UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de un chatbot para la evaluación de la usabilidad y la experiencia de usuario de una plataforma web de ayuda a la decisión médica

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Daniel Mora Blasco

Tutor: Ángeles Calduch Losa
Sabina Asensio Cuesta
Vicent Blanes Selva

Curso 2020-2021

Resum

Aleph és un sistema d'ajuda a la decisió mèdica web desenvolupat en *Django*, el qual ofereix eines de maneig de malalties, suport a la presa de decisions clíniques i suport al tractament. Per a millorar aquest sistema volem avaluar la seua usabilitat i l'experiència d'usuari.

Per dur a terme aquest treball hem dissenyat i desenvolupat Alf, un xatbot que realitza un test usabilitat i l'experiència d'usuari d'*Aleph* de manera interactiva al que s'adapten dos dels qüestionaris més usats per avaluar aquestes qualitats juntament amb tres tasques senzilles per familiaritzar l'usuari amb *Aleph*.

Els qüestionaris que hem adaptat són el *System Usabity Scale (SUS)* i el *User Experience Questionnaire (UEQ-S)*. A més en les tasques avaluem la seua dificultat, el temps emprat, la seua eficàcia i la satisfacció de l'usuari.

Aquest xatbot està compost per dos mòduls, un *Frontend* desenvolupat en *Django*, un *Backend* compost per una API de REST, un *Object-Relational Mapping (ORM)*, una base de dades i una màquina d'estats creada en *Python 3*.

Paraules clau: Chatbot, Django, Bootstrap, API REST, FastAPI, ORM, Bases de dades, Python, Màquina d'estats

Resumen

Aleph es un sistema de ayuda a la decisión médica web desarrollado en *Django*, el cual ofrece herramientas de manejo de enfermedades, apoyo a la toma de decisiones clínicas y apoyo al tratamiento. Para mejorar dicho sistema queremos evaluar su usabilidad y la experiencia del usuario.

Para llevar a cabo este trabajo hemos diseñado y desarrollado Alf, un chatbot que realiza un test usabilidad y la experiencia de usuario de *Aleph* de manera interactiva, en el que se adaptan dos de los cuestionarios más usados para evaluar dichas cualidades junto con tres tareas sencillas para familiarizar al usuario con *Aleph*.

Los cuestionarios que hemos adaptado son el *System Usabity Scale (SUS)* y el *User Experience Questionnaire (UEQ-S)*. Adicionalmente en las tareas evaluamos su dificultad, el tiempo empleado, su eficacia y la satisfacción del usuario.

Este chatbot está compuesto de dos módulos, un *Frontend*, desarrollado en *Django*, y un *Backend* compuesto por una *API de REST*, una *Object-Relational Mapping (ORM)*, una base de datos y una máquina de estados creada en *Python 3*.

Palabras clave: Chatbot, Django, Bootstrap, API REST, FastAPI, ORM, Bases de datos, Python, Máquina de estados

Abstract

Aleph is a web-based medical decision support system developed in *Django*, which offers disease management tools, clinical decision support, and treatment support. To improve this system we want to evaluate its usability and user experience.

To carry out this work we have designed and developed Alf, a chatbot that performs a usability test and the *Aleph* user experience in an interactive way, in which two of the most used questionnaires are adapted to evaluate these qualities together with three simple tasks to familiarize the user with *Aleph*.

The questionnaires that we have adapted are the *System Usabity Scale (SUS)* and the *User Experience Questionnaire (UEQ-S)*. Additionally, in the tasks we evaluate their difficulty, the time spent, their effectiveness and user satisfaction.

This chatbot is made up of two modules, a *Frontend*, developed in *Django*, and a *Backend* made up of a REST API, an *Object-Relational Mapping (ORM)*, a database, and a state machine created in *Python 3*.

Key words: Chatbot, Django, Bootstrap, API REST, FastAPI, ORM, Database, Python, State machine

Índice general

Índice de figuras

Agradecimientos

A mis tutoras Ángeles y Sabina por ayudarme y corregirme.

A Vicent por motivarme y ayudarme con este proyecto. Me siento muy afortunado de haberte tenido como tutor, sin ti este proyecto no sería lo que es.

A Isabel por soportarme y estar conmigo en lo más duro.

A mis abuelos Vicente, Daniel, Carmen y Fina por criarme y haberme hecho llegar hasta aquí, espero que estéis orgullosos.

CAPÍTULO 1
Introducción

1.1 Motivación

Aleph[1] es el proyecto *software* desarrollado por el Biomedical Data Science Lab (BDS-Lab) del Instituto ITACA durante el proyecto europeo InAdvance y que forma parte de la tesis doctoral de Vicent Blanes Selva y en el que participan los investigadores Juan Migel García-Gómez y Sabina Asensio Cuesta. Se trata de una aplicación web que ofrece diferentes servicios de ayuda a la decisión médica o *CDSS* (de sus siglas en inglés *Clinical Decision Support System*). La idea central de *Aleph* es poder ofrecer una plataforma común para diferentes servicios médicos de predicciones y análisis, utilizando tecnologías de microservicios para agregar sistemas de apoyo a la decisión clínica sin esfuerzo y siempre compartiendo una misma estética, diseño y usabilidad.

Este proyecto aún se encuentra en desarrollo, por lo que todavía se trata de una demostración, es decir, un prototipo funcional que no incluye toda la lista final de funcionalidades. No obstante, su interfaz actual (ver Figura 1.1) sí está validada y se ha evolucionado mediante rondas de entrevistas con los potenciales usuarios de esta web.

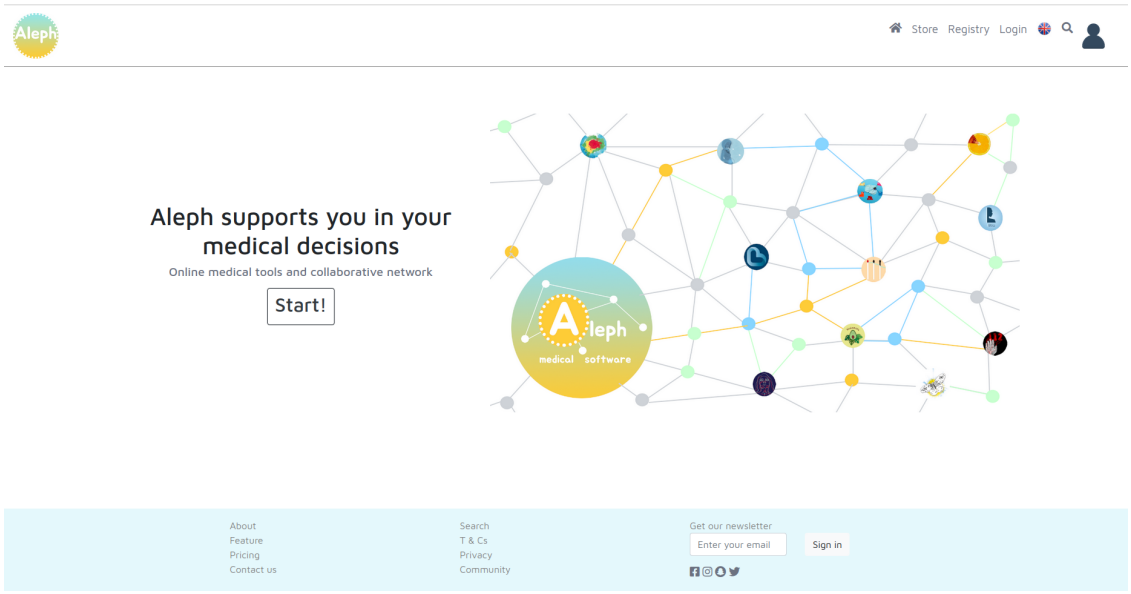

Figura 1.1: Página inicial de Aleph

Actualmente, en la demo de *Aleph* (ver Figura 1.2) se encuentra disponible solamente la gestión de servicios (*Service management*), las herramientas de manejo de enfermedades (*Disease management tools*), en las cuales se hallan los servicios:

- Cuidados Paliativos (*Palliative Care*)
- Calculadora de mortalidad (*Mortality Calculator*)

Las aplicaciones de apoyo a la decisión clínica y los tratamientos se encuentran inhabilitadas actualmente.

