about it.

1.4 Structure

The first chapter of this project offers an introduction to the context in which the main goal is intended to be developed as well as an analysis of related papers. This section is divided into four main categories (health application domains, conversational agents types, communication format and architecture) and a research about the popularity of chatbots.

The second one (**Methodology**) contains the proper requisites design, architecture, programming language and frameworks that are intended to be used in the system development, as well as its the most important modules and some code snippets to illustrate its functionality. The third one (**Environment Setup**) explains how to deploy both a development and a production environment for HOW-R-U. The fourth chapter (**Discussion**) analyses the system by performing a SWOT analysis. Finally, the last one (**Conclusions**) analyses the initial objectives of the project and if they were achieved or not.

2 State of the art

2.1 Number of publications related to chatbots

Several queries were performed on *scopus.com* to check the number of articles per year related to chatbots and their applications in health. The results obtained were the following ones:

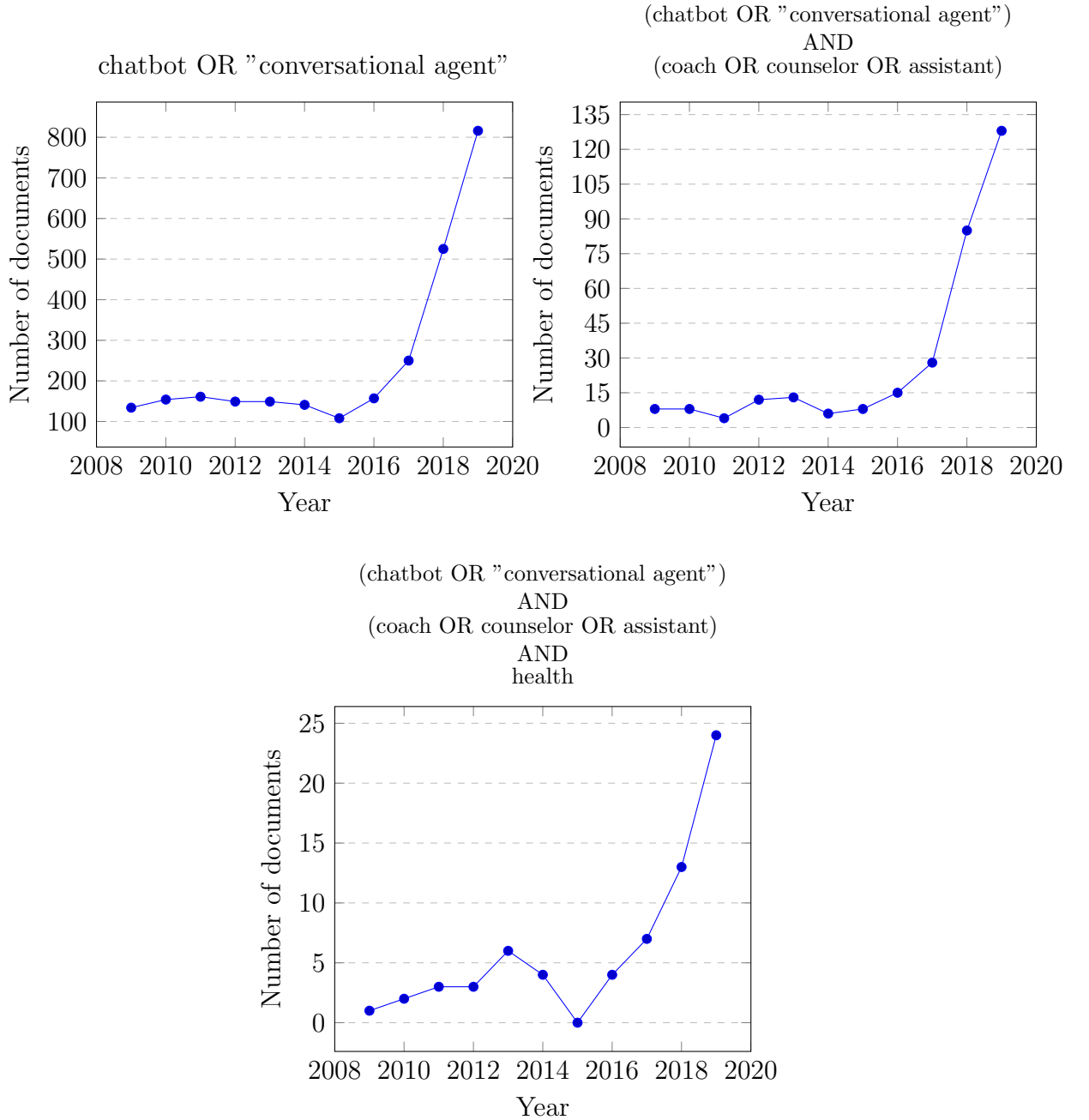


Figure 2: Search results of different queries performed in *scopus.com*.

We can observe that, in the last 5 years, all of them show a considerable increasing number of articles and papers related to virtual assistants.

2.2 Health application domains

Chatbots can be applied to many health domains. For example, (Alesanco, Sancho, Gilaberte, Abarca, & García, 2017) proposes an agent that provides help to perform dermatological medicament prescriptions. (Bennet Praba, Sen, Chauhan, & Singh, 2019) takes care about nutrition by proposing a chatbot to motivate patients to maintain weight. Another example of conversational interface oriented to health is (Falala-Sechet, Antoine, Thiriez, & Bungener, 2019). This paper shows a virtual agent that can establish a dialogue with patients to help them go through difficult situations that can affect their psychological status.

Chat agents can not only be classified according to the area they are applied to, but also according to the target group. We can find chatbots to help students learn, such as (López, Eisman, & Castro, 2008) proposal, that offers a simulation of a real patient that presents several symptoms so that students have to interview them and make a diagnostic as a doctor would make in real life. Other interesting project is (Shorey et al., 2019), that intends to improve communication between patients and hospital workers by creating simulated clinical scenarios with virtual patients so that undergraduates can practise with close to real situations instead of purely theoretical ones.

There are also chatbots aimed to help doctors do their work. For example, (Ni, Lu, Liu, & Liu, 2017) offers an agent that interviews sick people before the doctor does. Mandy elaborates a diagnostic based on several questions about patients' symptoms and sends it to the doctor so that can save time. As professional treatments do not last as much, complements such as (D'Alfonso et al., 2017) are used. They offer a long-term treatment after the professional one so that the patient's progress is not lost.

Virtual agents can be also aimed to help patients. For example, (Harilal, Shah, Sharma, & Bhutani, 2020) is a counsellor that helps patients with depression by establishing empathetic conversations with them. (Roca, Hernández, Sancho, García, & Alesanco, 2020) suggests an agent that offers help to people by reminding the intake of their medication, sending notifications to caregivers, providing summaries, etc.,

Health coaches can be really helpful. For example, (Bresó, Martínez-Miranda, Botella, Baños, & García-Gómez, 2016) proposes a platform to identify and provide early intervention for symptoms of depression and suicide, with a usability of 75.7% and an accuracy of 70.9%. Another example is (Hirano, Ogura, Kitahara, Sakamoto, & Shimoyama, 2017), which offers preventive therapy for mental healthcare. The audience's mental health punctuation significantly improved after using the application. Moreover, the usage rate and the number of suggested actions carried out was high. That indicates that people found the app useful. By last, (Ring, Bickmore, & Pedrelli, 2016) suggests an agent that responds to users' affective states during virtual therapy sessions. Facial expressions and voices are measured during the session and 70% of users affirmed that they felt understood by the agent. 50% of them stated that the agent evoked emotional responses in them during the interactions.

2.3 Conversational agents types and communication formats

Virtual assistants can be classified attending to their main goal. Coaches help the user to get what he wants. For example, (Hudlicka, 2013) presents a coach that assists people by motivating them to meditate. The paper results showed that using the agent was more effective in helping users to establish a regular meditation habit. (Guo, Liu, & Chen, 2020) offers another coach that assesses users in their workouts to improve their postural hygiene by using wearables as sensors. Conversational agents can also be counselors, which help the user to identify and solve problems. The research carried out by (Drislane et al., 2020) proves that counselors can be really helpful when trying to reduce alcohol and drugs consumption amongst users. (Yasavur, Lisetti, & Rishe, 2014) suggests a virtual therapist that gives brief speech interventions (3-5 minutes) aimed to let a person's alcohol consumption up and make him or her be aware of the problem.

They can use different methods to communicate with their audience: text, voice or both. If we focus on speech-enabled conversational agents, we can distinguish (Maharjan, Bækgaard, & Bardram, 2019), which proposes a smart speaker that regularly asks its audience about their sleep quality, mood and physical activity. By analysing the given responses and their tone, volume and intonation the agent is able to extract useful information useful for detecting markers of a possible mental illness. In text-based chatbots, the user's input can be free (the agent will translate it to an ontology to understand it) or limited (a custom keyboard with fixed options will be shown or the agent will ask the user to enter a valid option). Finally, the most popular virtual assistants are multimodal (they both admit text and speech communication).

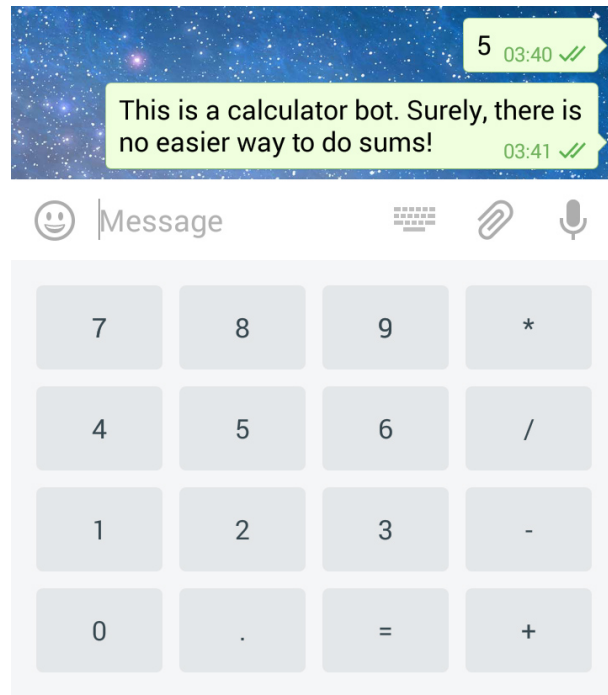


Figure 3: Example of a custom keyboard for a conversational agent
Reprinted from <https://core.telegram.org/bots>.

For a more extensive review of the state of the art in this area the reader is directed to (Montenegro, da Costa, & da Rosa Righi, 2019).

2.4 Technology

The chatbot market consists on a variety of chat applications, mainly aimed to smartphone users. However, the wide majority of them count with a web interface or a desktop application so they can be also used on personal computers. After performing some research over the most famous chat applications that allow building conversational agents, the following table was elaborated:

Platform	Number of daily active users	Free API to implement chatbots
(<i>Facebook Messenger</i> , 2008)	1.66 billion	✓
(<i>Whatsapp</i> , 2009)	1.5 billion	✗
(<i>WeChat</i> , 2011)	1.083 billion	✓
(<i>Telegram</i> , 2013)	0.2 billion	✓
(<i>Kik</i> , 2010)	0.015 billion	✓
(<i>Discord</i> , 2015)	0.014 billion	✓
(<i>Slack</i> , 2013)	0.012 billion	✓
(<i>Viber</i> , 2010)	0.008 billion	✓
(<i>Line</i> , 2012)	0.00723 billion	✓

Table 1: Comparison between different chat applications (2019).

We can see that Whatsapp, which is the most used messaging app, has no free API to develop conversational agents (the one it has is oriented to business). As opposed, the rest of the widely used chat applications offer an API for programmers to build chatbots in an easy way.

A key aspect on social networks is the privacy they ensure. (Kosinski, Stillwell, & Graepel, 2013) shows that Facebook’s likes can be helpful to predict people’s sensitive attributes such as age, gender or happiness. (Rastogi & Hendler, 2017) affirms that Whatsapp’s end-to-end encryption methods are not secure enough as metadata can reveal private information. On the other hand, (Sutikno, Handayani, Stiawan, Riyadi, & Subroto, 2016) states that Telegram provides more privacy protection than other apps. One of its main features is that users can create and customize an username so that they do not have to exchange phone numbers to chat (such as in Whatsapp).

3 Methodology

3.1 Design

3.1.1 Requirements

The MoSCoW prioritization method (Clegg & Barker, 1994) will be used to classify the requirements of this project. MoSCoW is an acronym for "Must have, Should have, Could have and Won't have", categories in which requirements are divided.

- *Must have* requirements. They are critical to the success of the project.
 - The conversational agent must ask questions defined by doctors to the patients.
 - The conversational agent must show a custom keyboard to the patient with the possible answers to the questions.
 - The application must allow doctors and patients enrollment and account deletion.
 - The application must allow the doctors to download a file with their patients answers between two given dates.
 - The system must support modularity.
- *Should have* requirements. These are important requirements, but not necessary for the system release.
 - The system should be under a CC BY-NC-SA 4.0 (*Creative Commons*, 2001) license.
 - The application should allow the doctors to add, modify and delete questions and their respective answers.
 - The application should allow the doctors to create public questions so that other doctors can assign them to their patients.
 - The conversational agent should be able to ask questions following a configurable schedule.
 - The application should show interactive charts to doctors with their patients data.
- *Could have* requirements. Desirable requirements that could improve user's experience or satisfaction. Will be included if there is time at the end of the development.
 - The system could implement a numeric priority to manage the order in which questions are asked.
 - The system could allow the doctors to configure the frequency of each question (only once, daily, weekly, monthly, etc.,).
 - The system could be translated to other languages.
 - The application could support password change functionality for doctors.
 - The application could support two factor authentication for doctors.
 - The conversational agent could support groups.

- The system could allow doctors and patients to delete all their data from the system.
- The agent could offer an option to the patients to view and modify their profile data.
- The applications could allow doctors to assign a question to all their patients when creating it.
- The system could support more timezones.
- *Won't have* requirements. They are inappropriate or the least important ones, so they are not included in the project.
 - The application won't be cross-platform.
 - The system won't share the retrieved data with third parties.

3.1.2 Architecture

The requirements that have been listed in the previous section are summarized in the following architecture:

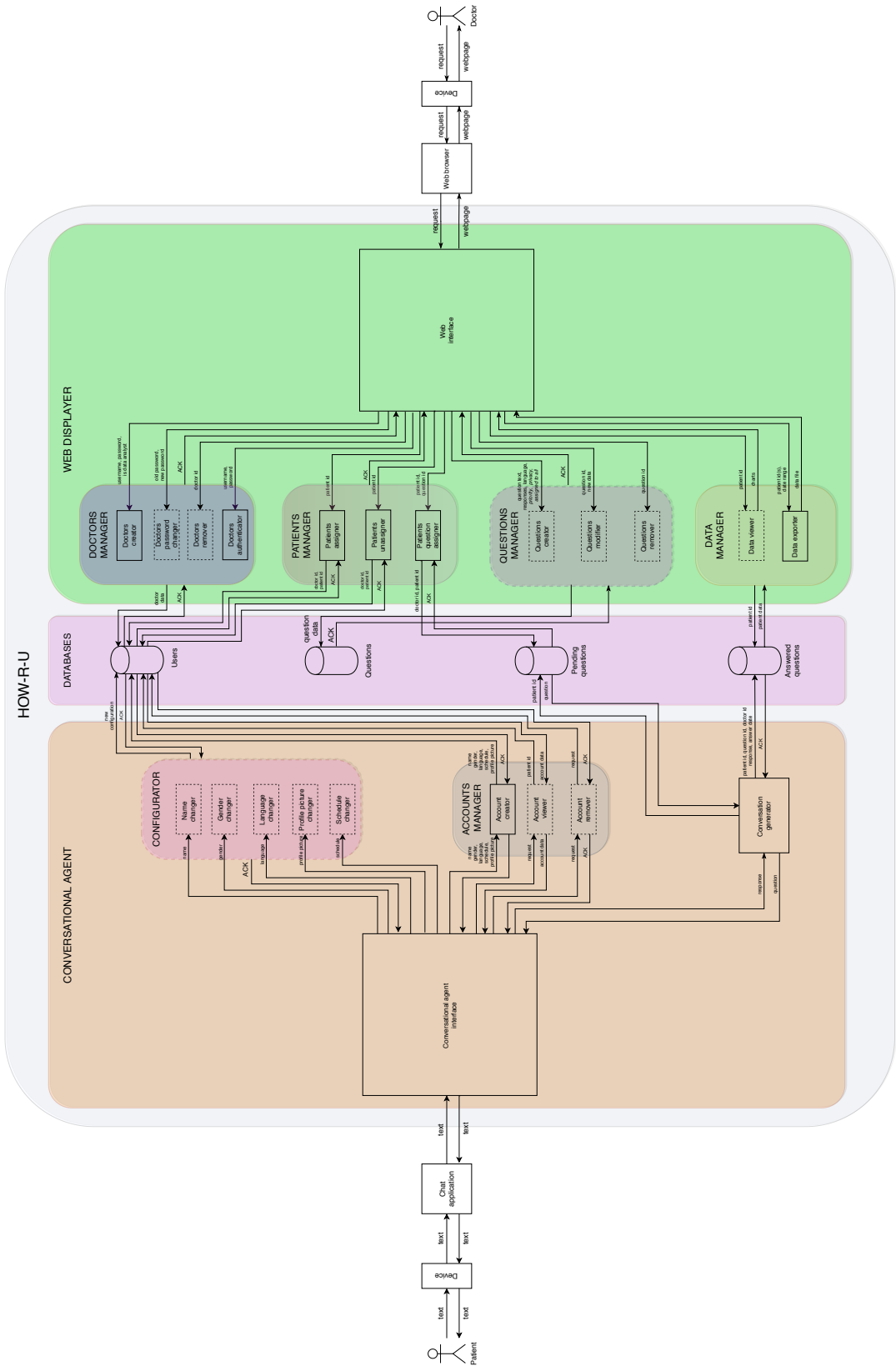


Figure 4: System architecture. Created using *diagrams.net* (*diagrams.net*, 2020).