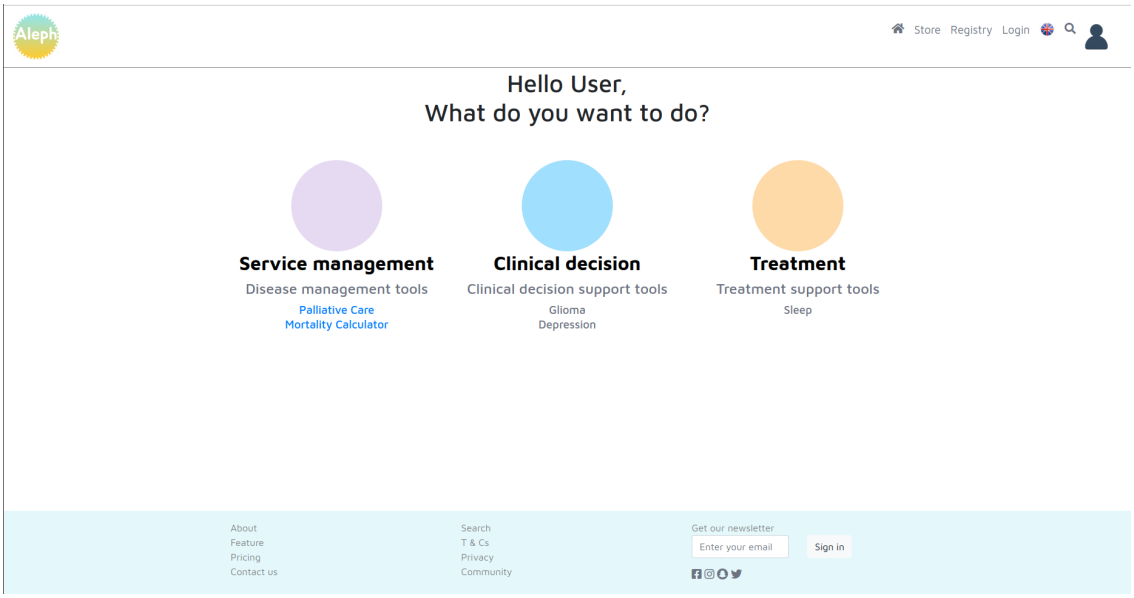

Figura 1.2: Herramientas de ayuda a la decisión médica de Aleph

Además, en este tipo de aplicaciones es primordial un diseño basado en la **usabilidad** y en la **experiencia de usuario**. Ambos atributos están muy presentes cuando se habla de aplicaciones, páginas web, *software* u otras herramientas del ámbito de la informática y de la tecnología. La relevancia de estas cualidades tiene un impacto significativo a la hora que elegir y usar un producto de dichos ámbitos, a corto, medio y largo plazo. La experiencia de usuario los primeros días no será la misma que cuando esta experiencia se haya prolongado a lo largo de varios meses o años.

En primer lugar, la **usabilidad** hace referencia a la facilidad o eficacia con la que los usuarios pueden hacer uso de una herramienta y con la que logran un fin concreto. Si se habla del ámbito de la informática, la herramienta sería todo aquel dispositivo *hardware* o aplicación *software* que le permite a las personas lograr un objetivo.

La usabilidad es una medida empírica y relativa. Empírica ya que se basa en pruebas de usabilidad llevadas a cabo por analistas, diseñadores y expertos en la materia y realizadas en laboratorios o mediante el propio uso de la herramienta con grupos de usuarios a los que está dedicada. Y relativa ya que sus resultados ni son positivos ni negativos, dependen del propósito último de esta herramienta. Lo que se obtiene después de evaluar estas pruebas se llama grado de usabilidad.

En segundo lugar, se halla la **experiencia de usuario**, esta, en contraposición con la primera se trata de una característica subjetiva, es decir, cada usuario tiene una percepción propia de la misma. Esta experiencia surge del conjunto de factores y elementos referentes al uso de la propia herramienta. Esta percepción depende tanto del diseño (dentro del diseño encontraríamos el grado de usabilidad) como de causas particulares de los propios usuarios como son: gustos, sentimientos, emociones, etc. La experiencia de usuario busca como fin la satisfacción del consumidor y esta no es relativa, sino que es positiva o negativa.

Hoy en día casi cualquier producto *software* se diseña atendiendo a estos dos atributos, ya que si se obtiene un buen grado de usabilidad y una buena experiencia de usuario se suele obtener una ventaja competitiva frente a otros productos de características similares.

Por último, en un artículo publicado en *Computers in Human Behavior* se recoge un estudio que evalúa el impacto de usar cuestionarios basados en conversaciones integradas en la aplicación [3], es decir, sobre un *chatbot* embebido en aplicaciones de salud móvil o *mHealth* (del inglés *mobile health*). Este proyecto, además, se fundamenta en la experiencia de los investigadores de *Aleph* (Vicent, Juan Miguel y Sabina) en el desarrollo del *chatbot Wakamola* [10] utilizado como base para la creación de nuestro *chatbot*.

Estos son los pilares centrales sobre los que se fundamenta este trabajo, añadiremos a *Aleph* un módulo para determinar el grado de los atributos, anteriormente mencionados, de la propia web.

1.2 Objetivos

El objetivo de este trabajo es desarrollar un módulo *software* adicional, en forma de *chatbot* embebido dentro del sistema *Aleph*, que ayude a los desarrolladores del proyecto a recabar datos sobre el grado usabilidad y la experiencia de usuario implementando los cuestionarios *System Usabity Scale (SUS)* [6] y *User Experience Questionnaire (UEQ-S)* [5]. Así mismo, se deberá crear una estructura que permita realizar *Task Test* con facilidad. En concreto en este trabajo se evaluará la realización de tres tareas sencillas con la web:

1. Registrarse en Aleph.
2. Crear un caso aleatorio de cuidados paliativos y ver los resultados.
3. Consultar la información referente al proyecto de Aleph.

Implementar todo esto no se tratará de una tarea sencilla, para evaluar los cuestionarios, anteriormente citados, el usuario tiene que tener experiencia usando la página web. Para ello será necesario implementar estas tres tareas sencillas que permitan al usuario adquirir dicha experiencia, en caso de no tenerla.

La idea principal de este proyecto reside en que el usuario, ya sea nuevo en la página web o ya haya participado en otros ensayos (que evalúan la usabilidad y la experiencia

de *Aleph*), pueda realizar un test de manera interactiva y con unas formas de expresarse más cercanas a las propias personas, evitando así la realización de cuestionarios más usuales como los de papel, entrevistas o cuestionarios *on-line* como *Google Forms*. Cabe mencionar la preferencia de los usuarios al decantarse por usar chatbots en vez de por cuestionarios más tradicionales. Así pues, según los desarrolladores de *Wakamola*:

Estudios anteriores encontraron que los chatbots pueden ser más atractivos para que los usuarios completen cuestionarios. Por lo tanto, es más probable que respondan preguntas a través de un chatbot que en un cuestionario o entrevista porque los asocian con el entretenimiento y los factores sociales y relacionales, además de ser curiosos ya que los ven como un fenómeno novedoso. Algunos estudios recientes analizaron las ventajas de los chatbots sobre las encuestas web [4].

El módulo debe almacenar la información proporcionada por los usuarios para su posterior análisis y mejora de la aplicación *Aleph*. También debe ser capaz de generar una buena impresión al usuario, tener personalidad y coherencia y hacer uso de un lenguaje natural humano, aunque sin necesidad de tratarse del procesamiento de lenguaje natural (NLP). Adicionalmente tiene que tener la capacidad de ser multilingüe, ofrecer la alternativa de realizar los cuestionarios en castellano, inglés y valenciano.

Asimismo, estas tareas evaluarán su dificultad, el tiempo empleado en su realización, su eficacia y la satisfacción del usuario tras realizarlas. Por tanto, el *software* que queremos crear tendrá que tener un diseño agradable al usuario y almacenar un conjunto de respuestas generadas por el usuario, realizando un procesamiento de los mensajes mediante una máquina de estados.

Para la implementación hemos usado *Django* y *Python 3*. *Django* es el *framework* en el que está implementado *Aleph*, en consecuencia, ha sido necesario usar *JavaScript*, *HTML* y *CSS*, ya que el *framework* hace uso de estos lenguajes para la programación de la interfaz web. *Python* debido a las cualidades del lenguaje que nos va a permitir tanto realizar el procesamiento de los mensajes de forma sencilla como la interacción con la base de datos.

1.3 Estructura de la memoria

En los siguientes capítulos se verá la memoria de desarrollo y sus conclusiones. Esta memoria está dividida en cinco partes, en las dos primeras veremos en detalle la explicación de las tareas propuestas y los cuestionarios, el porqué de las preguntas que deberá contestar el usuario y lo que se busca evaluar. En las dos siguientes se buscará explicar todo el desarrollo y la implementación del chatbot, más concretamente, en la tercera parte se presentará la interfaz del chatbot, el *Frontend* y en la cuarta parte se desarrollará el *Backend* y **Alf**, es decir, la máquina de estados que nos permitirá realizar las tareas, los cuestionarios, reconocer y almacenar las respuestas del usuario. Por último se explicará toda la orquestación y despliegue de **Alf**, dando una visión global del funcionamiento conjunto del *Frontend* y el *Backend*, además de todos los requerimientos necesarios para su despliegue.

En la Figura 1.3 se ve el diagrama de flujo de la aplicación para hacernos una primera imagen de cómo van a encajar las diferentes piezas de la aplicación.

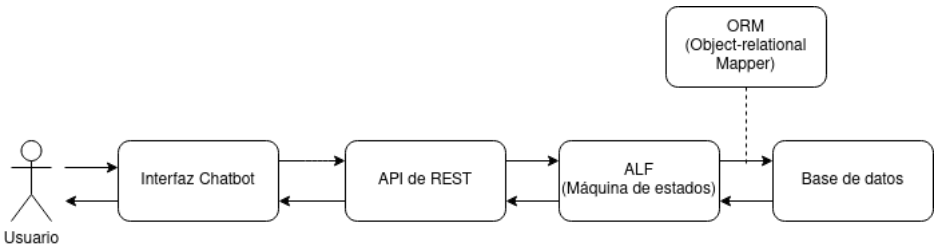

Figura 1.3: Diagrama de flujo de Alf

En las conclusiones analizaremos los resultados obtenidos tras el desarrollo y la implementación del módulo. Por último, veremos mejoras, ampliaciones y pasos siguientes que, por motivos de tiempo o ámbito, han quedado fuera del proyecto.

CAPÍTULO 2
Estado del arte

En este capítulo analizaremos los tres artículos científicos que conforman la columna vertebral de este proyecto. Daremos una explicación de sus características más importantes, metodologías y resultados obtenidos, ya que de no explicar este marco teórico no se entendería el objetivo de Alf. Finalmente, el capítulo concluirá con nuestra propuesta, en la que se ve el punto en el que confluyen estos artículos.

2.1 Cuestionarios basados en conversaciones integradas en la aplicación

En esta sección se desarrolla los puntos clave, la metodología y los resultados de **Evaluación de la experiencia del usuario a largo plazo en una aplicación de salud móvil a través de un cuestionario basado en conversación integrado en la aplicación** (*Assessing long-term user experience on a mobile health application through an in-app embedded conversation-based questionnaire* en inglés) [3]. En este artículo, sus varios autores estudiaron el uso de *chatbots* embebidos en aplicaciones de *mHealth* para evaluar la experiencia de usuario.

Según D. Biduski y el resto de autores, una experiencia satisfactoria es crítica para el usuario con el fin de mantenerse motivado a lo largo del tiempo, especialmente en aplicaciones de salud móvil [3]. El estudio trata de dar respuesta al efecto de usar conversaciones integradas o *chatbots* para evaluar la experiencia del usuario y el efecto de estimar dicha experiencia a largo plazo mediante este método. La metodología empleada en el estudio se dividió en cuatro fases:

- **Planificación**, en esta fase los autores caracterizaron el problema, realizaron una investigación sobre las diferentes opiniones de usuario en otras aplicaciones de *mHealth*, además de realizar otra investigación sobre la literatura académica referente a la experiencia de usuario.
- **Elaboración del cuestionario**, en el segundo paso que siguieron, desarrollaron el cuestionario y realizaron una validación de las preguntas. En este apartado es importante mencionar que hacen uso de preguntas con respuesta abierta y cerrada, dependiendo de los datos que se quieren obtener.

  Según D. Biduski et al. "Nosotros elegimos usar preguntas abiertas en algunos momentos cuando queríamos capturar tantos detalles como fuera posible, dejado a los

usuarios libertad para expresarse. [...] cuando preguntamos a los usuarios que expliquen cosas, a menudo revelan sorprendentes modelos mentales y estrategias de resolución de problemas, compartiendo diferentes tipos de motivaciones y preocupaciones [3]."

En cambio, las preguntas de respuesta cerrada permitieron recoger información más concreta, además de ser preguntas más breves.

- **Periodo de pruebas**, se trató de la fase más larga y a su vez se subdividió en tres subfases, preguntas antes de tener experiencia en el sistema, durante y después de un periodo de tiempo en el que los usuarios habían acumulado experiencia. Esta fase se completó en tres meses e involucró a 37 participantes.
- **Después del periodo de pruebas**, en esta última fase se les realizó una entrevista a los usuarios en la que explicaron algunas de sus respuestas e impresiones globales y se concluyó con un mapeo de la información, es decir, un análisis de todas las fases previas.

Los resultados demostraron que las mejores experiencias se producían durante las primeras semanas, los factores estaban asociados a las funciones de la aplicación, gráficos y recursos visuales. En contraposición, los peores resultados se obtuvieron debido a factores como errores técnicos de la aplicación y dificultad en su uso. Por último, según D. Biduski et al, todos los usuarios apreciaron el uso de un personaje en la interfaz conversacional como recolector de las respuestas al cuestionario de evaluación.

2.2 System Usabity Scale (SUS)

El ***System Usability Scale (SUS)*** se trata de un cuestionario que evalúa el grado de usabilidad y que fue creado por John Brooke en 1986. El articulo original *SUS-A quick and dirty usability scale* [6] daba forma a uno de los cuestionarios más usados para evaluar este atributo, cabe decir que este cuestionario se puede usar no solo para comprobar la usabilidad de aplicaciones *software*, permite evaluar una amplia variedad de productos y servicios, incluidos *hardware*, dispositivos móviles y sitios web.