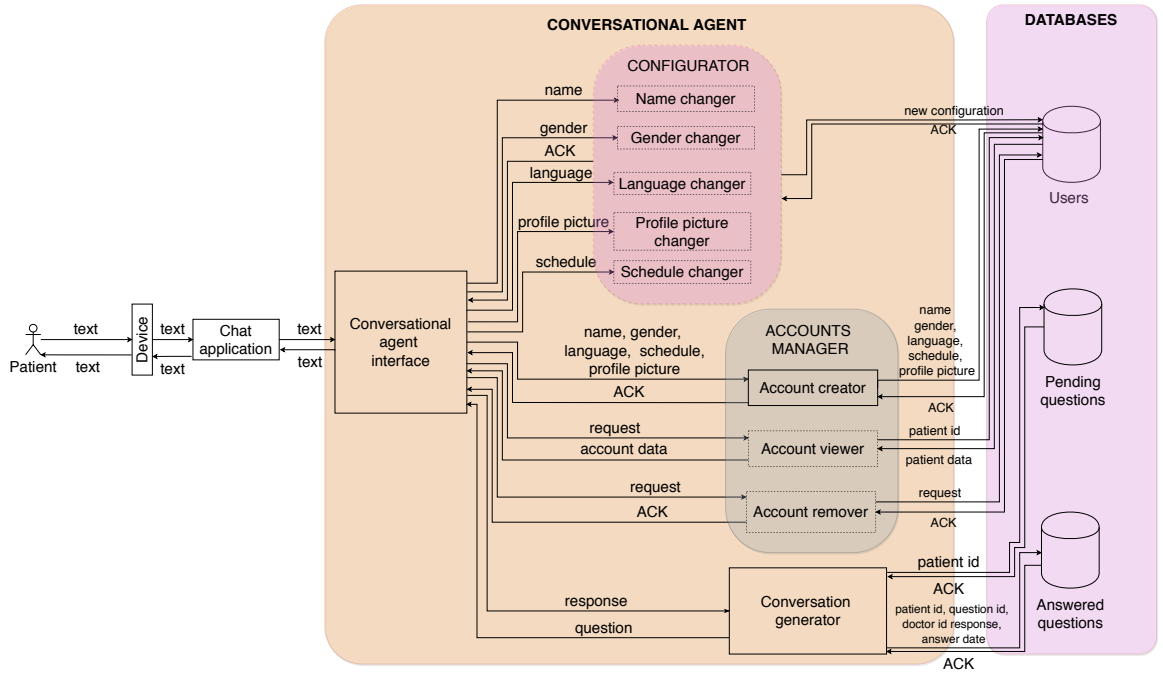


Figure 5: System architecture (conversational agent and databases).

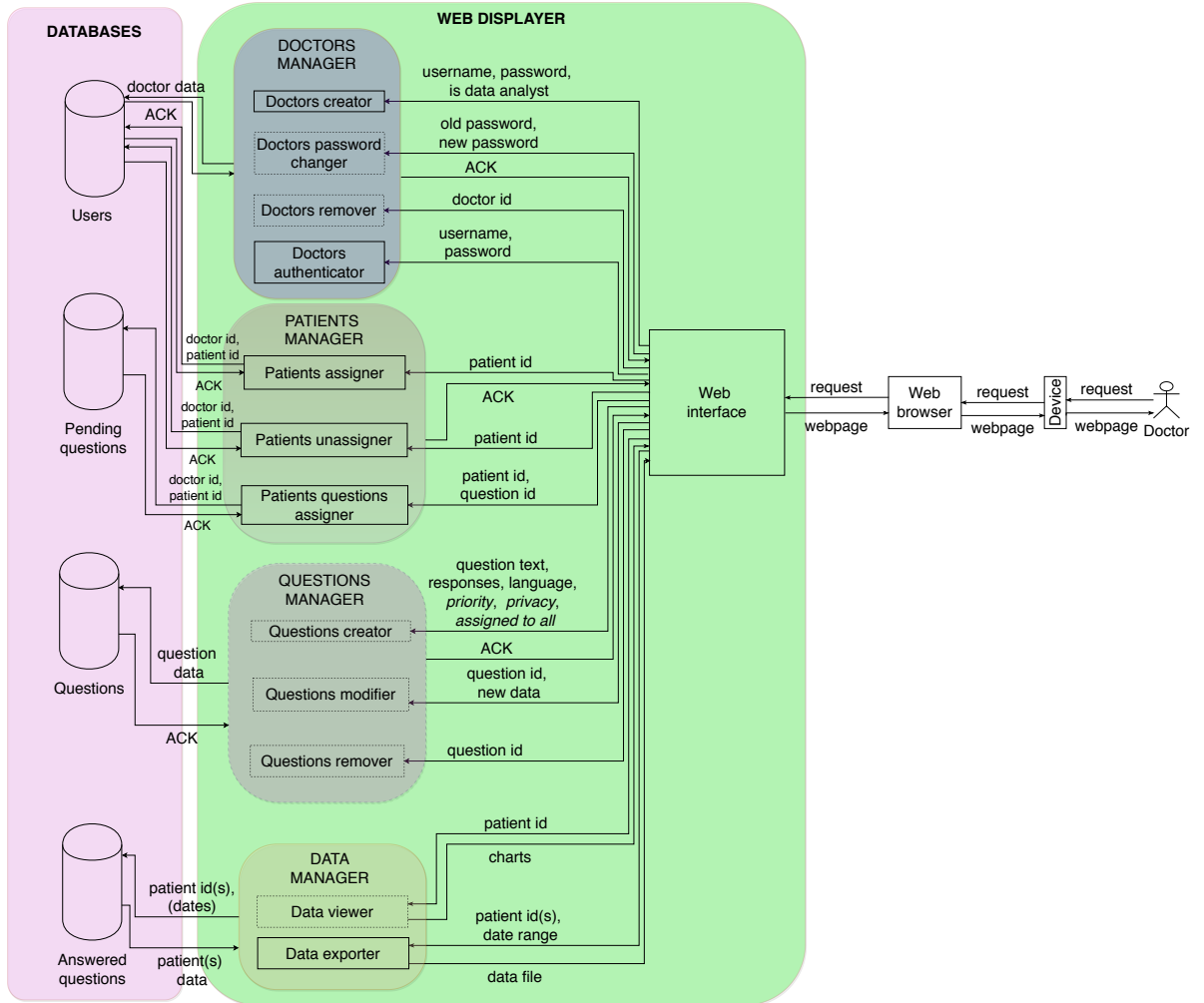


Figure 6: System architecture (web interface agent and databases).

Italic text and dashed lines represent optional features (should and could requirements), whereas continuous lines refer to mandatory ones (must). The architecture is divided into three main modules: conversational agent, databases and web interface. The conversational agent is in charge of interacting with patients by asking them questions, allowing them to change their settings, etc., On the other hand, the web interface provides the corresponding information for doctors (retrieves answers, associated patients, etc.,) as well as lets them create new questions, associate patients, etc., Finally, the databases store the corresponding data.

HOW-R-U is structured in four main databases. The users database contains doctors and patients information and their corresponding configurations. Patient objects stored in this database have properties such as name, username, chat id, gender, language, schedule and profile picture, whereas Doctor entries contain attributes like username, password, whether the doctor is a data analyst and session information. Questions database contain Question entries, consisting on question text, possible responses, question language, priority, frequency and author. Pending questions database holds entries that link a doctor, a question and a patient. Moreover, they also include a flag that determines if the question (assigned by the corresponding doctor) is being answered right now by the related patient. Finally, Answered questions database contains entries that also link a patient, a doctor and a question. They include two more properties: response given by the patient and answer date.

The conversational agent is formed by three submodules, triggered by different events. The first one is the accounts manager, including three components. The first one is the Account Creator, in which the conversational agent will ask the user the language in which the interface will be shown, their name, gender, profile picture and schedule for questions. This event is triggered when the patient interacts with the agent for the first time. The second one is the Account Viewer, which shows the patient all the details about their account. Finally, the Account Remover allows the patient to completely delete all their data from the system in an unrecoverable way. These two last components are triggered by an specific command sent by the patient.

The second module is the configurator. It is also triggered by a command. This interface shows a menu with the available options: change name, gender, language, profile picture or schedule. All these actions have a fallback command to cancel the current operation. When an operation is completed, the corresponding data is retrieved, modified or deleted from the Users database.

Finally, the last module is the Conversation Generator, that is triggered when the schedule time comes. The system will check the pending questions database to check if the user has to answer at least a question today. If the previous condition is fulfilled, the question will be prompted, along with a custom keyboard showing the different answers. Everytime a question is answered, an AnsweredQuestion entry containing the patient id, question id, doctor id and answer date is created and stored in the corresponding database. When all the pending questions have been answered, this flow terminates and the agent thanks the user. If there are no questions to ask, the chatbot will inform the patient.

Last but not least, the web interface interface contains four submodules. The first one is the doctors manager, which permits accounts creation, password change, authentication and deletion. The second one is the patient manager, which lets doctors to assign and unassign patients to its account as well as assign or unassign questions to them. The third one is the questions manager, which allows doctors to create, modify and delete questions. Finally, the data manager offers doctors the possibility to observe charts created with the

patient's answers along the time or download a file containing all these data between two dates.

3.2 Implementation

The main software that has been used to implement this project is be the following:

- Python 3.6.10 (Van Rossum & Drake, 2009) as the main programming language for both the web interface and the agent. The following libraries will also be used:
 - *python-telegram-bot* to develop the conversational agent.
 - *ujson* to create and read JSON files in a fast way.
 - *pytz* for timezone management.
 - *psycopg2* to connect python with the database management system.
 - *pillow* to work with images.
- Django 3.0.6 (*Django*, 2020) as the framework to develop the web interface.
- PostgreSQL 12.3 (*PostgreSQL*, 2020) as the database management system.
- Nginx 1.17 (*NGINX*, 2019) as the reverse proxy to redirect the requests from users to the web interface.
- Docker 19.03.6 (*Docker*, 2020) as the platform where the developing environment is built on.
- Telegram (*Telegram*, 2013) as the platform where the conversational agent will offer its services to the patients.

HOW-R-U will be structured in the following way:

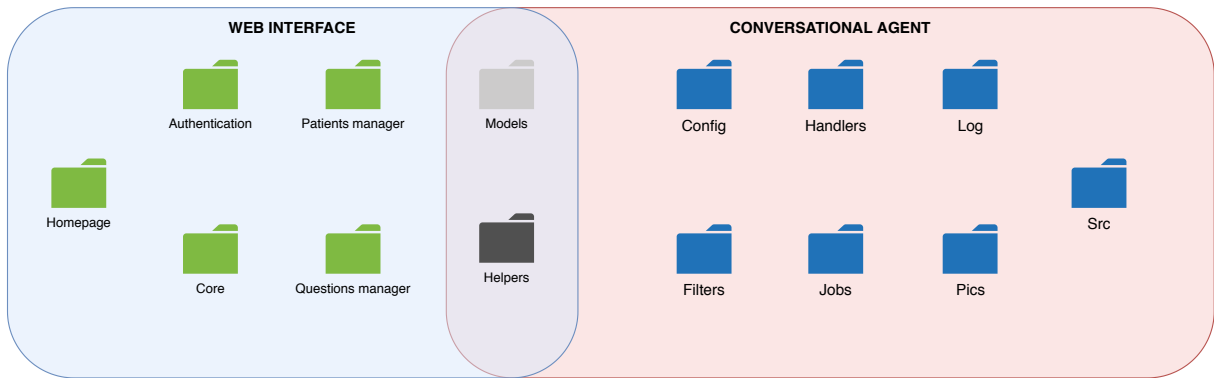


Figure 7: HOW-R-U modules structure. Created using *diagrams.net* (*diagrams.net*, 2020)

In addition, the system will be prepared to be compatible with Docker in order to deploy a HOW-R-U development environment by including a folder with *Dockerfiles*, *docker-compose* and configuration files.

3.3 Submodules: models and helpers

Models will be a submodule shared by both the web interface and the conversational agent. It will contain the definition of the different system classes. Django ORM will be used in both of them to simplify database and model operations. To add django to the conversational agent, a custom *manage.py* with the system settings (such as database details and models specification) has been created:

```
#!/usr/bin/env python
import json
import sys
import django
ROUTES_FILE_PATH = '/etc/howru/cfg/routes.json'
with open(ROUTES_FILE_PATH) as routes_file:
    json_file = json.load(routes_file)
    NAME = json_file['name']
    USER = json_file['user']
    PWD = json_file['password']
    HOST = json_file['host']
    PORT = json_file['port']
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': NAME,
        'USER': USER,
        'PASSWORD': PWD,
        'HOST': HOST,
        'PORT': PORT,
    }
}

INSTALLED_APPS = [
    'howru.models',
    'django.contrib.contenttypes',
    'django.contrib.auth'
]
from django.conf import settings
settings.configure(
    DATABASES=DATABASES,
    INSTALLED_APPS=INSTALLED_APPS,
    USE_TZ=True
)
django.setup()
```

Listing 1: Custom *manage.py* file to use Django ORM in the agent. Based in <https://stackoverflow.com/questions/45595750/use-django-orm-outside-of-django>

HOW-R-U models will follow the class diagram below:

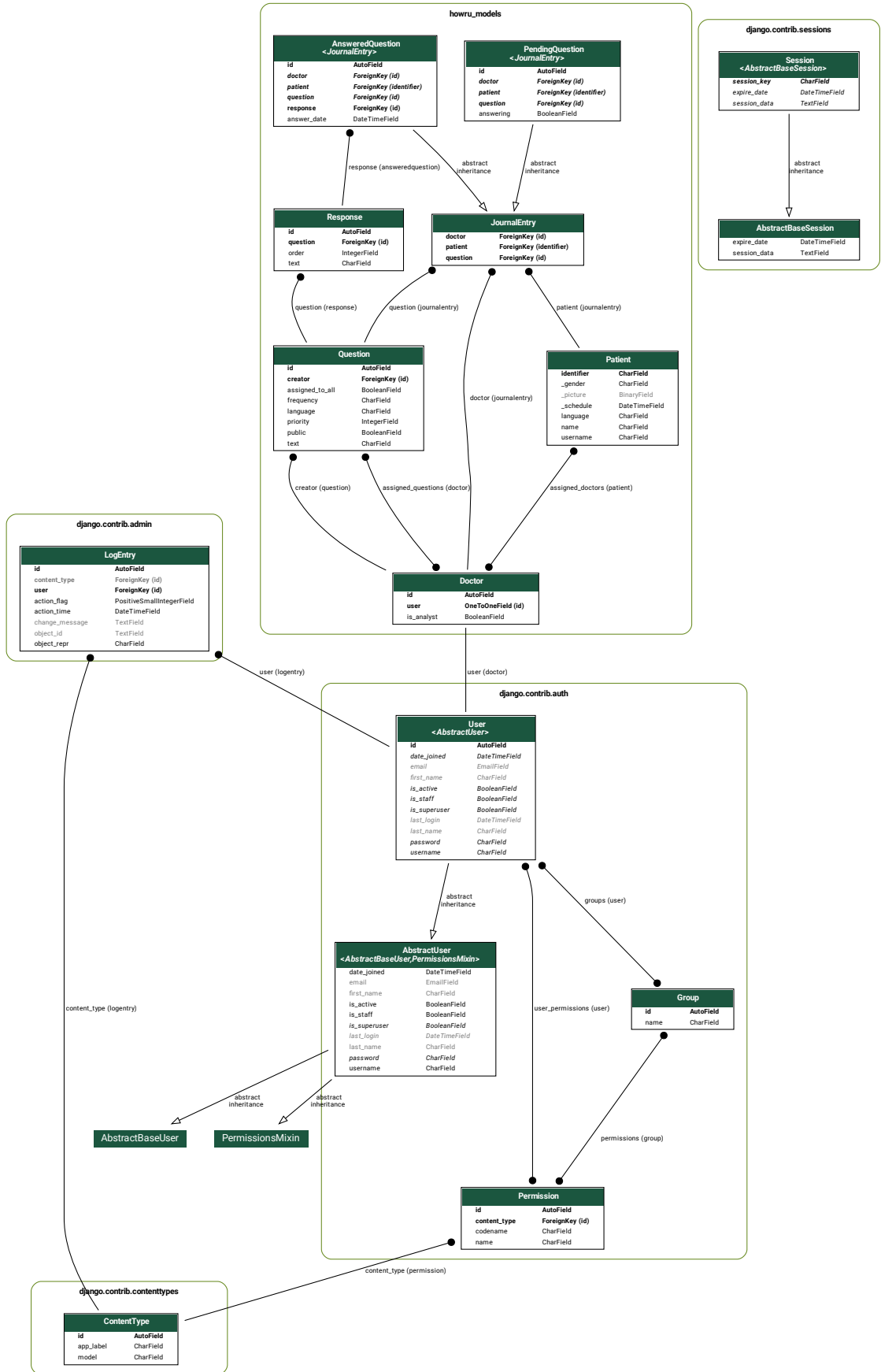


Figure 8: System class diagram. Generated by *django-extensions* (Django, 2020).