Consiste en un cuestionario de 10 preguntas con cinco opciones de respuesta para los participantes; siendo 5 totalmente de acuerdo (*Strongly agree*) y 1 totalmente en desacuerdo (*Strongly disagree*):

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.

8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

Las principales ventajas que ofrece son que es una escala muy fácil de administrar a los participantes, se puede utilizar en muestras pequeñas con resultados fiables y es válido, pues puede diferenciar eficazmente entre sistemas utilizables e inutilizables.

Por último, hace falta mencionar la forma de evaluación. Para las preguntas 1, 3, 5, 7 y 9 la puntuación variará de 0 a 4, ya que el valor se obtiene de la posición de la escala menos 1 (si en la pregunta 3 seleccionamos 4, el valor asociado a dicha cuestión sería 3). Para las cuestiones 2, 4, 6, 8 y 10 el resultado se obtiene de restar 5 menos la escala de la posición (si en la pregunta 2 seleccionamos 4, el valor asociado a dicha pregunta sería 1). La suma de todos resulta el grado de usabilidad, este comprende desde 0 hasta 40. Sin embargo, lo que se hace es normalizar el resultado, es decir, dar el resultado en base 100. Esto se consigue multiplicando el resultado por 2.5 (ver Figura 2.1).

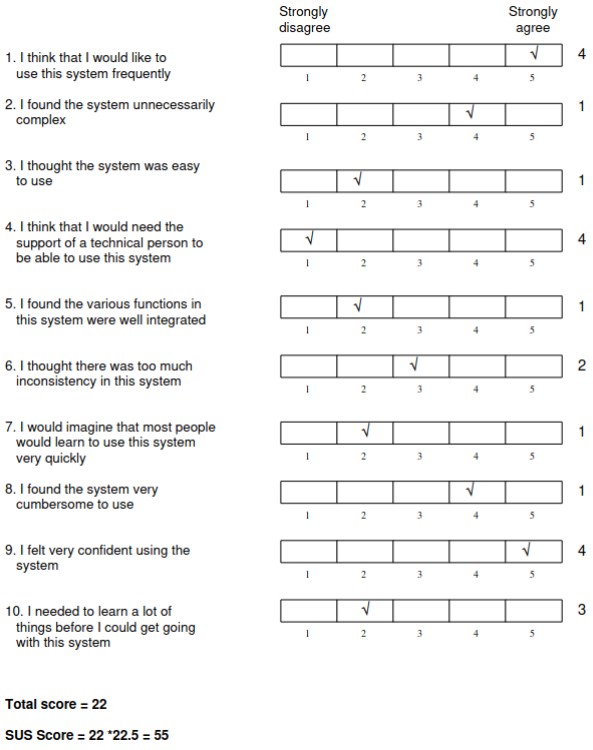

Figura 2.1: Ejemplo de calculo del cuestionario SUS

Según la página web [14], el resultado promedio es obtener una puntuación de 68 sobre 100, por ello obtener más de 68 puntos significaría estar haciendo uso de una herramienta, servicio, aplicación con una usabilidad muy buena, por encima del promedio. Por debajo de 68 estaríamos ante una usabilidad regular o mala, por lo menos, frente a otros competidores.

Según John Brooke [6], .[el] SUS, ha demostrado ser una valiosa herramienta de evaluación, robusta y confiable. Se correlaciona bien con otras medidas subjetivas de usabilidad

(por ejemplo, la subescala de usabilidad general del SUMI, inventario desarrollado en el proyecto MUSiC (Kirakowski, comunicación personal)). El SUS se ha puesto a disposición de forma gratuita para su uso en la evaluación de la usabilidad, y se ha utilizado para una variedad de de proyectos de investigación y evaluaciones industriales; el único requisito previo para su uso es que cualquier informe publicado debe reconocer la fuente de la medida."

2.3 User Experience Questionnaire (UEQ-S)

El ***User Experience Questionnaire (UEQ)* versión corta** [5], es un cuestionario creado por Martin Schrepp, Andreas Hinderks y Jörg Thomaschewski en 2017. La versión original fue desarrollada en Alemania durante los años 2006 y 2008. Se trata de un cuestionario ampliamente usado para evaluar la impresión subjetiva de los usuarios y traducido a varios idiomas.

El *UEQ* original es un cuestionario con 26 cuestiones semánticas (ver Figura 2.2). Finalizarlo se tarda aproximadamente entre tres y cinco minutos, es decir, el *UEQ* original ya es eficiente en cuanto al tiempo necesario para responder a todas las preguntas. Este se basa en un par de términos completamente opuestos (un término negativo y otro positivo) y un rango de 1 a 7. Según la impresión del usuario la escala de valores vendrá determinada por -3 (totalmente de acuerdo con el término negativo) a +3 (totalmente de acuerdo con el término positivo).

annoying	o o o o o o o	enjoyable	1
not understandable	o o o o o o o	understandable	2
creative	o o o o o o o	dull	3
easy to learn	o o o o o o o	difficult to learn	4
valuable	o o o o o o o	inferior	5
boring	o o o o o o o	exciting	6
not interesting	o o o o o o o	interesting	7
unpredictable	o o o o o o o	predictable	8
fast	o o o o o o o	slow	9
inventive	o o o o o o o	conventional	10
obstructive	o o o o o o o	supportive	11
good	o o o o o o o	bad	12
complicated	o o o o o o o	easy	13
unlikable	o o o o o o o	pleasing	14
usual	o o o o o o o	leading edge	15
unpleasant	o o o o o o o	pleasant	16
secure	o o o o o o o	not secure	17
motivating	o o o o o o o	demotivating	18
meets expectations	o o o o o o o	does not meet expectations	19
inefficient	o o o o o o o	efficient	20
clear	o o o o o o o	confusing	21
impractical	o o o o o o o	practical	22
organized	o o o o o o o	cluttered	23
attractive	o o o o o o o	unattractive	24
friendly	o o o o o o o	unfriendly	25

Figura 2.2: Lista de preguntas del UEQ original

El cuestionario, además, diferencia seis escalas: *atractivo*, *visibilidad*, *eficiencia*, *fiabilidad*, *estimulación* y *novedad*. Además, los diferencia entre aspectos pragmáticos (describen cualidades) y hedonísticos (relacionado con el placer). Según Martin S. et al. [6] "no se supone

que las escalas sean independientes. De hecho, la información general de un usuario, la impresión, se registra mediante la escala de atractivo, que debe ser influenciado por los valores de las otras 5 escalas (ver Figura 2.3)."

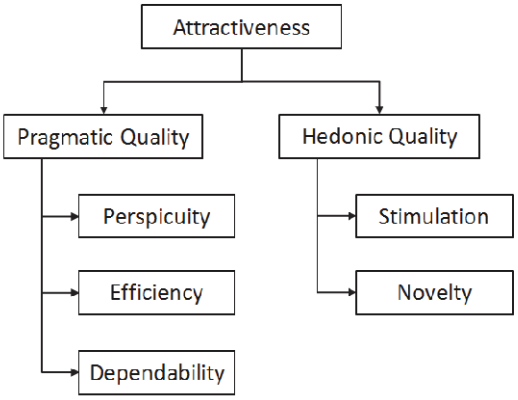

Figura 2.3: Escalas del UEQ

Sin embargo, hay situaciones que requieren un coste de tiempo más reducido. La versión corta proporciona una solución a este problema. Este artículo resume la creación de esta versión con tan solo 8 preguntas o ítems (ver Figura 2.4) y su validación [5].

obstructive	o o o o o o o	supportive	1
complicated	o o o o o o o	easy	2
inefficient	o o o o o o o	efficient	3
clear	o o o o o o o	confusing	4
boring	o o o o o o o	exiting	5
not interesting	o o o o o o o	interesting	6
conventional	o o o o o o o	inventive	7

Figura 2.4: Lista de preguntas del UEQ versión corta

Según los autores hay tres escenarios en los que se puede usar la versión corta del UEQ:

1. El primer escenario es la recopilación de datos cuando el usuario abandona una web, tienda o servicio web.

2. Segundo, incluirlo en una sección de experiencia de usuario muy breve dentro de un cuestionario de experiencia del cliente, ya que es a menudo la única forma de que los profesionales pueden recopilar comentarios sobre la experiencia de usuario de sus clientes.