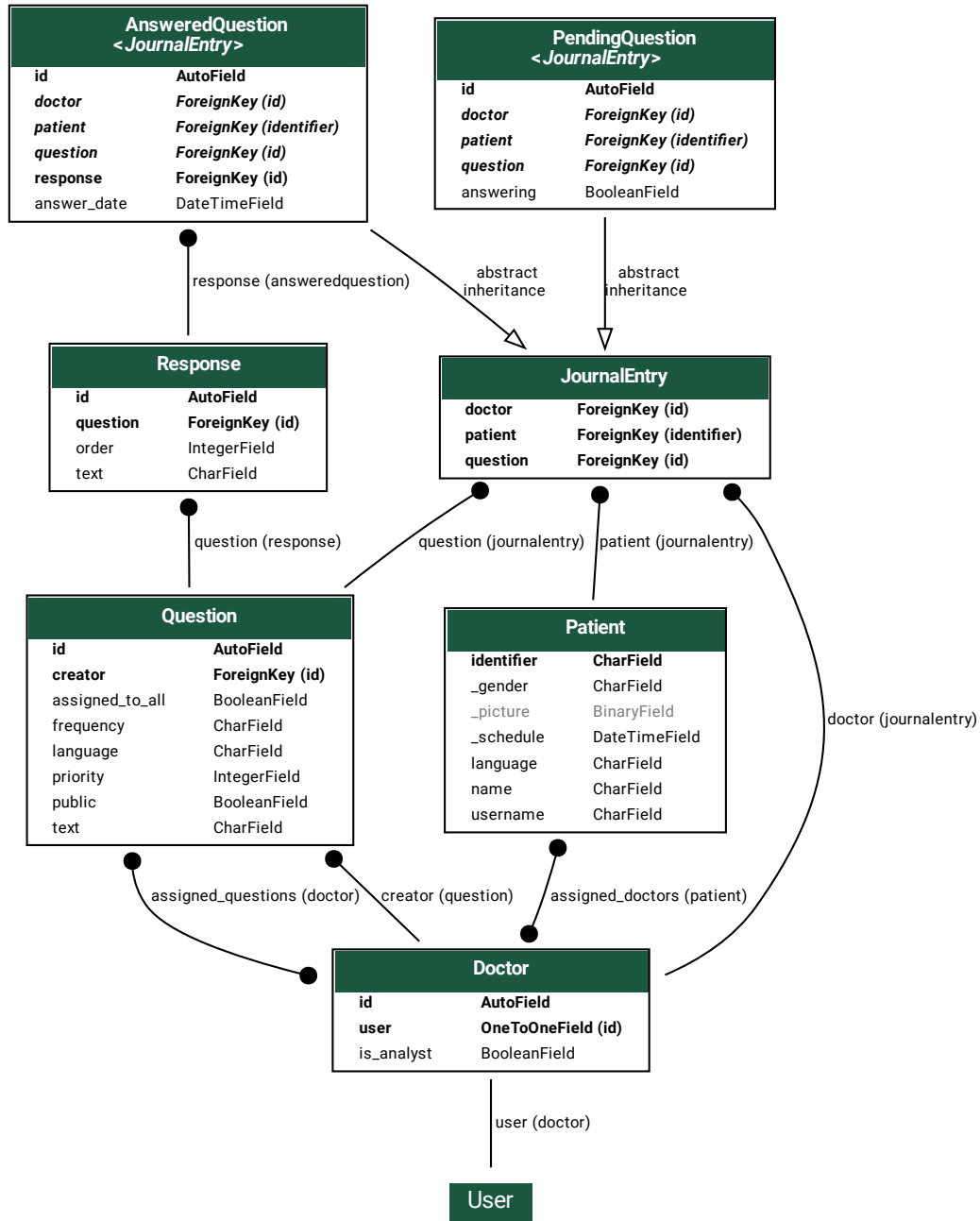


Figure 9: System class diagram (HOW-R-U classes).

As we can see, there are six main models in *howru_models*: Doctor, Patient, Question, Response, Pending Question and Answered Question (both inherit from JournalEntry, an abstract model). A Doctor has a one-to-one relationship with an user, a *Django* pre-defined model which takes care of all authentication, permissions, logs and session logic.

The other submodule, helpers, allows coding and decoding flag emojis from country codes and managing time conversions between UTC (the main time standard in the world) and the timezone in which the system is configured.

3.4 Conversational agent

The conversational agent module contains seven submodules: *config*, where the secret token is stored and messages, a file which contains all the messages the bot can send in English and Spanish. The bot secret token is a string which contains an unique identifier that links a Telegram bot user with the code it will run. This token is given by *@BotFather*, a bot used to create bots. More information about the bot creation process will be explained in the environment setup section.

The next submodule, *handlers*, defines the bot behaviour according to certain events, like sending the */start* command (the first one that is sent when an user interacts with a bot), sending the */config* one, etc., There are three main handlers:

- Start handler: asks the patients for their name, gender, language, schedule, and profile picture.
- Config handler: manages the configurator module, as well as the account viewer and remover.
- Questions handler: handles the questions prompting and answering.

Handlers are triggered by a certain event, caught by filters. For example, to catch the */start* message sent by the patient, the implemented procedure is the following:

```
GENDER, PICTURE, LANGUAGE, SCHEDULE = range(4)
@send_typing_action
def start(update, context):
    """
    Shows welcome message and asks for language
    """
    # Check that user is not registered
    try:
        patient = Patient.objects.get(identifier=update.message.from_user.id)
        logger.info(
            f'User {update.message.from_user.username} tried to register again.')
        update.message.reply_text(text=messages[patient.language]['already_exists'])
        return ConversationHandler.END
    except Patient.DoesNotExist:
        # The user should not exist in DB
        context.user_data['patient'] = Patient(name=update.message.from_user.first_name,
            ↳ identifier=str(update.message.from_user.id), username=update.message.
            ↳ from_user.username)
        logger.info(f'User {update.message.from_user.username} started a new
            ↳ conversation')
        update.message.reply_text(text=f'Hi {update.message.from_user.first_name}.
            ↳ Welcome to HOW-R-U psychologist bot.\nHola {update.message.from_user.
            ↳ first_name}. Bienvenido al bot psicologo HOW-R-U')
        update.message.reply_text(text=f'Please select a language:\nElija un idioma por
            ↳ favor:', reply_markup=keyboards.language_keyboard)
    return LANGUAGE
```

<----->

```
start_handler = ConversationHandler(
    entry_points=[CommandHandler('start', start)],
    states={
        LANGUAGE: [MessageHandler(Filters.regex(f'^({Flag.flag("es")})|{Flag.flag("gb")
            ↪ })$'), language)],
        GENDER: [MessageHandler(Filters.regex('^((Male|Female|Other|Masculino|Femenino|
            ↪ Otro)$'), gender)],
        PICTURE: [MessageHandler(Filters.photo, picture), CommandHandler('skip',
            ↪ skip_picture)],
        SCHEDULE: [MessageHandler(Filters.regex('^([0-1]?[0-9]|2[0-3]):[0-5][0-9]$
            ↪ '), schedule)]
    },
    fallbacks=[],
    name="starter"
)
```

Listing 2: Patient start callback

We can see how the **start** command is the entry point of the start handler. When the event is detected, **start** callback is invoked. This callback ensures that the patient is not already registered (if so, informs the user and finishes the conversation) and prompts a welcome message as well as a custom keyboard with flags, one for each available language. Then, it returns *LANGUAGE*, an state that shows that the agent is waiting for language selection. Start callback creates a Patient object whose data will be filled in the following procedures and stores it in context user data, a reserved storage space for each user. After language, profile picture, gender and schedule are asked, the agent finally saves the entry in the database in the following way:

```
@send_typing_action
def finish(update, context):
    """
    Saves patient in DB, assigns him/her to data_analyst, creates PendingQuestion
    ↪ entries for assigned_to_all questions and finally creates the user's
    ↪ PendingQuestionJob
    """
    patient = context.user_data['patient']
    patient.save()
    # Add patient to data analysts and assigned_to_all questions
    try:
        data_analysts = Doctor.objects.filter(is_analyst=True)
        for doctor in data_analysts:
            patient.assigned_doctors.add(doctor)
            patient.save()
            assigned_to_all = doctor.assigned_questions.filter(assigned_to_all=True)
            for question in assigned_to_all:
                pending_question = PendingQuestion(doctor=doctor, question=question,
```



```

        ↪ patient=patient, answering=False)
    pending_question.save()
    logger.info("Patient %s assigned to data_analysts", patient.username)
except:
    logger.exception("Exception while adding patient %s to data_analysts.",
        ↪ patient.username)
update.message.reply_text(messages[patient.language]['registration_ok'])
logger.info(f'Creating pending_questions job for user {update.message.from_user.
    ↪ username}')
PendingQuestionJob(context, patient)
return ConversationHandler.END

```

Listing 3: Patient saver callback

The database entry is created simply with the command *patient.save()* thanks to *django* ORM. After saving the patient for the first time, we need to assign it to all data analysts in the system and assign him/her the questions the analyst have created and configured as assigned to all. Finally, a *job* is created so that assigned questions to the patient are sent. When these procedures are completed, the callback returns *ConversationHandler.END*, indicating that the conversation has ended.

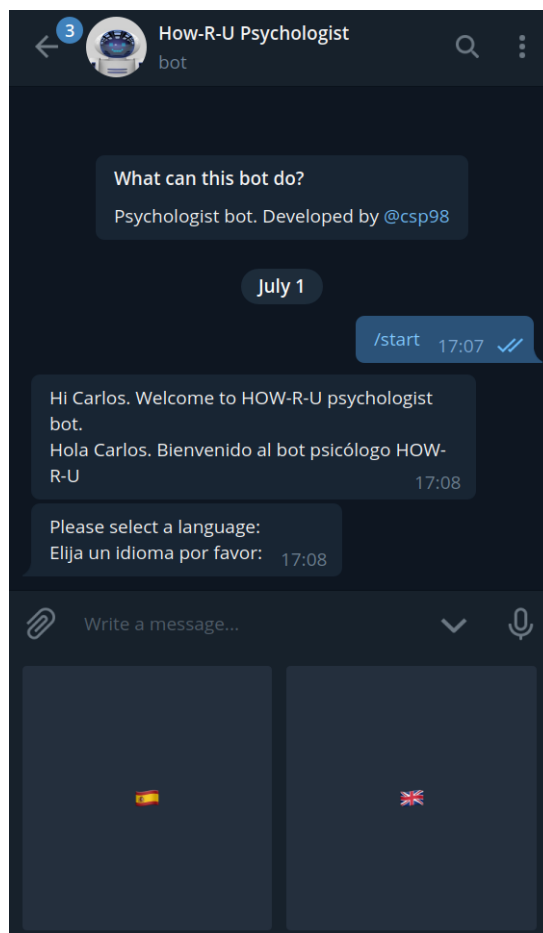


Figure 10: Agent showing the welcome message and asking for language selection.

The *config handler* is invoked when the */config* command is sent. This command ap-

pears in a pop-up menu next to the text input. The configurator callbacks are pretty similar between them. There are two types of them: *ask_for_<property>* and *process_<property>*. The first ones send a message to the patients with the old value of the property they want to change and ask them to send the new value. The second ones update the patient's database entry with the new value. For example, the schedule changer is implemented as follows:

```
@send_typing_action
def ask_change_language(update, context):
    """
    Sends old language to the user and asks for the new one
    """

    patient = context.user_data['patient']
    logger.info(f'User {update.message.from_user.username} asked to change language')
    update.message.reply_text(messages[patient.language]['current_language'] + patient
        ↪ .language)
    update.message.reply_text(messages[patient.language]['change_language'],
        ↪ reply_markup=keyboards.language_keyboard)
    return PROCESS_LANGUAGE

@send_typing_action
def process_language(update, context):
    """
    Saves the new language
    """

    patient = context.user_data['patient']
    patient.language = Flag.unflag(update.message.text)
    patient.save(update_fields=['language'])
    logger.info(f'User {update.message.from_user.username} changed language to {
        ↪ patient.language}')
    update.message.reply_text(messages[patient.language]['language_updated'])
    return config_menu(update, context)
<----->
config_handler = ConversationHandler(
    entry_points=[CommandHandler('config', config)],
    states={
        CHOOSING: [
            MessageHandler(Filters.regex(f'^(Cambiar nombre|Change name)$'),
                ↪ ask_change_name),
            MessageHandler(Filters.regex(f'^(Cambiar idioma|Change language)$'),
                ↪ ask_change_language),
            ....
        ],
        PROCESS_NAME: [MessageHandler(~is_answering_filter & ~Filters.command,
            ↪ process_name)],
        PROCESS_LANGUAGE: [MessageHandler(Filters.regex(f'^({Flag.flag("es")})|{Flag.
            ↪ flag("gb")})$'),
                                process_language)],
```

```

        .....
    },
    fallbacks=[CommandHandler('cancel', cancel),
               CommandHandler('exit', _exit)],
    name="configurator"
)

```

Listing 4: Name changer configurator callbacks

The name changer needs a specific filter. As the desired input format is text, we need to distinguish between text and a command (started by `/`). To do so, the command filter is negated (`~Filters.command`). In addition, the agent must distinguish when the name is being changed (a string is expected) and a question is being answered (a string is also being expected). To do so, a custom filter has been developed. This filter matches when a question is being answered at the moment (`is_answering_filter`). Finally, these two filters are combined using an *AND* logic operation. Last but not least, `/cancel` allows the user to cancel the current operation and go back to the configurator menu and `/exit` closes the configurator.

```

class IsAnsweringFilter(BaseFilter):
    def filter(self, message):
        """
        Checks if the patient is answering a question
        :return: True if a question is being answered, False otherwise
        """

        patient = Patient.objects.get(identifier=message.from_user.id)
        return patient.pendingquestion_set.all().filter(answering=True)
# Initialize the class.
is_answering_filter = IsAnsweringFilter()

```

Listing 5: IsAnswering filter

Finally, the questions handlers contains just one callback and no states. The callback is invoked when the user sends a message that is not a command and a question is being answered at the moment.

```

@send_typing_action
def answer_question(update, context):
    """
    Prompts user's question by querying PendingQuestion DB.
    Creates an AnsweredQuestion object.
    """

    user = update.message.from_user
    response = update.message.text
    # Get question that is being answered from DB:
    try:
        question_task = _get_pending_question_task(str(user.id))

```

```

except PendingQuestion.DoesNotExist:
    logger.info(
        f'User {user.username} id {user.id} wrote {response} while there was no
        ↳ question to answer')
    update.message.reply_text("Unrecognized command\nComando no reconocido",
        ↳ reply_markup=ReplyKeyboardRemove())
    return ConversationHandler.END
logger.info(f'User {user.username} id {user.id} answered "{response}" to question
    ↳ "{question_task.question}")
# Create answered question entry
answered_question = AnsweredQuestion(patient_id=user.id, doctor=question_task.doctor
    ↳ , answer_date=datetime.now(pytz.timezone('Europe/Madrid')), response=
    ↳ response, question=question_task.question)
answered_question.save()
# Set answering to false
question_task.answering = False
question_task.save()
return ConversationHandler.END

def _get_pending_question_task(user_id):
    """
    Obtains the question that the user is answering
    """
    return PendingQuestion.objects.get(patient_id=user_id, answering=True)

question_handler = ConversationHandler(
    entry_points=[MessageHandler(~Filters.command & is_answering_filter, answer_question
        ↳ )],
    states={},
    fallbacks=[],
    name="questions_handler"
)

```

Listing 6: Questions handler

When the user sends an answer, this callback retrieves the pending question that is being answered and creates the corresponding AnsweredQuestion entry. After that, the answering flag is set to False in the PendingQuestion object.

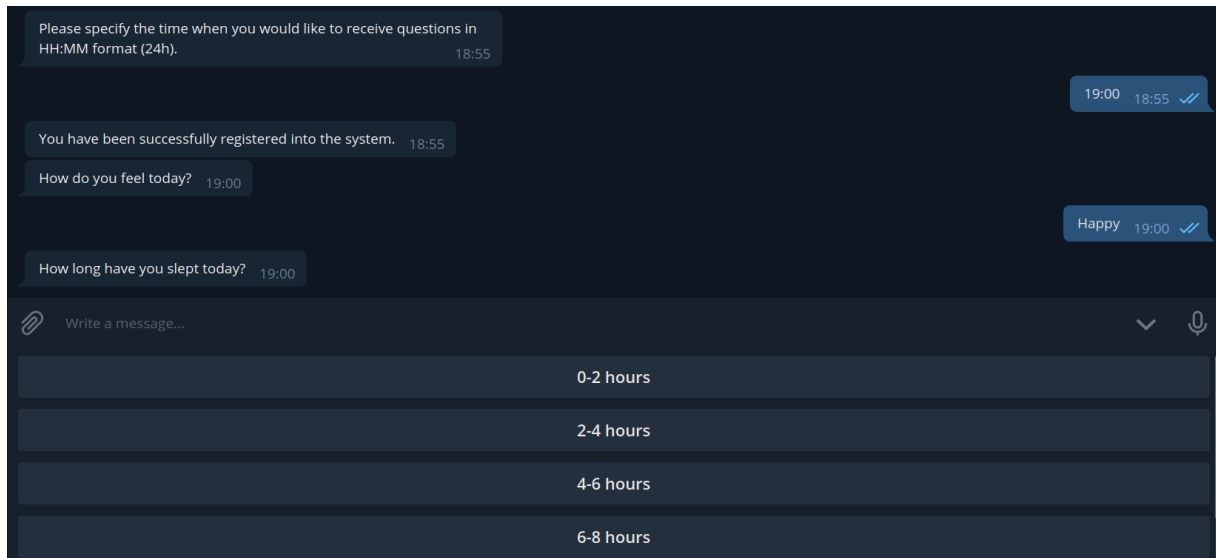


Figure 11: HOW-R-U conversational agent asking a question to a patient.