3. El tercer escenario es cuando los escenarios son experimentales, donde se le pide a un participante que juzgue la experiencia del usuario de varios productos o variantes de un producto en una sesión.

2.4 Propuesta

Partiendo de la experiencia extraída de *Assessing long-term user experience on a mobile health application through an in-app embedded conversation-based questionnaire* [3], podemos concluir que crear un *chatbot* para medir la usabilidad y la experiencia de usuario es una idea de interés para la evolución de este campo. En consecuencia, para llevar a cabo este propósito hemos optado por estos dos cuestionarios ampliamente usados y validados por la comunidad científica en ámbitos informáticos, como lo es el desarrollo de una página web de apoyo a la decisión clínica.

Se ha optado por el *UEQ* versión corta ya que la situación se trata de una variante de tercer escenario en el que se puede usar esta versión más breve del *UEQ*. Así pues, nos encontramos frente a un escenario de pruebas en el que los usuarios van a ser los que juzguen el producto, asimismo, tendrá que realizar varias tareas y el cuestionario *SUS*. Hemos considerado que la versión escogida es la apropiada con el fin de no saturar, abrumar y cansar a la persona que realice la evaluación.

CAPÍTULO 3
Tareas y Cuestionarios

En este capítulo abordamos las cuestiones referentes al proceso de creación, desarrollo y modificación de las tareas y los cuestionarios. Para la implementación de las preguntas se han creado varios archivos de texto plano (.txt) en el *Backend*. Adicionalmente, los comentarios, explicaciones y las tareas a realizar se encuentran en un archivo de valores separados por comas (.csv), también ubicados en la misma capa. Estos archivos son leídos, procesados y cargados en la base de datos en el *Backend*, hay dos métodos específicos para automatizar esta tarea. Cada batería de preguntas se encuentra en castellano, valenciano e inglés. La explicación de la estructura de ficheros corresponde a capítulos posteriores.

3.1 Tareas

El ámbito de esta sección corresponde a las **tareas**. La idea de realizar estas tareas se debe a la necesidad de familiarizar al usuario con *Aleph*. No se puede evaluar la usabilidad ni la experiencia de usuario si este no ha sido capaz de adquirirla, aunque sea mínimamente.

Dado el estado de *Aleph*, que, como hemos mencionado anteriormente se encuentra en desarrollo y actualmente solo está disponible la versión de demostración, dado que la lista de funcionalidades está limitada por la naturaleza de la *demo* hemos diseñado tres tareas sencillas para que los usuarios puedan probar diferentes partes de la web.

3.1.1. Elección de las tareas

Estas tareas se han ideado y desarrollado siguiendo las recomendaciones del *grupo Nielsen Norman* [15], ampliamente especializado en evaluar la experiencia de usuario. Según este grupo encontramos tres directrices que deben tener las tareas.

1. Hacer que las tareas sean realistas.
2. Hacer que la tarea sea accionable.
3. No dar pistas ni pasos.

La primera de las indicaciones, hace referencia a que no tenemos que proponer tareas que en una situación normal un usuario no haría. Esto depende del tipo de usuarios que van a realizar las pruebas, ya que no es lo mismo hacer una prueba sobre una página web de venta de zapatillas que sobre una página web de apoyo a la decisión clínica, la clase usuarios encuestados cambiará mucho. Una tarea es realista cuando se trata de una tarea que el usuario realiza habitualmente.

En segundo lugar se trata de pedirles a los usuarios que realicen una acción, ya que resulta mejor que preguntarles cómo la harían. Preguntarles podría hacer que hablasen sobre lo que harían y no nos permitiría observar la facilidad o la frustración que conlleva el uso de la interfaz. Se trata de animar a los usuarios a interactuar con la interfaz.

En último lugar, no tenemos que dar pistas ni pasos. Las descripciones de los pasos a menudo contienen pistas ocultas sobre cómo utilizar la interfaz. Hacer uso de pasos en las tareas sesgan el comportamiento de los usuarios y aportan resultados menos útiles. Atendiendo a estas indicaciones, hemos propuesto estas tareas:

1. Si quieres usar Aleph vas a tener que registrarte.
2. Estudia los resultados de un caso totalmente aleatorio de cuidados paliativos.
3. Chequear la información relativa a Aleph.

3.1.2. Preguntas

Debemos recordar el estado de demostración de *Aleph* y que los usuarios están poco familiarizados con la interfaz. Es posible que las tareas cambien a lo largo de la vida del proyecto, ya que la dificultad puede variar, pero son un ejemplo factible para comprobar el correcto funcionamiento del bot. Estos factores se han elegido siguiendo las indicaciones de la **UNE-EN ISO 9241-11** [16].

- Objetivo
- Eficiencia
- Eficacia
- Satisfacción

La elección de estos factores tiene su razonamiento, con la **eficiencia** deseamos medir el tiempo que tarda cada usuario en concluir cada tarea. Con esto queremos comprobar que el usuario gaste una cantidad de tiempo razonable, un gasto excesivo podría deberse a fallos de la aplicación, una tarea muy compleja o a una interfaz poco clara y confusa.

El **objetivo** nos permite conocer si el usuario ha sido o no, capaz de finalizar la tarea con éxito y la **eficacia** saber si se han producido o el usuario ha encontrado errores en la aplicación. Estos factores nos permiten evaluar de una manera muy simple la usabilidad de las tareas y, en consecuencia, la de *Aleph*.

El factor de la eficacia se evaluará con una pregunta de respuesta abierta, aquí nos interesa conocer cualquier fallo, error o experiencia que haya tenido el usuario. Además,

si recordamos el estado del arte, primera sección, en el que se explicaba el artículo *Assessing long-term user experience on a mobile health application through an in-app embedded conversation-based questionnaire* [3], los factores que producían las malas experiencias eran principalmente dos, dificultad y fallos de la aplicación.

Ahora bien, con estas preguntas no podemos medir el grado de usabilidad. No obstante, recordemos que en el mismo artículo hacían mención al tipo de preguntas que se usaban dependiendo de los datos que querían recabar, en este caso hemos hecho uso de las preguntas de respuesta abierta para conocer cuáles son los principales fallos de la aplicación.

Conocer estos dos factores será crucial para que los desarrolladores de *Aleph* puedan solucionar problemas y mejorar progresivamente la aplicación, en pos de ofrecer una aplicación de mejores cualidades.

Finalmente, el último factor es la satisfacción en el que queremos saber si la tarea realizada ha sido del agrado del consumidor, si se ha sentido cómodo. O sea, evaluar de una manera simple la experiencia del usuario.