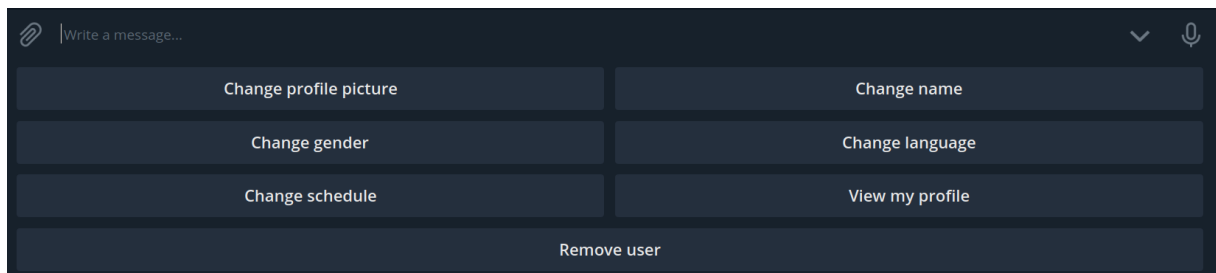


Figure 12: Configurator menu.

The next conversational agent module is *jobs*. Jobs are tasks that run a callback with a fixed frequency (daily, monthly, hourly, etc.). In this project, jobs are used to ask the patient the corresponding questions when the schedule time arrives.

```
class PendingQuestionJob(object):
    def __init__(self, context, patient):
        self.patient = patient
        self._create_job(context)
    def job_callback(self, context):
        """
        Prompts PendingQuestions to the user.
        """
        pending_questions = self._get_pending_questions()
        for task in pending_questions:
            if not self.is_question_answered(task):
                question = task.question
                task.answering = True
                task.save()
                context.bot.send_message(chat_id=self.patient.identifier, text=question.
                    ↪ text, reply_markup=keyboards.get_custom_keyboard(question.responses)
                    ↪ )
```

```

        while not self.is_question_answered(task):
            time.sleep(0.5)
        message = messages[self.patient.language]['finish_answering'] if self.
        ↪ answered_questions.today() else messages[self.patient.language]['
        ↪ no_questions']
        logger.info(f'User {self.patient.username} answered all the questions')
        context.bot.send_message(chat_id=self.patient.identifier, text=message,
        ↪ reply_markup=ReplyKeyboardRemove())
        time.sleep(0.1)
    if was_configurator_running(self.patient.identifier, context):
        logger.info(f'Reopening configurator for user {self.patient.username} id {
        ↪ self.patient.identifier}')
        context.bot.send_message(chat_id=self.patient.identifier, text=messages[self.
        ↪ patient.language]['select_config'], reply_markup=keyboards.
        ↪ config_keyboard[self.patient.language])

def _create_job(self, context):
    context.job_queue.run_daily(callback=self.job_callback, time=self.patient.schedule
    ↪ , name=f'{self.patient.identifier}_pending_questions_job')

```

Listing 7: PendingQuestion job

When a PendingQuestionJob instance is created, a daily job is established (running at the patient schedule time). The job callback queries the PendingQuestions database and select the ones that should be answered (depending on their frequency and last answer) and orders it by priority, so that questions with lower priority are asked first. Then, it creates a custom keyboard with one button per response and sends the patient the question text as a message. Finally, when all questions have been answered, the job informs the user and opens the configurator if it was open. Sleep times have been added to make the process smoother.

pics folder is a temporary directory where profile pictures are stored and then deleted after encoding them in base-64 to be uploaded to the database. *log* contains a basic logging utility that shows useful information, such as time, process and thread. Finally, *src* contains the main file which starts the bot service:

```

def main():
    logger.info("Started HOW-R-U psychologist")
    updater = Updater(token=bot_config.TOKEN, use_context=True)
    dispatcher = updater.dispatcher
    handlers = [start_handler, config_handler, question_handler]
    # Add handlers and error callback to dispatcher
    for handler in handlers:
        dispatcher.add_handler(handler)
    dispatcher.add_error_handler(error_callback)
    updater.start_polling()
    updater.idle()

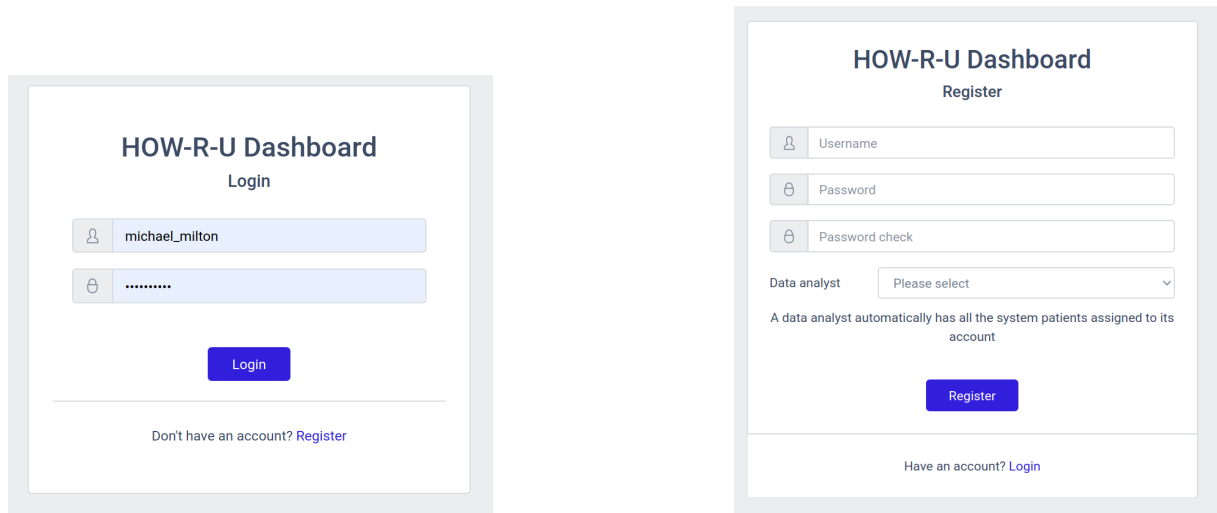
```

Listing 8: Bot file, the one that starts the service

3.5 Web interface

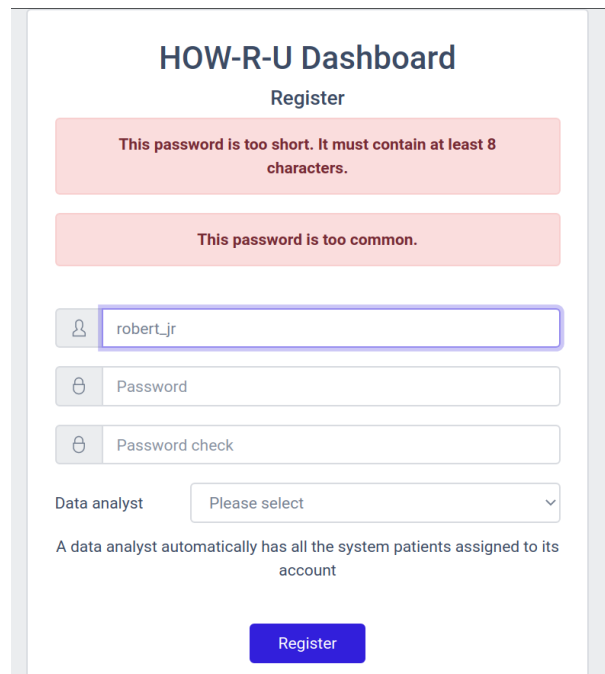
HOW-R-U web interface is a *django* project based on (*Appseed CoreUI Django Dashboard*, 2020) that is structured into four main applications: app (the homepage), questions manager, patients manager and authentication (create users, modify passwords and login/logout functionalities).

The first page that is shown to the user is the login interface, with text boxes for user and password fields. The user can also register in the system with a button.



The figure shows two side-by-side screenshots of the HOW-R-U Dashboard. The left screenshot is the 'Login' page, featuring a title 'HOW-R-U Dashboard' and a subtitle 'Login'. It has two input fields: 'Username' with the value 'michael_milton' and 'Password' with masked characters '.....'. Below these is a blue 'Login' button. At the bottom, there is a link: 'Don't have an account? [Register](#)'. The right screenshot is the 'Register' page, also titled 'HOW-R-U Dashboard' with the subtitle 'Register'. It has three input fields: 'Username', 'Password', and 'Password check'. Below these is a dropdown menu for 'Data analyst' with the value 'Please select'. A note states: 'A data analyst automatically has all the system patients assigned to its account'. There is a blue 'Register' button. At the bottom, there is a link: 'Have an account? [Login](#)'.

Figure 13: Login and register pages.



The figure shows a screenshot of the HOW-R-U Dashboard 'Register' page. At the top, it says 'HOW-R-U Dashboard' and 'Register'. There are two red error messages: 'This password is too short. It must contain at least 8 characters.' and 'This password is too common.'. Below these are three input fields: 'Username' with the value 'robert_jr', 'Password', and 'Password check'. Below these is a dropdown menu for 'Data analyst' with the value 'Please select'. A note states: 'A data analyst automatically has all the system patients assigned to its account'. There is a blue 'Register' button.

Figure 14: Register form validations.

The register form has several validations, such as password length, strength, etc.,. These validators are provided by *django auth* application. The user can also choose to be a *data analyst*, a special user that does not have to add patients to their account

manually. Everytime a patient registers into the system is automatically assigned to all data analysts. This feature is useful to collect statistics.

```
def register_user(request):
    if request.method == "POST":
        form = SignUpForm(request.POST)
        if form.is_valid():
            form.save()
            # If the doctor is an analyst, assign him/her all the patients in the system
            analyst = form.cleaned_data['is_analyst']
            form.instance.doctor.is_analyst = analyst
            if analyst:
                form.instance.doctor.patient_set.set(Patient.objects.all())
            form.save()
            username = form.cleaned_data.get("username")
            raw_password = form.cleaned_data.get("password1")
            user = authenticate(username=username, password=raw_password)
            login(request, user)
            return redirect("/")
        else:
            form = SignUpForm()
    return render(request, "accounts/register.html", {"form": form})
```

Listing 9: Method to register an user in the system

After logging in to the system, the homepage is shown. It is divided into three main sections. The first one shows statistics about the logged in doctor: the number of associated patients, submitted questions and total answers. Below, there is a panel that shows the percentage of answers per hour and patients per gender. Finally, a top 5 with the patients that have answered the most is exhibited.

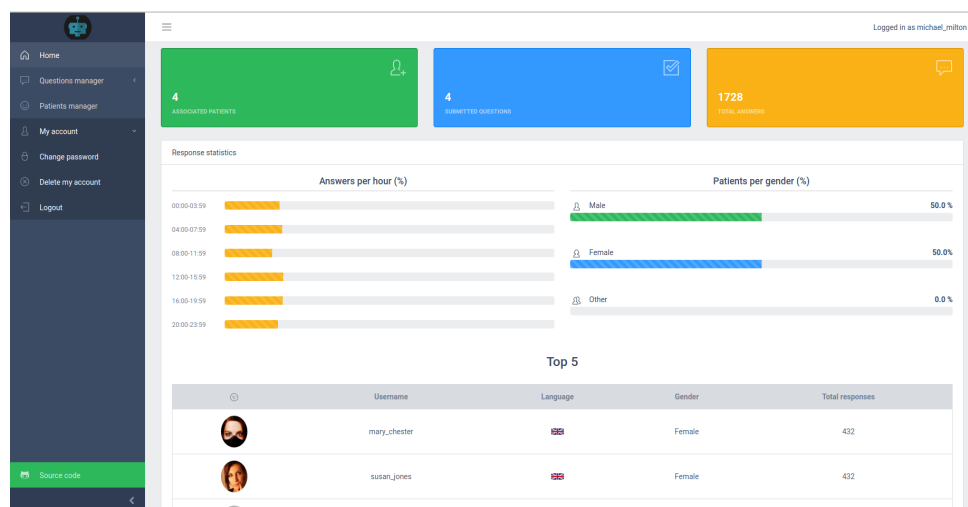


Figure 15: HOW-R-U homepage.

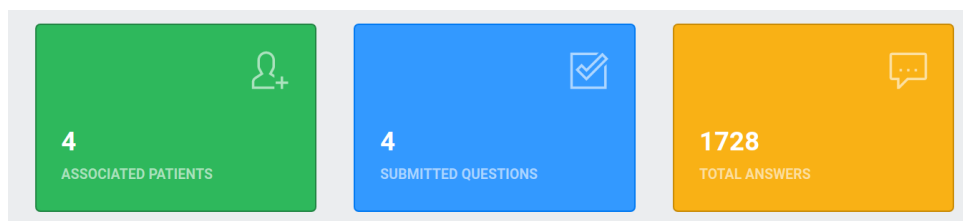


Figure 16: HOW-R-U homepage (doctor statistics).

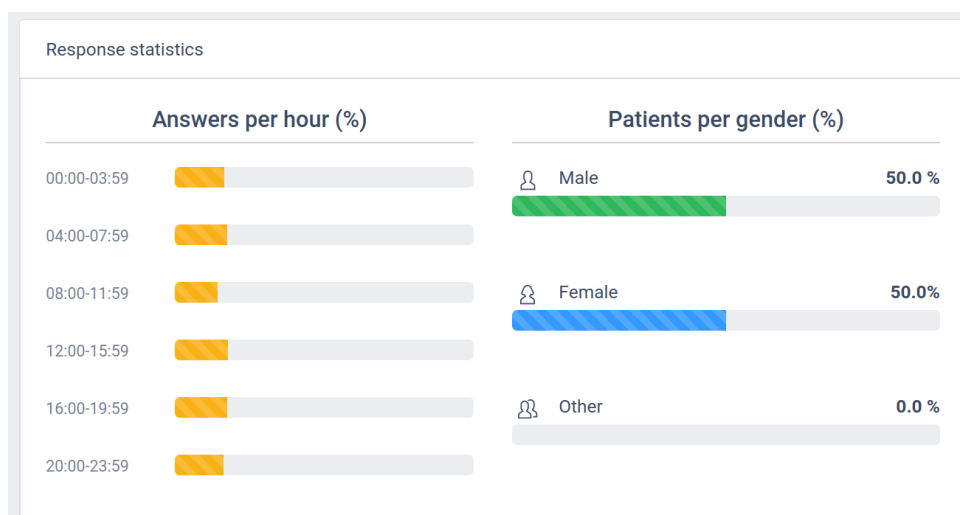


Figure 17: HOW-R-U homepage (response statistics).

Top 5









	Username	Language	Gender	Total responses
	susan_jones		Female	432
	oliver_morton		Male	432
	mary_chester		Female	432
	robert_garfield		Male	432

Figure 18: HOW-R-U homepage (top 5).

```

@login_required(login_url="/login/")
def index(request):
    """
    Shows the index page, including global parameters (top patients, number of
    ↪ associated patients, answers, gender and time percentages, etc.,)
    """

    doctor = request.user.doctor
    top_patients = get_top_patients(doctor)
    doctor_patients = doctor.patient_set
    number_associated_patients = doctor_patients.count()
    submitted_questions = Question.objects.filter(creator=doctor).count()
    total_answers = get_total_answers(doctor)
    male_percentage, female_percentage, other_percentage = get_gender_stats(doctor,
    ↪ number_associated_patients)
    answers_per_hour = get_answers_per_hour(doctor)
    context = {
        "top_patients": top_patients,
        "number_associated_patients": number_associated_patients,
        "submitted_questions": submitted_questions,
        "total_answers": total_answers,
        "male_percentage": male_percentage,
        "female_percentage": female_percentage,
        "other_percentage": other_percentage,
        "answers_per_hour": answers_per_hour
    }
    return render(request, "index.html", context)

```

Listing 10: Method to show the main homepage components

In the sidebar the user has access to the main components of the system. The first one is the **questions manager**. When the doctor clicks on it, a dropdown is opened, letting them choose public or private questions.

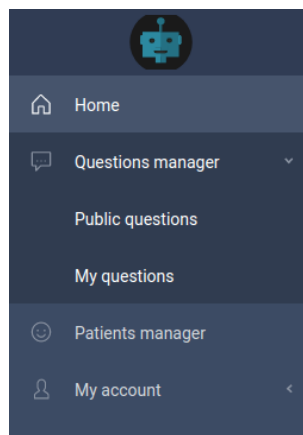


Figure 19: Questions manager dropdown.

The public question interface shows a paginated table containing all the questions that

have been incorporated into the system and marked as public. Moreover, there is a search bar and a button to add a public question to the user's questions list, so that he/she can assign it to the patients. When the user searches a question, the search term is encoded in the URL (GET request). The system has to build custom URLs when a search is being performed and the user clicks the next page. This is achieved thanks to the code below.

Question text (empty to get all questions)							
<input type="text"/>							
<input type="button" value="Search"/>							
Question text	Possible responses	Assigned to all	Frequency	Priority	Creator	Language	Actions
Do you feel sad and cry easily?	Yes No	✓	Daily	1	michael_milton (You)		<input type="button" value="Added"/>
Have you felt insecure about yourself today?	Yes No	✗	Daily	1	john_clive		<input type="button" value="Add to My Questions"/>
How do you feel today?	Sad Tired Happy Very happy	✓	Daily	1	michael_milton (You)		<input type="button" value="Added"/>
How long have you slept today?	0-2 hours 2-4 hours 4-6 hours 6-8 hours More than 8 hours	✓	Daily	1	michael_milton (You)		<input type="button" value="Added"/>

Figure 20: Public questions page.

Have you felt insecure about yourself today?	Yes No	✗	Daily	1	john_clive		<input type="button" value="Add to My Questions"/>
--	-----------	---	-------	---	------------	--	--

Figure 21: Public questions page (question created by other user).

Do you feel sad and cry easily?	Yes No	✓	Daily	1	michael_milton (You)		<input type="button" value="Added"/>
---------------------------------	-----------	---	-------	---	----------------------	--	--------------------------------------

Figure 22: Public questions page (question created by the current user).

```
@login_required(login.url="/login/")
def public_questions(request):
    """
    Shows public questions inside the system.
    """
    if 'search' in request.GET:
        term = request.GET['search']
        all_questions = Question.objects.filter(text__icontains=term, public=True).
            order_by('text')
    else:
        all_questions = Question.objects.filter(public=True).order_by('text')
    page = request.GET.get('page', 1)
    paginator = Paginator(all_questions, settings.PAGE_SIZE)
    try:
```

```

        questions = paginator.page(page)
    except PageNotAnInteger:
        questions = paginator.page(1)
    except EmptyPage:
        questions = paginator.page(paginator.num_pages)
    request.session['public_questions_page'] = page
    return render(request, 'questions_manager/public_questions.html', context={
        'questions': questions,
        'success_msg': request.session.pop('message', None)
    })

```

Listing 11: Method to show public questions page

The HTML page (template) to show the public questions is built following the pseudo-code below:

```

<table>
<thead>
<tr>
<th>Question text</th>
<th>Possible responses</th>
<th>Assigned to all</th>
<th>Frequency</th>
<th>Priority</th>
<th>Creator</th>
<th>Language</th>
<th>Actions</th>
</tr>
</thead>
{% for question in questions %}
    <tr>
        <td>
            {{ question.text }}
        </td>
        <td>
            {% for response in question.responses %}
                {{response}} <br>
            {% endfor %}
        </td>
        <td>
            {% if question.assigned_to_all %}
                check_icon
            {% else %}
                cross_icon
            {% endif %}
        </td>
        <td>
            {{ question.get_frequency_display }}

```

```

</td>
<td>
    {{ question.priority }}
</td>
<td>
    {{ question.creator.user.username }}
    {% if question.creator.user.username == request.user.username %}
    (You)
    {% endif %}
</td>
<td>
    {% if question.language == "ES" %}
    spanish_flag
    {% else %}
    england_flag
    {% endif %}
</td>
<td>
    {% if question in request.user.doctor.assigned_questions.all %}
    added_button
    {% else %}
    add_button
    {%endif%}
</td>
</tr>
{% endfor %}
</tbody>
</table>

```

Listing 12: Pseudo-code to generate public questions template page

```

@register.filter
def paginate(paginator, current):
    num_pages = settings.PAGE_SIZE
    if paginator.num_pages > 2 * num_pages:
        start = max(1, current - num_pages)
        end = min(paginator.num_pages, current + num_pages)
        if end < start + 2 * num_pages:
            end = start + 2 * num_pages
        elif start > end - 2 * num_pages:
            start = end - 2 * num_pages
        if start < 1:
            end -= start
            start = 1
        elif end > paginator.num_pages:
            start -= (end - paginator.num_pages)
            end = paginator.num_pages

```

```

    pages = [page for page in range(start, end + 1)]
    return pages[: (2 * num_pages + 1)]
return paginator.page_range

```

```

@register.simple_tag
def get_url(request, field, value):
    query_string = request.GET.copy()
    query_string[field] = value
    return query_string.urlencode()

```

Listing 13: Pagination code snippets. Retrieved from <https://medium.com/@sumitlni/paginate-properly-please-93e7ca776432>

The user's questions page is similar to the public ones. The main differences is that there are three new buttons: modify, delete and create new questions.

Question text (empty to get all questions)

Q Search

Question text	Possible responses	Assigned to all	Frequency	Priority	Privacy	Creator	Language	Actions
Do you feel sad and cry easily?	Yes No	✓	Daily	1	Public	michael_milton (You)	🇬🇧	<div>Modify</div> <div>Delete</div>
Have you noticed an appetite decrease in the last week?	Yes No	✓	Weekly	1	Private	michael_milton (You)	🇬🇧	<div>Modify</div> <div>Delete</div>
How do you feel today?	Sad Tired Happy Very happy	✓	Daily	1	Public	michael_milton (You)	🇬🇧	<div>Modify</div> <div>Delete</div>
How long have you slept today?	0-2 hours 2-4 hours 4-6 hours 6-8 hours More than 8 hours	✓	Daily	1	Public	michael_milton (You)	🇬🇧	<div>Modify</div> <div>Delete</div>

+ Create a new question

Figure 23: User's questions page.

Have you noticed an appetite decrease in the last week?	Yes No	✓	Weekly	1	Private	michael_milton (You)	🇬🇧	Modify Delete
---	-----------	---	--------	---	---------	----------------------	----	------------------

Figure 24: User's questions page (example).

The create and modify interfaces show a form with help messages to build or edit them. The main validations in these ones are that all fields are required and there should be at least two responses per question. Questions creator and modifier forms are created by inheritance from *django.forms*.

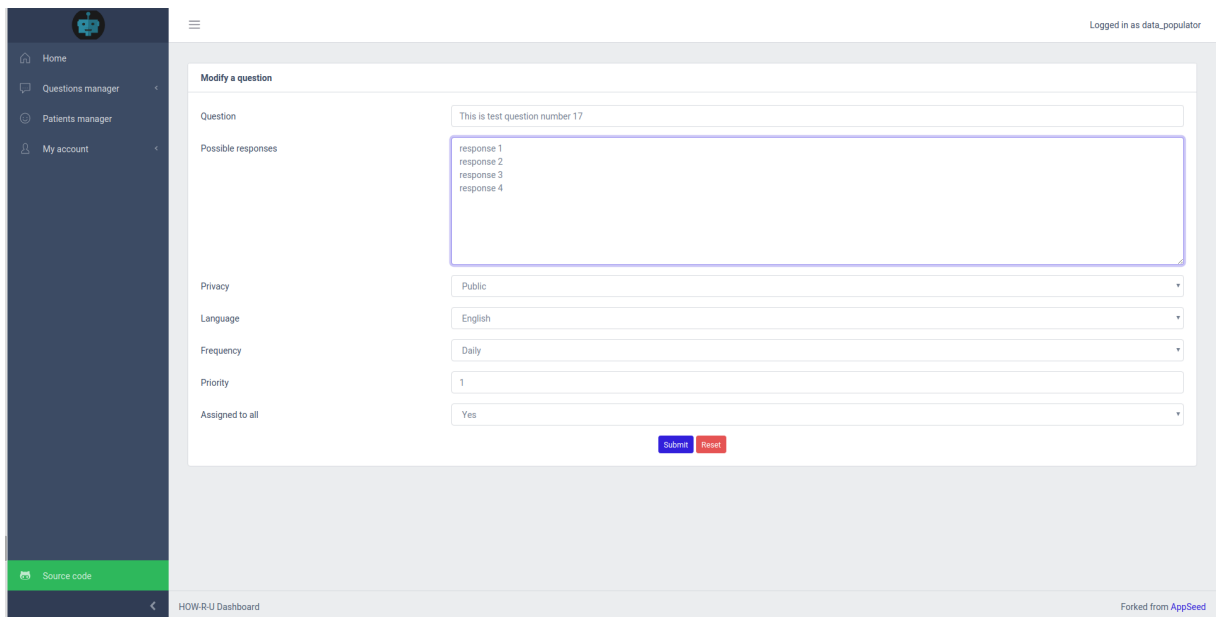


Figure 25: Questions creator and modifier.

Question	<input type="text" value="Do you feel sad and cry easily?"/>
Possible responses	<div> <input type="text" value="Yes"/> <input type="text" value="No"/> </div>
Privacy	<input type="text" value="Public"/>
Language	<input type="text" value="English"/>
Frequency	<input type="text" value="Daily"/>

Figure 26: Questions creator and modifier fields (1/2).

Frequency	<input type="text" value="Daily"/>
Priority	<input type="text" value="1"/>
Assigned to all	<input type="text" value="Yes"/>

Figure 27: Questions creator and modifier fields (2/2).

The available fields for the Questions Creator are the following ones:

- **Question text:** what should be asked to the patient.
- **Responses** (one per line). They should be ordered from bad to good so that plots are properly rendered.
- **Privacy** (public or private). Whether other users can add this question to their profile.
- **Language** (English or Spanish).
- **Frequency** (once, daily, weekly or monthly). How often the question should be asked.
- **Priority** ($[1, \infty[$). Order in which the question should be asked. Lower priority questions will be asked first.
- **Assigned to all** (yes or no). If set, the question will be automatically assigned to all the user's patients.

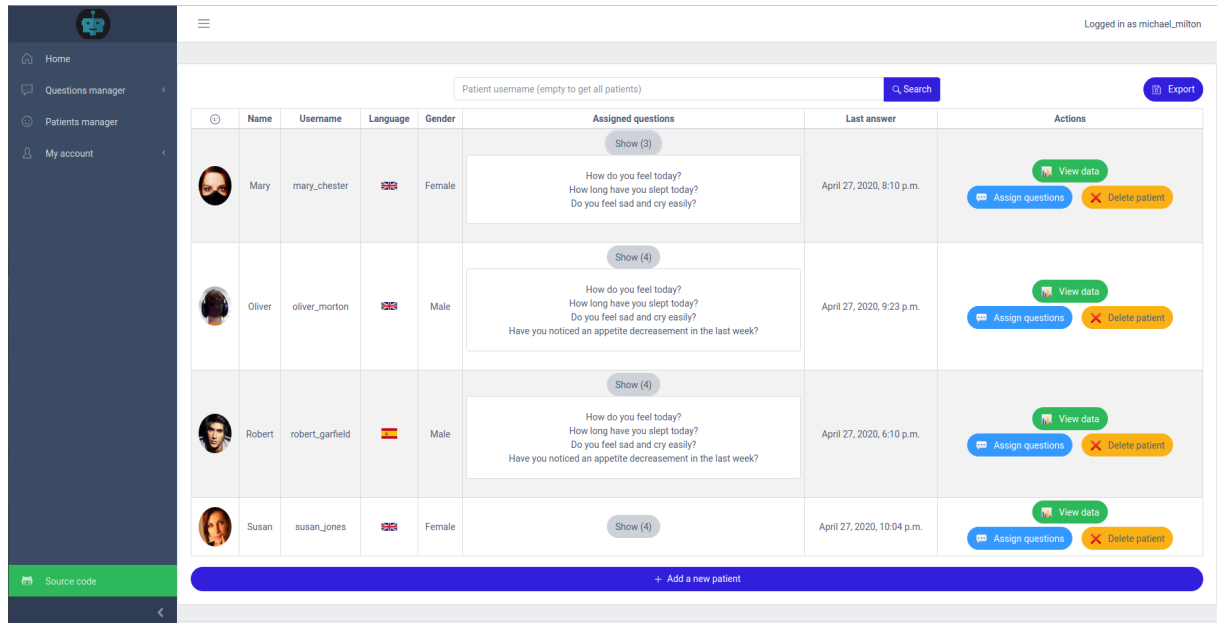
```
class QuestionForm(ModelForm):
    privacy = forms.CharField()
    to_all = forms.CharField()
    responses_field = forms.CharField()
    class Meta:
        model = Question
        fields = ["text", "public", "language", "priority", "frequency"]
    def clean(self):
        cd = self.cleaned_data
        clean_responses = cd.get("responses_field").replace('\r', '')
        response_list = list()
        for response in clean_responses.split('\n'):
            if response:
                clean_response = response.strip()
                if clean_response:
                    response_list.append(response)
        if len(response_list) < 2:
            raise ValidationError("You must specify at least two possible responses")
        self.cleaned_data['responses'] = response_list
        privacy = cd.get("privacy")
        self.cleaned_data['public'] = privacy == "Public"
        self.cleaned_data['assigned_to_all'] = self.cleaned_data['to_all'] == "yes"
```

Listing 14: Form to create or modify questions

As *public* and *assigned to all* are boolean in the models but text in the views, we have to manually analyse the data retrieved from the form. The number of answers is also validated in the *clean* method.

Finally, the questions deletion page consists of a view showing a confirmation message and two buttons, **confirm** and **back**.

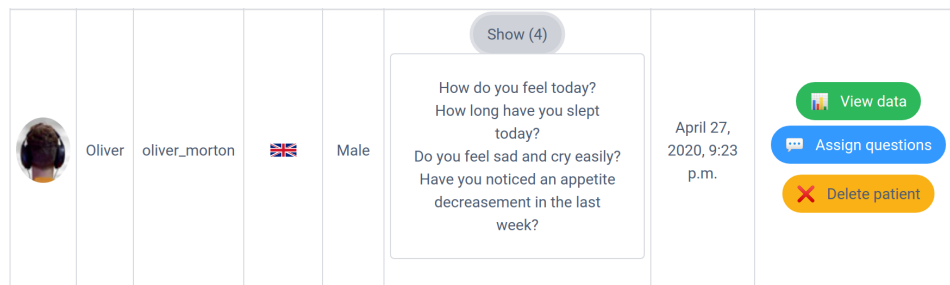
The patients manager consists of another paginated table that shows all the patients associated to the doctor. There is also a search bar, a button to add new patients and three buttons to manage each patient: view data, assign questions and delete patient from the doctor's page. Finally, there is also a export button to dump responses to a CSV file.



The screenshot shows a web application interface for a doctor's Patients manager. On the left is a dark sidebar with navigation links: Home, Questions manager, Patients manager (active), and My account. At the top right, it says 'Logged in as michael_milton'. The main area features a search bar labeled 'Patient username (empty to get all patients)' and an 'Export' button. Below is a table with columns: Name, Username, Language, Gender, Assigned questions, Last answer, and Actions. Four patients are listed: Mary, Oliver, Robert, and Susan. Each row has a 'Show' button to expand the 'Assigned questions' column. At the bottom, there is a '+ Add a new patient' button.

Name	Username	Language	Gender	Assigned questions	Last answer	Actions
Mary	mary_chester	English	Female	Show (3) How do you feel today? How long have you slept today? Do you feel sad and cry easily?	April 27, 2020, 8:10 p.m.	View data Assign questions Delete patient
Oliver	oliver_morton	English	Male	Show (4) How do you feel today? How long have you slept today? Do you feel sad and cry easily? Have you noticed an appetite decrease in the last week?	April 27, 2020, 9:23 p.m.	View data Assign questions Delete patient
Robert	robert_garfield	Spanish	Male	Show (4) How do you feel today? How long have you slept today? Do you feel sad and cry easily? Have you noticed an appetite decrease in the last week?	April 27, 2020, 6:10 p.m.	View data Assign questions Delete patient
Susan	susan_jones	English	Female	Show (4)	April 27, 2020, 10:04 p.m.	View data Assign questions Delete patient

Figure 28: Patients manager.



This table provides a detailed view of the patient entry for Oliver Morton. It shows his profile picture, name, username, language (English), and gender (Male). The 'Assigned questions' column is expanded, showing four questions. The 'Last answer' column shows the date and time of the last response. The 'Actions' column contains three buttons: 'View data', 'Assign questions', and 'Delete patient'.


	Oliver	oliver_morton	English	Male	Show (4) How do you feel today? How long have you slept today? Do you feel sad and cry easily? Have you noticed an appetite decrease in the last week?	April 27, 2020, 9:23 p.m.	View data Assign questions Delete patient
---	--------	---------------	---------	------	--	---------------------------	---

Figure 29: Patients manager (example).

The button to add new patients redirects the user to a page so that he/she can enter the patient's Telegram username. Once done, the patient will be shown in the main view.

The assign question page consists of a paginated table showing the doctor's questions list. The assignment/unassignment is performed with a button at the end of each row. These processes consist on creating or deleting a *PendingQuestion* object linking doctor, patient and question.