En conclusión, conforme al usuario se le propone una tarea, se le realizan varias preguntas que coinciden con cada uno de los factores mencionados, menos el tiempo, el cual se medirá internamente y el usuario no tendrá conocimiento explícito de la existencia de esta medición. Una vez que responda la primera pregunta el tiempo se podrá extraer de la base de datos de forma trivial mediante una resta, ya que se guarda la fecha de cuando se realiza cada pregunta. Preguntas a contestar por el usuario (ejemplo de funcionamiento (ver Figura 3.1)):

1. ¿Lo has conseguido?

2. ¿Has tenido algún problema? Si es así puedes comentarlos, escribe 'fin' cuando hayas acabado.

3. ¿Te ha resultado satisfactoria la tarea?

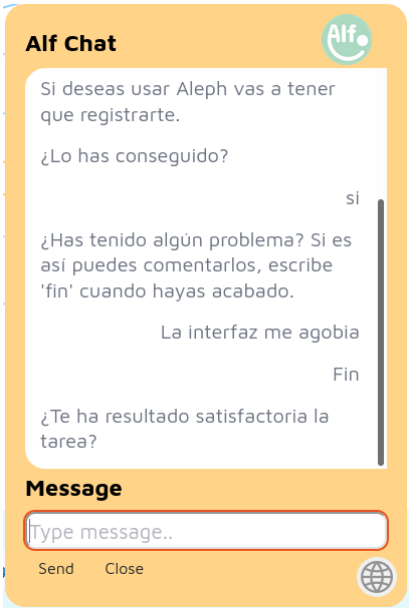

Figura 3.1: Ejemplo funcionamiento tareas

3.2 Cuestionarios

Si bien, con las preguntas propuestas en la sección anterior ya podemos empezar a recoger datos sobre la usabilidad y la experiencia de usuario no son suficientes, con dichas preguntas sobre usabilidad no nos permiten obtener el grado de usabilidad, solo saber si las tareas son asequibles y los fallos que puedan conllevar.

En esta sección abordaremos el desarrollo y la modificación de los cuestionarios **SUS** y **UEQ versión corta**, ya que ambos cuestionarios son una herramienta para arrojar datos cuantitativos y que complementarán los datos obtenidos de contestar las preguntas anteriores.

3.2.1. SUS

El *System Usability Scale (SUS)* [14] está compuesto por 10 ítems semánticos, estos están redactados en primera persona haciendo que el usuario se identifique con la propia frase y conteste en función de lo mucho o poco que esté de acuerdo con ella. Aquí tenemos las cuestiones originales traducidas al castellano [8]:

1. Me gustaría usar esta herramienta frecuentemente.
2. Considero que esta herramienta es innecesariamente compleja.
3. Considero que la herramienta es fácil de usar.
4. Considero necesario el apoyo de personal experto para poder utilizar esta herramienta.
5. Considero que las funciones de la herramienta están bien integradas.
6. Considero que la herramienta presenta muchas contradicciones.

7. Imagino que la mayoría de las personas aprenderían a usar esta herramienta rápidamente.

8. Considero que el uso de esta herramienta es tedioso.

9. Me sentí muy confiado al usar la herramienta.

10. Necesité saber bastantes cosas antes de poder empezar a usar esta herramienta.

Sin embargo, nosotros, al querer hacer uso de un *chatbot* que emule una conversación humana, no le podemos pedir al usuario que conteste cómo se siente respecto a una afirmación copiando palabra por palabra el cuestionario, tenemos que hacer sentir al usuario que está conversando con alguien. Seremos nosotros los que le haremos la pregunta. Así pues, cambiamos la primera persona por segunda y la afirmación por cuestión, adicionalmente, se ha optado por un registro más informal, hablando de tú a la persona, queriendo transmitir proximidad:

1. ¿Crees que te gustaría utilizar Aleph con frecuencia?

2. ¿Has encontrado la aplicación innecesariamente compleja?

3. ¿Consideras que Aleph es fácil de usar?

4. ¿Crees que necesitarías el apoyo de un técnico para poder utilizar esta aplicación?

5. ¿Consideras que las funciones de este sistema está bien integradas?

6. ¿Pensaste que había demasiadas contradicciones en Aleph?

7. ¿Imaginas que la mayoría de la gente aprendería a utilizar esta aplicación muy rápidamente?

8. ¿Te ha resultado tedioso utilizar este sistema?

9. ¿Te has sentido muy confiado al utilizar Aleph?

10. ¿Has necesitado aprender muchas cosas antes de poder empezar a usar Aleph?

Las preguntas vienen precedidas por un comentario clarificando el rango aceptable de las respuestas (ver Figura 3.2):

"Te voy a hacer unas preguntas sobre Aleph, debes contestar con 1 si estás muy en desacuerdo y 5 muy de acuerdo con la pregunta."

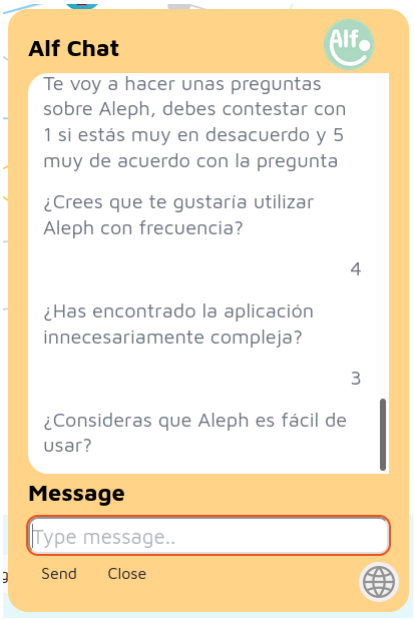

Figura 3.2: Ejemplo inicio del cuestionario SUS

3.2.2. UEQ-S

La modificación del *User Experience Questionnaire (UEQ)* se ha producido de forma muy leve, el UEQ es un cuestionario muy esquemático, en el que se proponen dos términos totalmente opuestos, en este caso hemos optado por conservar la mayor integridad posible del cuestionario, ya que alterar en exceso un cuestionario puede causar grandes diferencias con el original.

En este caso hemos buscado una traducción validada del UEQ [9] y se han reformulado como pregunta:

1. ¿Obstructivo o impulsor?
2. ¿Complicado o fácil?
3. ¿Ineficiente o eficiente?
4. ¿Claro o confuso?
5. ¿Aburrido o emocionante?
6. ¿No interesante o interesante?
7. ¿Convencional u original?

Las preguntas vienen precedidas por un comentario clarificando el rango aceptable de las respuestas (ver Figura 3.3):

"Te voy a proponer dos términos, tienes que contestar cómo has sentido la página web de Aleph, siendo 1 muy de acuerdo con el término de la izquierda y 7 muy de acuerdo con el de la derecha. ¡Vamos!"

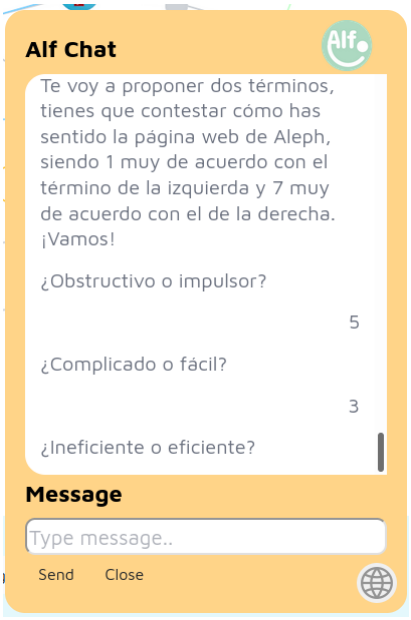

Figura 3.3: Ejemplo inicio del cuestionario UEQ

CAPÍTULO 4
Frontend

El objetivo de este apartado del *software* es crear la interfaz de usuario (*frontend*), o sea, la parte del sitio web con la que se va a interactuar. El resultado final puede apreciarse en las Figuras 4.3 y 4.4.

4.1 Chat

En esta parte del código del proyecto hemos implementado el chat, este está desarrollado sobre la misma base del código de *Aleph* que está construido en *Django*.

Django es un *framework* de desarrollo web de código abierto, programado en *Python* y con una arquitectura modelo-vista-controlador (MVC). *Aleph* hace uso de *Bootstrap* una librería de *Java Script*, *CSS* y *HTML* a la hora de programar y diseñar las páginas web.

Para esta sección hemos creado un fichero de estilo personalizado **chat-widget.css**, un fichero *JavaScript* **chat.js**, en el que se encuentra todos los métodos necesarios y algunos auxiliares, y se ha modificado **base.html** en el que hemos añadido todo el código html, **base.html** se trata de un html que contiene los elementos comunes de las diferentes pantallas de *Aleph*. (ver Apéndice A.1). Aquí se muestran los directorios donde se encuentra cada fichero (ver Figura 4.1).

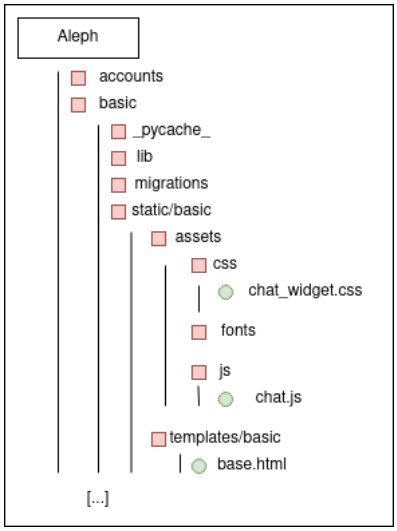

Figura 4.1: Estructura de ficheros Frontend

4.1.1. Métodos principales

- Los métodos **OpenForm** y **closeForm**: se encargan de abrir y cerrar la ventana de chat. Son métodos sencillos pero necesarios (ver Listing 4.1).

```
function openForm() {

  if (document.getElementById("balloonChatId").style.visibility == "visible") {
    document.getElementById("balloonChatId").style.visibility = "hidden";
  }

  document.getElementById("myForm").style.display = "block";
  document.getElementById("alfBtn").style.display = "none";
}

function closeForm() {
  document.getElementById("myForm").style.display = "none";
  document.getElementById("alfBtn").style.display = "block";
}
```

Listing 4.1: Métodos para desplegar el chat openForm y closeForm

- **getRespuesta**: Adquiere el valor del input, imprime el mensaje por pantalla y tras esperar un segundo, llama al método getHardResponse. Hemos puesto ese segundo de demora con el setTimeout para dar una sensación de naturalidad al usuario (ver Listing 4.2).

```
function getRespuesta() {

  let userText = $("#textInput").val();
  let userHtml = '<p class="userText">' + userText + '</p>';

  $("#textInput").val("");
  $("#chatBox").append(userHtml);

  updateScroll();

  setTimeout(()=> {
    getHardResponse(userText);
  }, 1000)

}
```

Listing 4.2: Método getRespuesta

- **getHardResponse**: Se trata de un método similar al anterior. En él, se imprime por la interfaz del chat todos los mensajes de la máquina de estados. Pueden ser de tipo comentario (*comment*) o de tipo texto (*text*). El bot siempre va a devolver un mensaje. Finalmente llamamos al método updateScroll. (ver Listing 4.3)

```
function getHardResponse(userText) {

  let response = getBotResponse(userText);

  let botComment = response['comment'];
  if (botComment != null) {
    let botComHtml = '<p class="botText">' + botComment + '</p>';
    $("#chatBox").append(botComHtml);
  }

  let botResponse  = response['text'];
  let botHtml='<p class="botText">' + botResponse + '</p>';

  $("#chatBox").append(botHtml);

```

```
16      updateScroll();
17
18    }
```

Listing 4.3: Método getHardResponse

- **getBotResponse**: Es el método más importante, se encarga de recoger los parámetros necesarios para que la máquina de estados procese el mensaje. Estos se regogen en un formato JSON:

  - **idSession**: Identificador del mensaje, nos permite conocer en qué parte del cuestionario se encuentra y actua como parte de la clave primaria para guardar las respuestas.
  - **text**: Inicialmente envía la respuesta del usuario.
  - **comment**: Está vacío, inicialmente.
  - **language**: Este parámetro indica el idioma según la variable lang.

  Acto seguido se establece la conexión con el backend, mediante una API de REST, una vez el backend procesa el mensaje, que se encuentra en text, este lo devuelve modificado, siendo ahora text la respuesta del bot y comment un comentario en caso de ser necesario, este campo puede volver vacío (ver Listing 4.4).

```
1   /**  Respuesta BOT  **/
2   function getBotResponse(input) {
3
4     var sessionId = getSessionId();
5     console.log(sessionId);
6
7     let answer = { "idSession": sessionId, "text": input, "comment": "", "language"
      : lang};
8
9     var xhr = new XMLHttpRequest();
10    var url = "http://127.0.0.1:7777/bot/";
11    xhr.open("POST", url, false);
12    xhr.setRequestHeader("Content-Type", "application/json");
13    var data = JSON.stringify(answer);
14    xhr.send(data);
15    var response = JSON.parse(xhr.responseText);
16    console.log(response);
17    return response;
18  }
```

Listing 4.4: Método getBotResponse

4.1.2. Métodos auxiliares

- **Métodos para el idioma**: En primer lugar está el metodo que despliega las tres opciones de idiomas disponibles, castellano, valenciano e inglés (ver Figura 4.2). Los siguientes tres métodos cambian el valor de la variable global del idioma, esta variable está inicializada con valor esp, es decir, en castellano (ver Listing 4.5).

Figura 4.2: Captura de chat con opciones de idiomas desplegadas

```
/** Seleccionar Idioma */

function openSelectorButton() {
  if(document.getElementById("langSelector").style.display == "block") {
    document.getElementById("langSelector").style.display = "none";
  }
  else {
    document.getElementById("langSelector").style.display = "block";
  }
}

function changeLangButtonEng() {
  lang = "eng";
  document.getElementById("langSelector").style.display = "none";
  document.getElementById("presentationText").innerHTML = "Hello, you're going to
   participate in a usability test. All your answers are correct Thank you for
  participating!";
}

function changeLangButtonEsp() {
  lang = "esp";
  document.getElementById("langSelector").style.display = "none";
  document.getElementById("presentationText").innerHTML = "Hola vas a participar
  en una prueba de usabilidad. Todas tus respuestas son correctas Gracias por
  participar!";
}

function changeLangButtonVal() {
  lang = "val";
  document.getElementById("langSelector").style.display = "none";
  document.getElementById("presentationText").innerHTML = "Hola, participaras en
  una prova d'usabilitat. Totes les teues respostes son correctes Gracies per
  participar!";
}
```

Listing 4.5: Métodos para cambiar el idioma

- **sendButton**: Este método auxiliar ejecuta el método principal getRespuesta, este se activa cuando se pulsa sobre el botón send de la ventana del chat. La segunda parte del método se encarga de hacer click sobre el botón cuando se pulsa la tecla intro (*enter*) (ver Listing 4.6).

```
function sendButton() {
  getRespuesta();
}

$("#textInput").keypress(function(e) {
  if(e.which == 13) {
    e.preventDefault();
    document.getElementById("sendBtn").click();
  }
});
```

Listing 4.6: Método para activar getRespuesta()