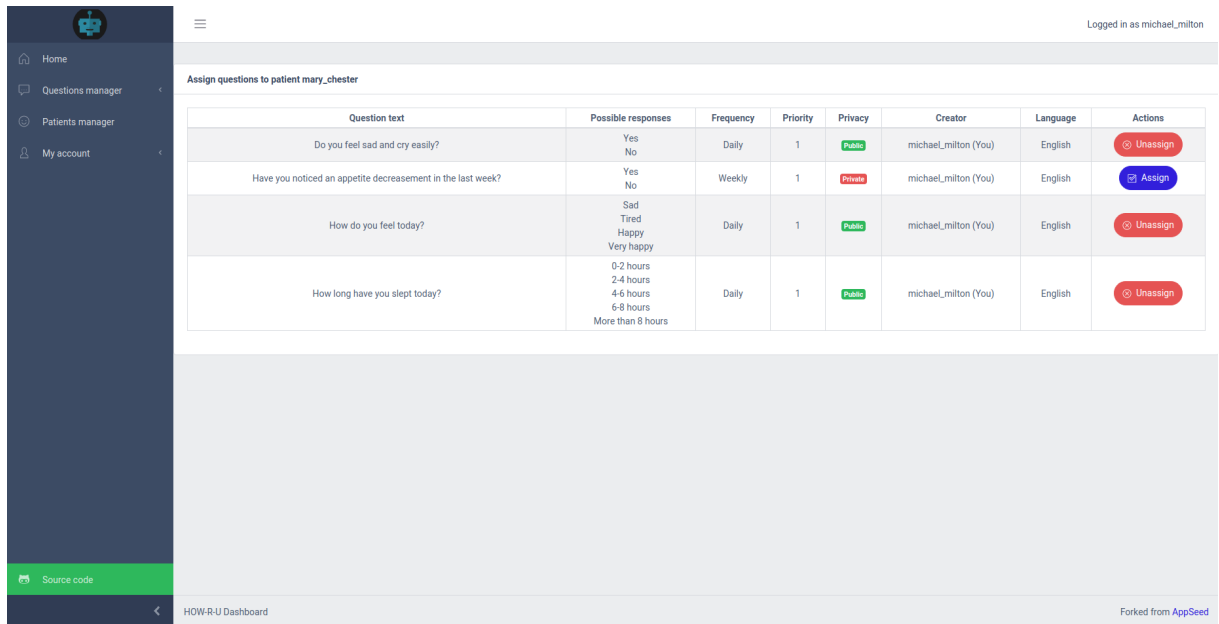


Figure 30: Patients manager assign questions page.

Have you noticed an appetite decrease in the last week?	Yes No	Weekly	1	Private	michael_milton (You)	English	Assign
---	-----------	--------	---	---------	----------------------	---------	--------

Figure 31: Patients manager assign questions page (unassigned question).

Do you feel sad and cry easily?	Yes No	Daily	1	Public	michael_milton (You)	English	Unassign
---------------------------------	-----------	-------	---	--------	----------------------	---------	----------

Figure 32: Patients manager assign questions page (assigned question).

The patients deletion button removes it from the doctor's associated patients list. After that, it can be added again. By deleting a patient, all the related *PendingQuestion* are also deleted so that questions assigned to that user by the doctor that deletes it are asked no more. However, answers are preserved so that data can be analysed in the future if it is added again. If the users wants to delete this data, he/she must delete the patient account.

The export page offers a form to select patients, start date and end date. When done, it generates a CSV file with four columns: patient username, question, answer and date.

Patient username	Question	Answer	Date
robert_garfield	How do you feel today?	Tired	2020-04-01 04:12:20
robert_garfield	How do you feel today?	Sad	2020-04-02 05:24:20
robert_garfield	How long have you slept today?	0-2 hours	2020-04-01 12:56:20
robert_garfield	How long have you slept today?	4-6 hours	2020-04-02 11:28:20
robert_garfield	Do you feel sad and cry easily?	Yes	2020-04-01 14:22:21
robert_garfield	Do you feel sad and cry easily?	No	2020-04-02 06:31:21
oliver_morton	How do you feel today?	Very happy	2020-04-01 01:45:19
oliver_morton	How do you feel today?	Happy	2020-04-02 02:59:19
oliver_morton	How long have you slept today?	More than 8 hours	2020-04-01 20:51:20
oliver_morton	How long have you slept today?	2-4 hours	2020-04-01 22:24:20
oliver_morton	Do you feel sad and cry easily?	No	2020-04-01 07:47:20
oliver_morton	Do you feel sad and cry easily?	Yes	2020-04-02 17:26:20
oliver_morton	Have you noticed an appetite decrease in the last week?	Yes	2020-04-01 21:41:20
oliver_morton	Have you noticed an appetite decrease in the last week?	Yes	2020-04-08 21:09:20

Table 2: Example data generated with the *Export* feature.

Figure 33: Export page

Finally, the *View Data* section generates plots showing patient's answers. There are two plots per question: a line chart showing the different answers amongst time and a pie one manifesting the number of times each answer has been replied. Both charts are interactive so that when the doctor hovers the mouse on them, a box is shown specifying more data about the selected section. The View Data page is also paginated and includes a search bar to filter questions.

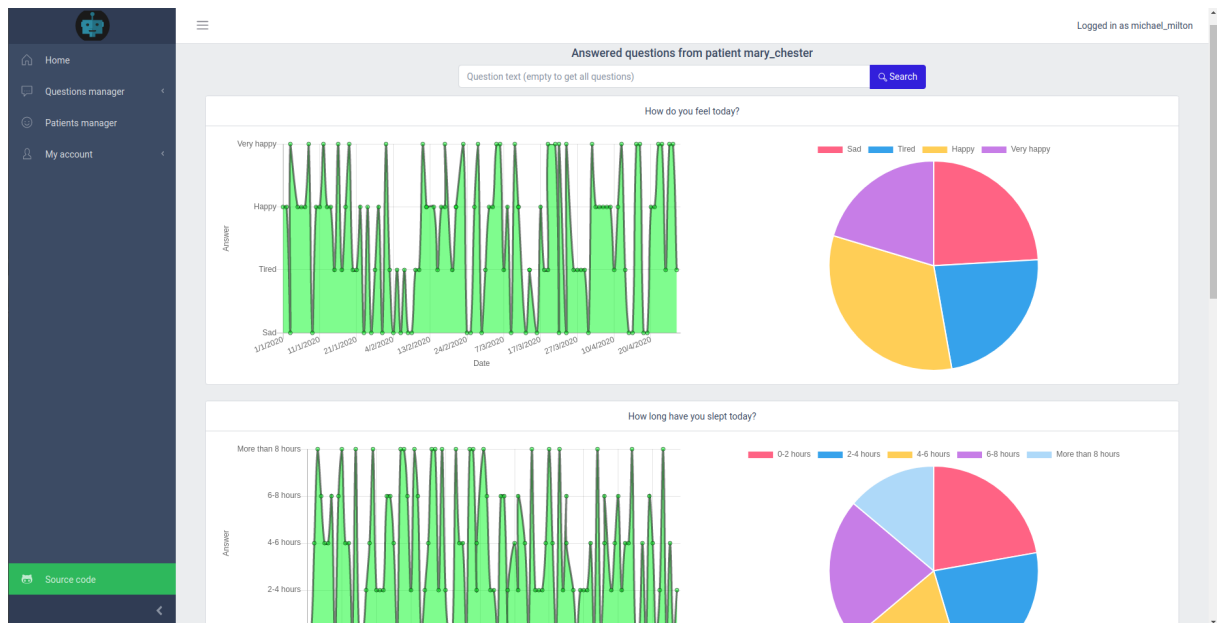


Figure 34: View data page.

Figures 34 and 35 show an extended view of the generated charts for a given question: the line one and the pie one. Charts are interactive, so that if the user hovers the mouse over them, a tooltip box will pop, showing the date in the line chart and the response and number of times that it has been answered in the pie one.

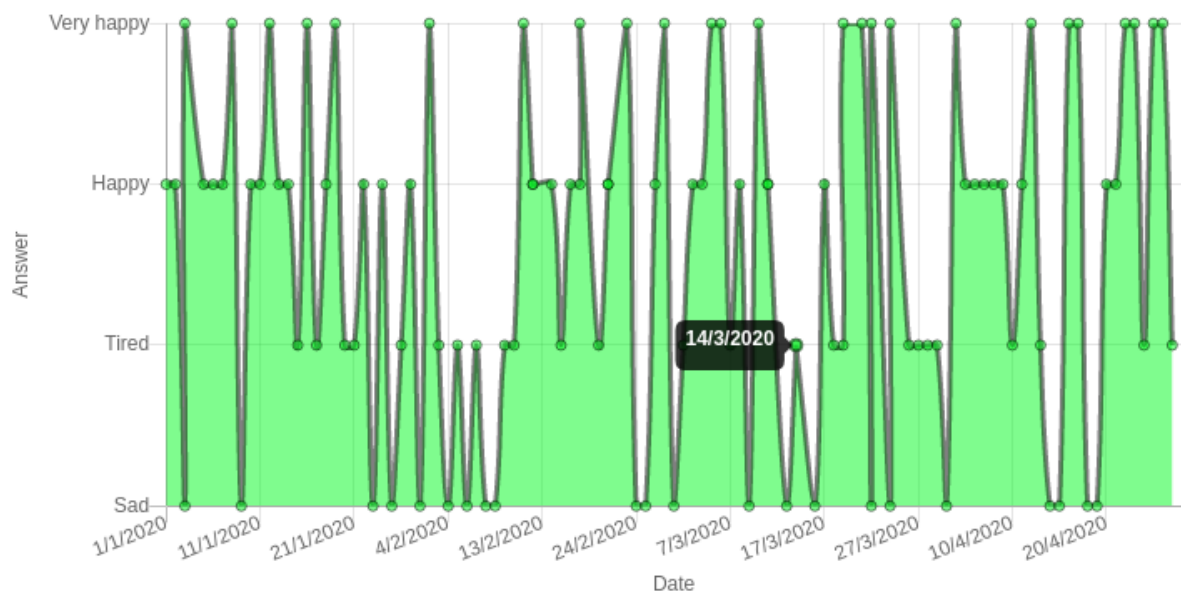


Figure 35: View data page (line chart).

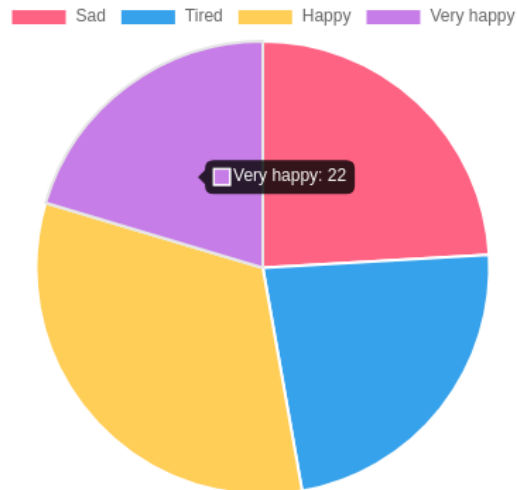


Figure 36: View data page (pie chart).

The last system view is the account manager, which consists of a dropdown that lets the user change the password, delete the account or log out.

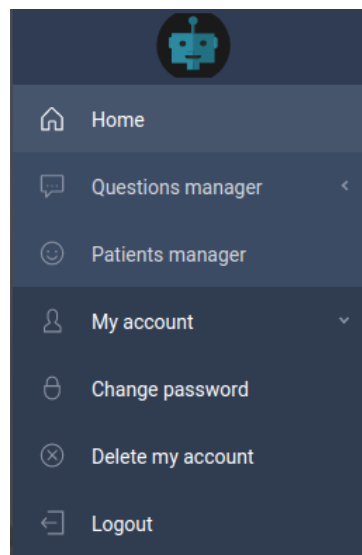


Figure 37: Account manager dropdown.

A screenshot of a 'Change your password' form. It has a title 'Change your password' at the top. Below the title are three input fields: 'Old password', 'New password', and 'New password (check)'. At the bottom of the form are two buttons: 'Submit' (blue) and 'Reset' (red).

Figure 38: Change password page.

```

@login_required(login_url="/login/")
def change_password(request):
    """
    Allows an user to change his/her password
    """
    success = False
    if request.method == 'POST':
        form = PasswordChangeForm(request.user, request.POST)
        if form.is_valid():
            user = form.save()
            update_session_auth_hash(request, user) # Important!
            success = True
    else:
        form = PasswordChangeForm(request.user)
    return render(request, 'change_password.html', {
        'form': form,
        'success': success
    })

```

Listing 15: Method to change user's password

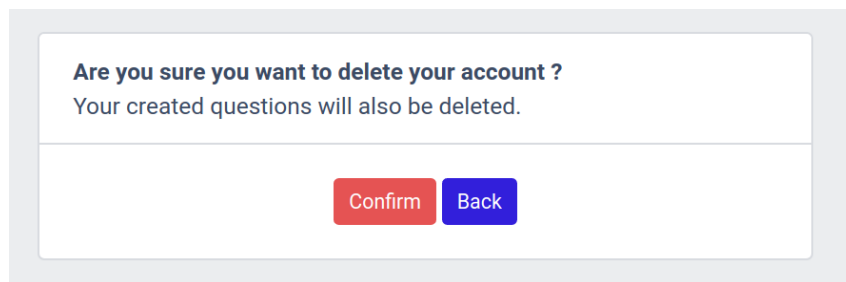


Figure 39: Delete account page.

```

@login_required(login_url="/login/")
def delete_account(request):
    """
    Deletes a doctor from the system
    """
    if request.method == "POST":
        request.user.doctor.delete()
        request.user.delete()
        return redirect('/')
    context = {}
    return render(request, 'delete_account.html', context)

```

Listing 16: Method to delete an account

HOW-R-U web interface icons have been gathered from (*Pixabay*, 2012)

3.6 Docker

The HOW-R-U development environment is composed of four main containers: conversational agent, web interface, database and reverse proxy. The containers use volumes, so that changes made to their files take effect immediately, avoiding an environment restart to test them. The *docker-compose* specification is the following one:

```
version: "3.1"
services:
  conversational-agent:
    image: conversational-agent
    links:
      - db:db
    volumes:
      - ~/Documents/TFG1920/howru/howru_chatbot:/opt/chatbot
    depends_on:
      - db
    restart: always
  db:
    image: postgres
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    restart: always
  web-interface:
    image: web_interface
    links:
      - db:db
    volumes:
      - ~/Documents/TFG1920/howru/howru_web_interface:/opt/web_interface
    ports:
      - "8080:8080"
    depends_on:
      - proxy
      - db
    restart: always
  proxy:
    image: proxy
    ports:
      - "85:85"
    volumes:
      - ./proxy:/etc/nginx/conf.d
    restart: always
```

Listing 17: HOW-R-U docker-compose file

The individual *Dockerfiles* of the previous containers are the following ones:

```
FROM python:3.6.10-stretch

# Install requirements

ADD config/requirements.txt /root/requirements.txt
ADD config/routes.json /etc/howru/cfg/routes.json
RUN pip install -r /root/requirements.txt

# iPython for debugging
RUN pip3.6 install ipython

# Set PYTHONPATH
ENV PYTHONPATH /opt/chatbot

# Run bot
WORKDIR /opt/chatbot
CMD python3 manage.py makemigrations && \
    python3 manage.py migrate && \
    python3 src/bot.py
#CMD tail -f /dev/null
```

Listing 18: HOW-R-U conversational agent Dockerfile

```
FROM python:3.6.10-stretch
ENV PYTHONUNBUFFERED 1

# Copy config files
ADD config/requirements.txt /root/requirements.txt
ADD config/routes.json /etc/howru/cfg/routes.json
ADD config/initial_data.json /initial_data.json

# Install dependencies
RUN pip install -r /root/requirements.txt

# iPython for debugging
RUN pip3.6 install ipython

# Set workdir
WORKDIR /opt/web_interface

# Run server
CMD python3 manage.py makemigrations && \
    python3 manage.py migrate && \
    python3 manage.py loaddata /initial_data.json && \
```



```
python3 manage.py runserver 0:8080
#CMD tail -f /dev/null
```

Listing 19: HOW-R-U web interface Dockerfile

```
FROM nginx:latest
COPY web_interface.conf /etc/nginx/conf.d/web_interface.conf
```

Listing 20: HOW-R-U web reverse proxy Dockerfile

The reverse proxy configuration file just indicates the port where the web interface will be hosted:

```
server {
    listen 85;
    location / {
        proxy_pass http://localhost:8080/;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

Listing 21: HOW-R-U web reverse proxy configuration file

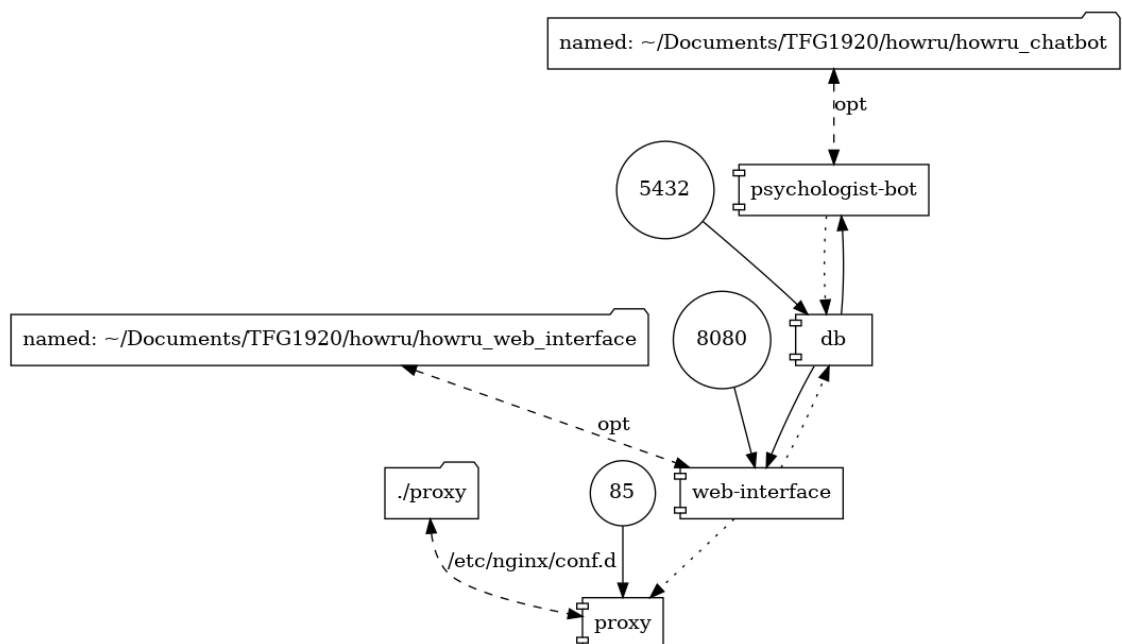


Figure 40: Docker-compose file schema.

4 Environment setup

4.1 Development environment

The development environment will be deployed using (*Docker*, 2020). To do that, the user just needs to install *Docker*, clone the code repository and run the attached script:

```
$> sudo apt-get update
$> sudo apt install apt-transport-https ca-certificates curl gnupg-agent
    ↪ software-properties-common
$> curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
    ↪ add -
$> sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/
    ↪ linux/ubuntu $(lsb_release -cs) stable"
$> sudo apt update
$> sudo apt install docker-ce docker-ce-cli containerd.io
```

Listing 22: Commands to install Docker

After that, the user needs to log out and back in so that permissions are re-evaluated. After that, the HOW-R-U development environment can be run by typing the following commands:

```
$> git clone https://github.com/csp98/TFG1920
$> cd howru/docker
$> ./start.sh
```

Listing 23: Commands to deploy development environment

The *start.sh* scripts build all the images and then launches them in *docker*:

```
#!/bin/zsh
# Chatbot
docker build -t psychologist-bot chatbot

# Web interface
docker build -t web_interface web_interface
docker build -t proxy proxy

docker-compose -p "howru" up -d --remove-orphans --force-recreate
```

Listing 24: start.sh script

4.2 Production environment

The production environment will be deployed using *Amazon Web Services*. There will be three types of instances: psychologist bot, web interface and databases.

4.2.1 Pre-deployment: Telegram bot creation

The first step is to create the production conversational agent in Telegram. To do so, the user must establish a conversation with *@BotFather* and send the bot creation commands:

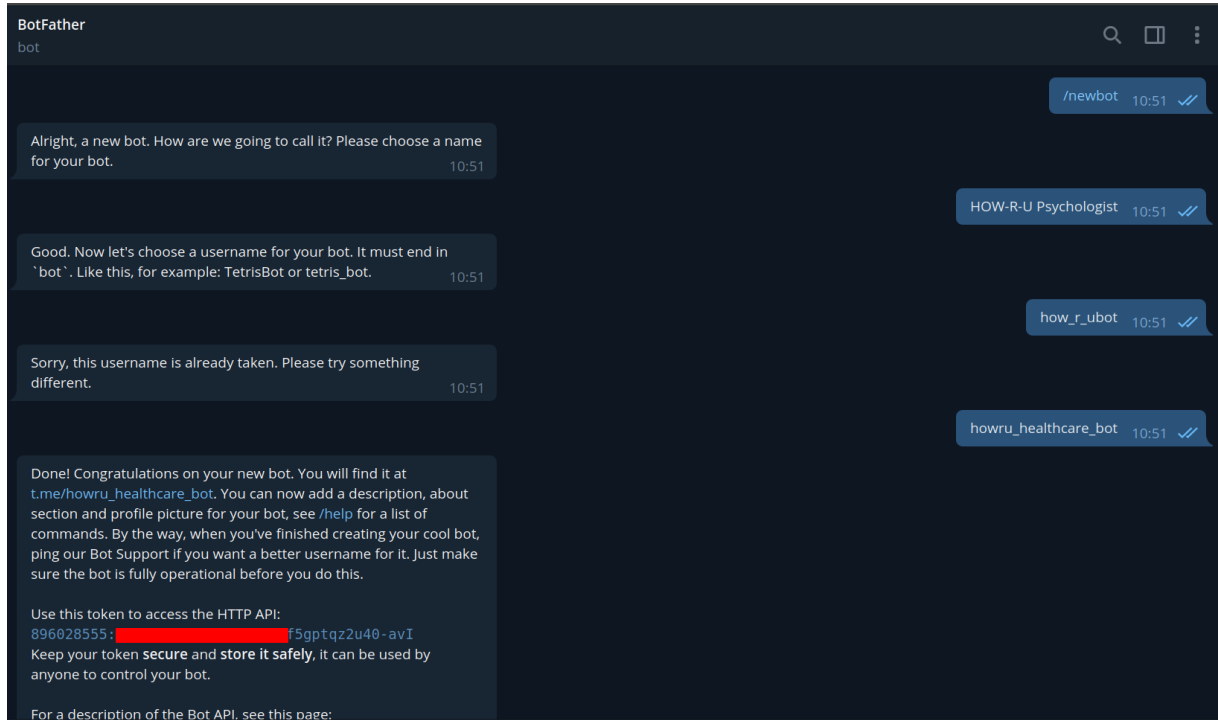


Figure 41: Telegram bot creation.

@BotFather will provide the user with a private authentication token that HOW-R-U will use to control the conversational agent. The user can customize the bot by adding a profile picture, changing description, etc., by using the related *@BotFather* commands.

4.2.2 System deployment

The next step is to deploy the system into AWS. The first component that is going to be deployed will be the database.

The user has to log in into *Amazon Relational Database System (RDS)* and create a PostgreSQL database, setting an username and password. The user can use this tutorial as reference: <https://aws.amazon.com/getting-started/tutorials/create-connect-postgresql-db/>.

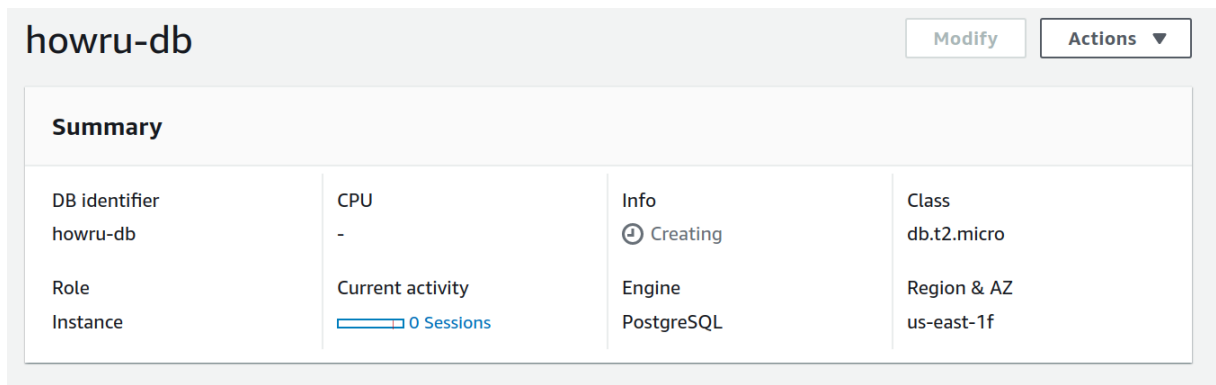


Figure 42: Database instance being created.

Then, the chatbot and web interface instances should be created using EC2. They should work under a Linux distribution. The web interface needs an specific rule in the security group so that it can be accessed from outside (port 80, HTTP):

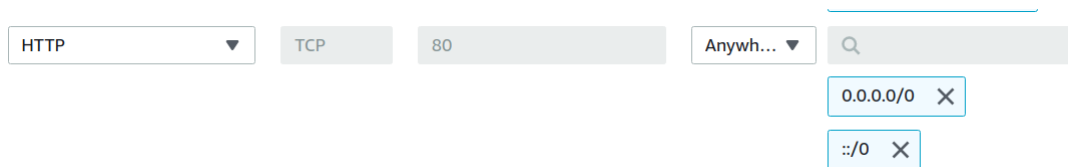


Figure 43: Security group rule for web interface instance.

The next step is to deploy the web interface. To do so, the user must clone the repository from the instance:

```
web_interface > sudo su
web_interface > yum install git
web_interface > mkdir /opt/web_interface
web_interface > cd /opt/web_interface
web_interface > git clone --recurse-submodules https://github.com/csp98
    ↪ /howru_web_interface.git .
```

Listing 25: Commands to download the web interface repository

Now we should configure four parameters: DB routes file, debug mode (should be disabled), allowed hosts and Django secret key. The first one consists of creating a file (*/etc/howru/cfg/routes.json*) with the following content:

```
{
  "host": "<DB instance DNS>",
  "port": 5432,
  "name": "postgres",
  "user": "<DB username>",
  "password": "<DB password>"
}
```

Listing 26: Routes file

The Django secret key can be generated with the following website: <https://miniwebtool.com/django-secret-key-generator/>. Once retrieved, the user should create the following file:

```
web_interface > nano /opt/web_interface/core/key.py
*****
KEY = '<KEY>'
*****
```

Listing 27: Command to create Django key file and key.py file contents

Finally, the user must edit `/opt/web_interface/core/settings.py` setting `DEBUG=False` and `ALLOWED_HOSTS = ['*']`

The environment files are now ready. The next steps consist on installing *Python*, the requirements and running the server.

```
web_interface > yum install python3 python3-pip python-psycopg2
    ↪ postgresql gcc python3-devel
web_interface > pip3 install -r requirements.txt
web_interface > python3 manage.py makemigrations
web_interface > python3 manage.py migrate
```

Listing 28: Commands to install Python and its requirements

Finally, we will configure *screen* to detach the session where the interface will run:

```
web_interface > yum install screen
web_interface > screen
web_interface > python3 manage.py runserver 0:80
```

Listing 29: Commands to detach the session and run the web interface server

And finally press *Ctrl+A+D* and close the SSH session. The web interface is now successfully configured and deployed and doctors can now use it by accessing through the public DNS.

Finally, let's deploy the conversational agent. Log in into the instance via SSH and do the following:

```
conv_agent > sudo su
conv_agent > yum install python3 python3-pip python3-devel git screen
conv_agent > mkdir /opt/chatbot
conv_agent > cd /opt/chatbot
conv_agent > git clone --recurse-submodules https://github.com/csp98/
    ↪ howru_chatbot.git .
```

Listing 30: Commands to download HOW-R-U chatbot module and install its dependencies

Create the routes file (same as the web interface one) in the same directory. After that, the file with the bot secret token, `/opt/chatbot/config/bot_config.py`, and add the following:

```
TOKEN = "<SECRET TOKEN>"
```

Listing 31: bot_config.py file contents

Now install the requirements, detach the screen and launch the conversational agent:

```
conv_agent > python3 -m pip install -r requirements.txt
conv_agent > screen
conv_agent > export PYTHONPATH='pwd'
conv_agent > export DJANGO_SETTINGS_MODULE="manage.settings"
conv_agent > python3 manage.py makemigrations
conv_agent > python3 manage.py migrate
conv_agent > python3 src/bot.py
```

Listing 32: Commands to detach the screen, install the chatbot requirements and run it

After that, HOW-R-U system is ready to be used.

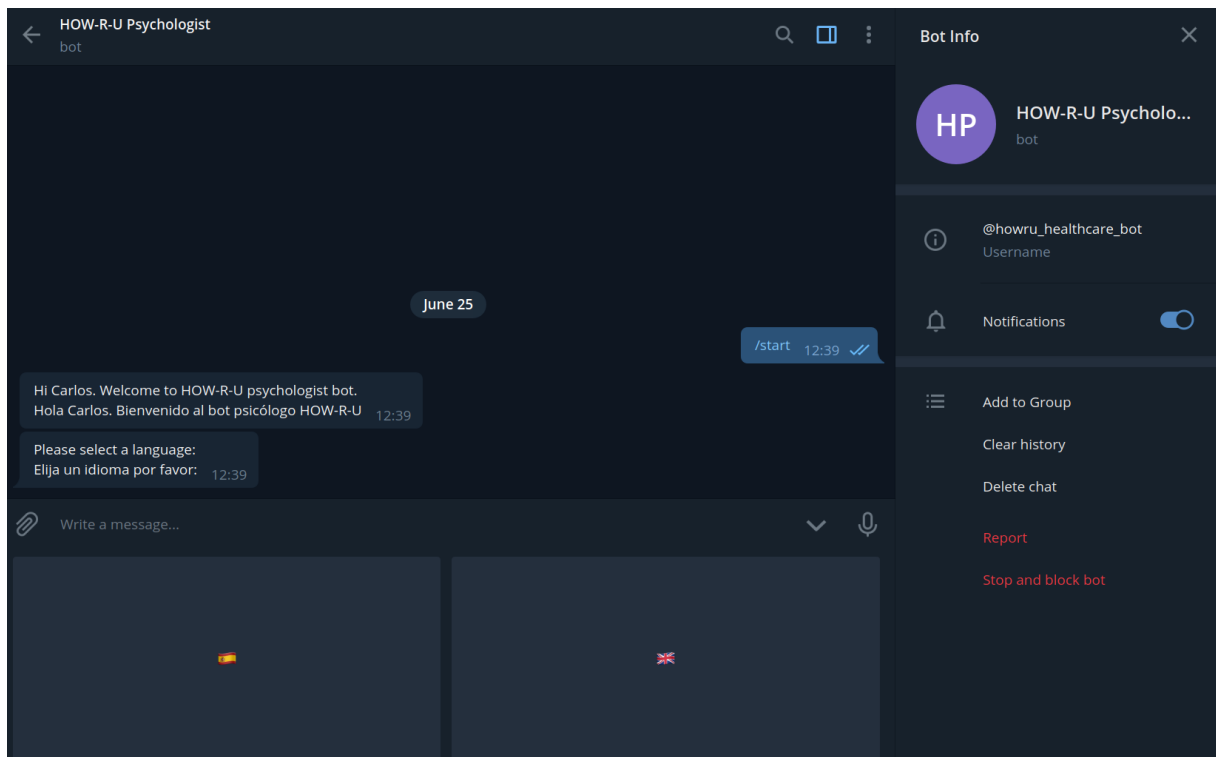


Figure 44: HOW-R-U production conversational agent.

5 Discussion

The discussion of the proposed system will be based on a SWOT analysis (**S**trengths, **W**eaknesses, **O**pportunities and **T**hreats). The main strength of HOW-R-U is its versatility: it can be used in many different scenarios, from health domains to analysis of data based on surveys. For example, a health clinic could offer a suite of e-coaches based on HOW-R-U: a nutritionist, a psychologist, a dermatologist, etc., that would make the patients regular questions so that their daily status is tracked by doctors, obtaining a big quantity of useful data. Moreover, the interface is intended to be easy to use and learn so that users do not struggle when manipulating it. Another strength is that the patient does not need to install a specific application to use it because HOW-R-U conversational agent can be used both through Telegram Web or Desktop. Regarding the opportunities of the system, a strong case of use could be a wide data analysis about COVID-19 symptoms to identify patterns over the population, build a map based on infected patients, etc.,. Furthermore, as Telegram is one of the chat applications that offers more privacy for its users, the system can be more attractive to them than other systems that use other apps ensuring less privacy.

The main weakness of the system is that its deployment requires specific knowledge about architectures, system shells, etc., so it is not suitable for everyone. In addition, Telegram is not used by the majority of people, so it is likely that most patients will have to install it. This weakness leads to the threat of another system that uses its own specific app. As patients would have to install an application in both cases, maybe they could opt for the particular one, that could offer another user interface, animations, etc.,. Another threat is that elder people may not know how to use Telegram. Lastly, not every patient has a smartphone, so a minor percentage of the couldn't use the system.

STRENGTHS	WEAKNESSES
<ol style="list-style-type: none"> 1. Different scenarios. 2. Multiple e-coaches. 3. Intuitive interface. 4. No extra software for patients. 	<ol style="list-style-type: none"> 1. Telegram is not used by the majority of people. 2. System deployment requires knowledge on computer science.
OPPORTUNITIES	THREATS
<ol style="list-style-type: none"> 1. COVID-19 data analysis. 2. Telegram ensures privacy. 	<ol style="list-style-type: none"> 1. Elder people may not know how to use Telegram. 2. If patients do not have Telegram installed, they could install a specific app instead of Telegram. 3. Not all patients have smartphones.

Figure 45: HOW-R-U SWOT analysis. Based on <http://www.mostlycolor.ch/2015/07/swot-matrices-in-latex.html>

6 Planification and budget

6.1 Planification

The planification of this project has been divided into two main groups: documentation and system implementation. The first part refers to the build process of this memory, along with learning to code it in L^AT_EX(Michel Goossens & Samarin, 1993), whereas the second part consists on the process of learning to use the different frameworks that compose HOW-R-U to create it.

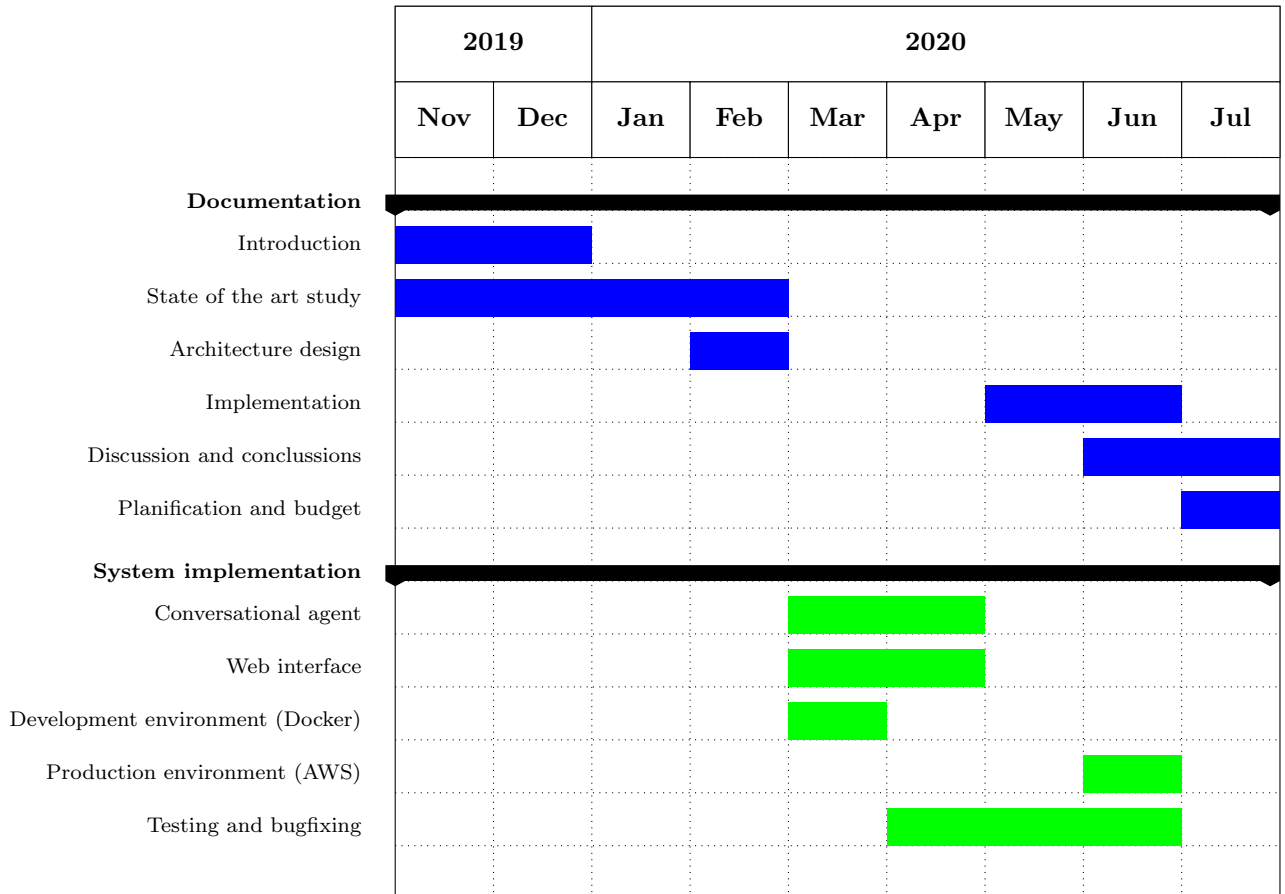


Figure 46: Gantt Diagram showing the project's planification time.

6.1.1 Budget

HOW-R-U is distributed under a Creative Commons “Attribution-NonCommercial-ShareAlike 4.0 International” license, which means that the project can be freely used **as long as** it is not generate profit, so the main budget element will be the hosting. HOW-R-U can be hosted in a proprietary infrastructure (in which the system administrator would have to buy and maintain all the equipment) or in the cloud (where the system administrator would pay per use). Regarding the cloud model, there could be three approaches depending on the system usage: a high and scalable one, a low and non scalable one and a single instance.

6.1.2 Scalable environment

Scalable configuration is suitable if the system is widely used. It is formed by three types of instances, one per module: database, web interface and conversational agent. All these machines must be able to do autoscaling, (the number of machines increases and decreases according to the workload).

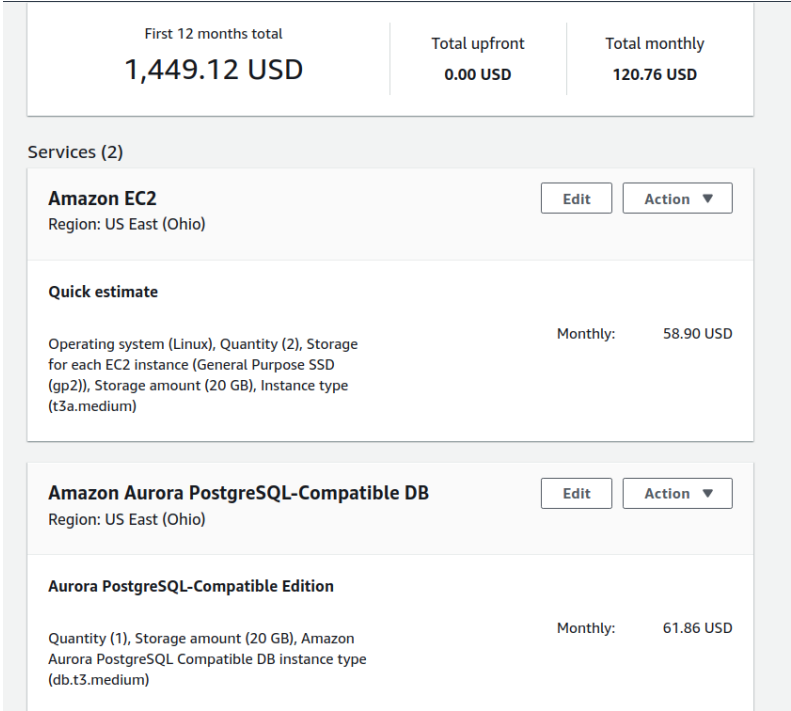


Figure 47: AWS cost for a HOW-R-U scalable production environment (Amazon Web Services Calculator (<https://calculator.aws/#/estimate>))

6.1.3 Non scalable environment

Non-scalable configuration is oriented for environments where the workload is not high. It consists on two machines: a database server and an instance that will host both the conversational agent and the web interface. If the number of accesses to the system increases, it would be easy to create another instance and transition to the scalable environment. Moreover, no database migrations would be needed, as this architecture provides an independent database instance.

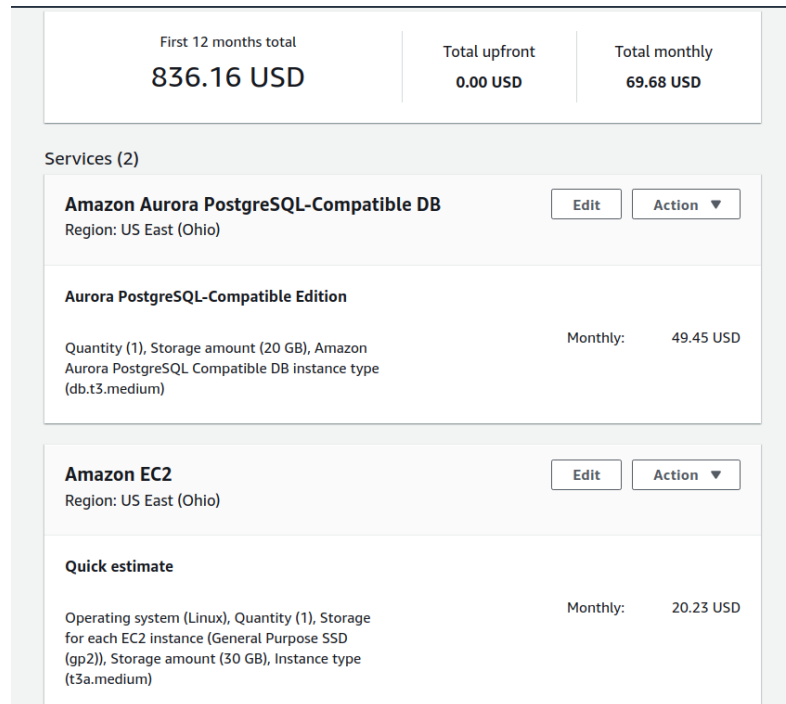


Figure 48: AWS cost for a HOW-R-U non scalable production environment (Amazon Web Services Calculator (<https://calculator.aws/#/estimate>))

6.1.4 One single instance

This is the cheapest alternative, but the worst one if the system expects to get a high workload. It consists on a single instance where all modules run: database, web interface and conversational agent. Apart from the performance, the main disadvantage of this setup is that if the administrator wanted to switch to a scalable one it would be necessary to perform a database migration.

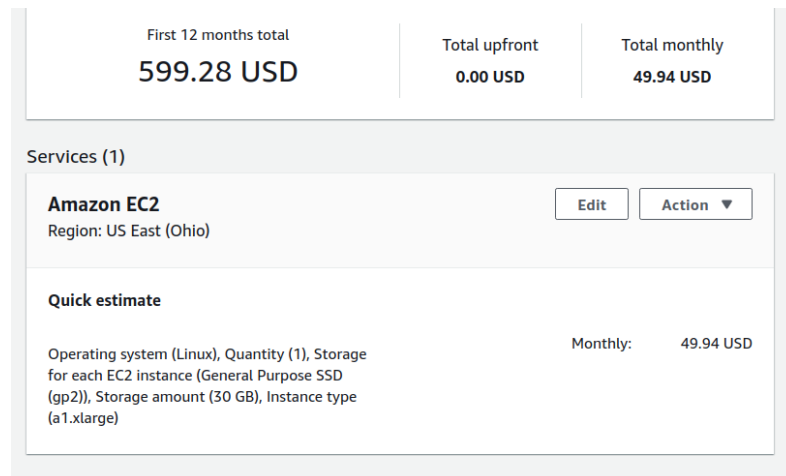


Figure 49: AWS cost for a HOW-R-U one-instance production environment (Amazon Web Services Calculator (<https://calculator.aws/#/estimate>))

7 Conclusions

- **Main goal: to develop a conversational-agent-as-a-sensor which will be able to interact with a person with a disorder and ask questions defined by specialists.** This objective is achieved, as the conversational agent module manages patient-system interactions. Moreover, the system allows doctors to create custom questions through the web interface. In addition, they can use questions created by other specialists (if they set them as public).
- **Secondary goals**
 - **To design a graphical web interface where doctors can consult their patient’s responses.** This goal is also reached, as shown in the web interface implementation section. Besides, doctors can either analyse the retrieved data through the integrated charts or download a CSV file to elaborate their own metrics.
 - **To design a flexible and scalable architecture to add functionality to the system.** As doctors can add custom questions with the system, HOW-R-U can be used in a lot of scenarios. The *data analyst* feature also allows the system to be used not only in medical situations, but also in survey ones.
 - **To design an architecture based on containers to host the different system modules.** As shown in the Docker implementation section, the system is structured in containers so that a development environment can be easily set up.
 - **To implement a system that covers the previous goals.** As the implementation section illustrates, the proposed system has been successfully developed.
 - **To test a beta version of the assistant in real people and analyse the retrieved data as well as target audience’s feelings about it.** This is the only objective that has not been accomplished. HOW-R-U started being a much more elementary system: it did not have a web interface (doctors would have had to control everything from another Telegram bot), questions were not customizable (the developer would have had to define and hardcode them), there were no concepts such as frequency or priority (questions were asked daily in a random order), etc.,. We decided to leave this goal behind in order to give more functionality and customization to HOW-R-U so that it could be used in a wider variety of scenarios, not only the health care ones.

The whole source code of HOW-R-U is available at <https://github.com/csp98/TFG1920>

References

- Alesanco, Á., Sancho, J., Gilaberte, Y., Abarca, E., & García, J. (2017). Bots in messaging platforms, a new paradigm in healthcare delivery: Application to custom prescription in dermatology. In (Vol. 65, p. 185-188).
- Appseed coreui django dashboard*. (2020). <https://github.com/app-generator/django-dashboard-coreui>.
- Bandelow, B., & Michaelis, S. (2015). Epidemiology of anxiety disorders in the 21st century. *Dialogues in clinical neuroscience*, 17(3), 327-335.
- Bennet Praba, M., Sen, S., Chauhan, C., & Singh, D. (2019). Ai healthcare interactive talking agent using nlp. *International Journal of Innovative Technology and Exploring Engineering*, 9(1), 3470-3473.
- Bresó, A., Martínez-Miranda, J., Botella, C., Baños, R., & García-Gómez, J. (2016). Usability and acceptability assessment of an empathic virtual agent to prevent major depression. *Expert Systems*, 33(4), 297-312.
- Clegg, D., & Barker, R. (1994). *Case method fast-track: A rad approach*. Addison-Wesley Publishing Company.
- Creative commons*. (2001). <https://creativecommons.org/>.
- D'Alfonso, S., Santesteban-Echarri, O., Rice, S., Wadley, G., Lederman, R., Miles, C., ... Alvarez-Jimenez, M. (2017). Artificial intelligence-assisted online social therapy for youth mental health. *Frontiers in Psychology*, 8(JUN).
- Davies, M. (2000). The stigma of anxiety disorders. *International journal of clinical practice*, 54, 44-7.
- diagrams.net*. (2020). Retrieved from <https://www.diagrams.net/>
- Discord*. (2015). <https://discordapp.com/>.
- Django*. (2020). <https://djangoproject.com>.
- Docker*. (2020). Retrieved from <https://www.docker.com/>
- Drislane, L., Waller, R., Martz, M., Bonar, E., Walton, M., Chermack, S., & Blow, F. (2020). Therapist and computer-based brief interventions for drug use within a randomized controlled trial: effects on parallel trajectories of alcohol use, cannabis use and anxiety symptoms. *Addiction*, 115(1), 158-169.
- Facebook messenger*. (2008). <https://www.facebook.com/messenger/>.
- Falala-Sechet, C., Antoine, L., Thiriez, I., & Bungener, C. (2019). Owlle: A chatbot that provides emotional support for coping with psychological difficulties. In (p. 236-237).
- Guo, X., Liu, J., & Chen, Y. (2020). When your wearables become your fitness mate. *Smart Health*, 16.

- Harilal, N., Shah, R., Sharma, S., & Bhutani, V. (2020). Caro: An empathetic health conversational chatbot for people with major depression. In (p. 349-350).
- Hirano, M., Ogura, K., Kitahara, M., Sakamoto, D., & Shimoyama, H. (2017). Designing behavioral self-regulation application for preventive personal mental healthcare. *Health Psychology Open*, 4(1).
- Hudlicka, E. (2013). Virtual training and coaching of health behavior: Example from mindfulness meditation training. *Patient Education and Counseling*, 92(2), 160 - 166.
- Kik. (2010). <https://www.kik.com/>.
- Kosinski, M., Stillwell, D., & Graepel, T. (2013). Private traits and attributes are predictable from digital records of human behavior. *Proceedings of the National Academy of Sciences*, 110(15), 5802–5805.
- Line. (2012). <https://line.me/>.
- López, V., Eisman, E., & Castro, J. (2008). A tool for training primary health care medical students: The virtual simulated patient. In (Vol. 2, p. 194-201).
- Maharjan, R., Bækgaard, P., & Bardram, J. (2019). “hear me out”: Smart speaker based conversational agent to monitor symptoms in mental health. In (p. 929-933).
- Michel Goossens, F. M., & Samarin, A. (1993). *The latex companion*.
- Montenegro, J., da Costa, C., & da Rosa Righi, R. (2019). Survey of conversational agents in health. *Expert Systems with Applications*, 129, 56-67.
- Nginx. (2019). Retrieved from <https://www.nginx.com/>
- Ni, L., Lu, C., Liu, N., & Liu, J. (2017). Mandy: Towards a smart primary care chatbot application. *Communications in Computer and Information Science*, 780.
- of Health Authorities, N. A., & Confederation, T. (2014). The future of mental health.
- Pixabay. (2012). <https://pixabay.com/>.
- Postgresql. (2020). Retrieved from <https://www.postgresql.org/>
- Rastogi, N., & Hendler, J. (2017, 01). Whatsapp security and role of metadata in preserving privacy.
- Ring, L., Bickmore, T., & Pedrelli, P. (2016). Real-time tailoring of depression counseling by conversational agent. *iproc*, 2(1), e27.
- Ritchie, H., & Roser, M. (2018). Mental health. *Our World in Data*.
- Roca, S., Hernández, M., Sancho, J., García, J., & Alesanco, á. (2020). Virtual assistant prototype for managing medication using messaging platforms. In (Vol. 76, p. 954-961).
- Shorey, S., Ang, E., Yap, J., Ng, E., Lau, S., & Chui, C. (2019). A virtual counseling application using artificial intelligence for communication skills training in nursing education: Development study. *Journal of medical Internet research*, 21(10), e14658.

Slack. (2013). <https://slack.com/>.

Sutikno, T., Handayani, L., Stiawan, D., Riyadi, M., & Subroto, I. (2016, 06). What-sapp, viber and telegram which is best for instant messaging? *International Journal of Electrical and Computer Engineering (IJECE)*, 6, 909.

Telegram. (2013). <https://telegram.org/>.

Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual*. Scotts Valley, CA: CreateSpace.

Viber. (2010). <https://www.viber.com/>.

Vicente, D. (2019). La ratio de psicólogos, a 16 puntos de europa. *El Mundo*.

Wechat. (2011). <https://www.wechat.com/>.

Whatsapp. (2009). <https://www.whatsapp.com/>.

Yasavur, U., Lisetti, C., & Rishe, N. (2014). Let's talk! speaking virtual counselor offers you a brief intervention. *Journal on Multimodal User Interfaces*, 8(4), 381-398.

This work is licensed under a Creative Commons “Attribution-NonCommercial-ShareAlike 4.0 International” license.