- **getSessionId**: Este método recupera la cookie *sessionid*, que sirve como identificador del usuario (ver Listing 4.7).

```
/**  Get cookie sessionid  **/
function getSessionId() {
  var jsId = document.cookie.match(/sessionid=[^;]+/);
  if(jsId != null) {
    if (jsId instanceof Array)
        jsId = jsId[0].substring(10);
    else
        jsId = jsId.substring(10);
  }
  console.log(jsId);
  return jsId;
}
```

Listing 4.7: Método para recuperar la cookie

- **updateScroll**: Se utiliza para mantener actualizado el chat cuando aparece un mensaje nuevo. De otra forma el usuario debe bajar manualmente el scroll del chat (ver Listing 4.8).

```
function updateScroll(){
  var element = document.getElementById("chatBox");
  element.scrollTop = element.scrollHeight;
}
```

Listing 4.8: Método para actualizar el chat

- **Métodos para el bocadillo o globo**: Es un detalle meramente estético, no cumple ninguna funcionalidad a parte de generar una buena impresión en el usuario. En el diseño abordaremos la explicación del uso de este elemento (ver Listing 4.4).

```
/** Hover bocadillo chatbot */

if (document.getElementById("balloonChatId").style.visibility == "visible") {
  setTimeout(function(){ document.getElementById("balloonChatId").style.
  visibility = "hidden"; }, 5000);
}

document.getElementById("alfBtn").addEventListener("mouseover", balloonChat);

function balloonChat() {
  document.getElementById("balloonChatId").style.visibility = "visible";
  setTimeout(function(){ document.getElementById("balloonChatId").style.
  visibility = "hidden"; }, 5000);
}
```

Listing 4.9: Métodos para ocultar/descubrir el bocadillo

4.2 Diseño

El objetivo principal que hemos buscado realizando el diseño es la cohesión e integración con el proyecto predecesor *Aleph*. También, en su conjunto hemos querido transmitir al usuario una sensación amable y de cercanía, de acuerdo con el objetivo del proyecto que debe ser crear una interfaz agradable con el usuario (ver Figuras 4.3 y 4.4). Asimismo, para dotar de personalidad propia al módulo, hemos decidido llamarle *Alf*, por su similitud con *Aleph* (en un principio se le llamó *Alph*) y en referencia al icónico extraterrestre del planeta Melmac, que aparecía en la serie homónima de la televisión en los 80.

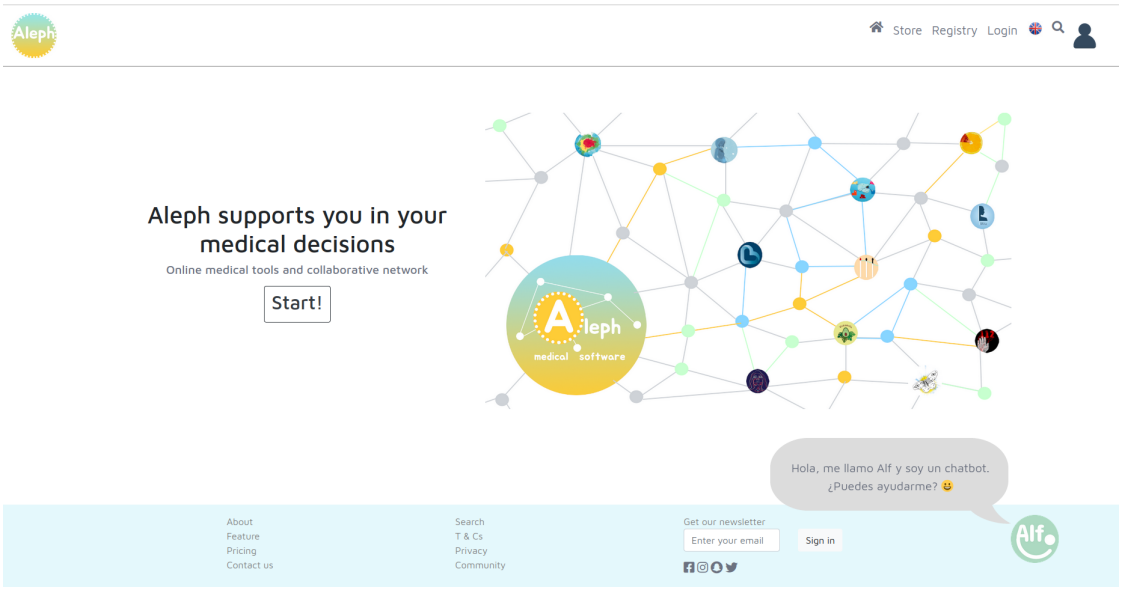

Figura 4.3: Captura de pantalla con Alf sin desplegar

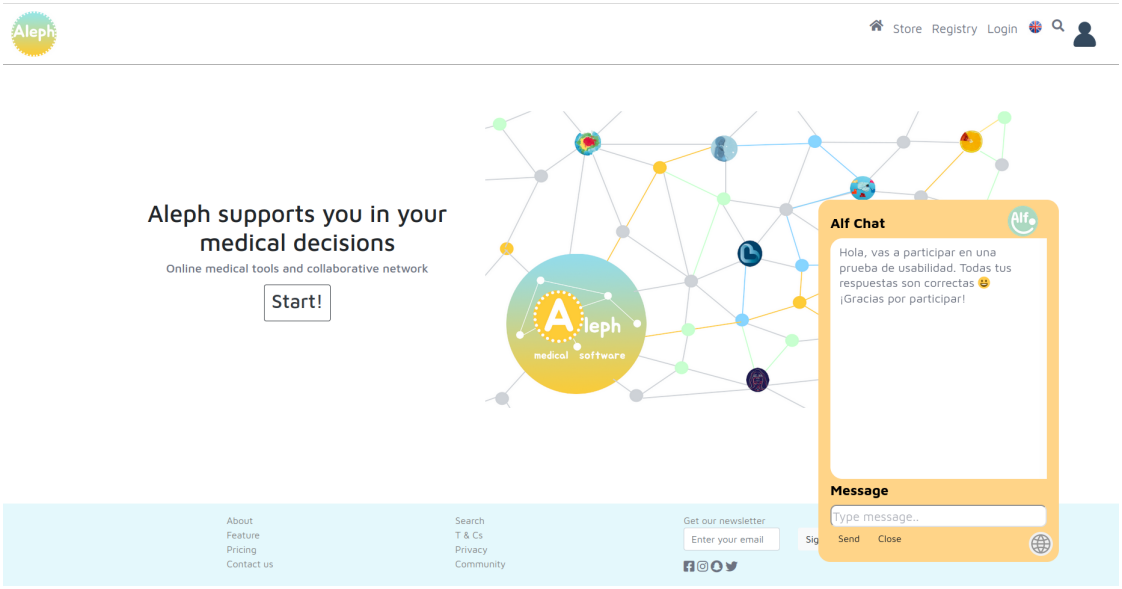

Figura 4.4: Captura de pantalla con Alf desplegado

4.2.1. Color, formas y bocadillo

La elección del diseño se ha realizado de forma cuidadosa y concienzuda, de acuerdo con lo que se quiere transmitir, el diseño final ha sido fruto de una evolución en la que hemos propuesto diferentes colores y estilos.

- **La paleta de colores** (ver Figura 4.5) se ha formado siguiendo la combinación de colores de la página principal en la que predominan amarillos y verdes, la cual resulta bastante viva y alegre, nosotros hemos optado por la elección de estos mismos colores, pero menos saturados, unos tonos pastel para que sean más armónicos entre sí y transmitan esa sensación de cercanía que buscamos.

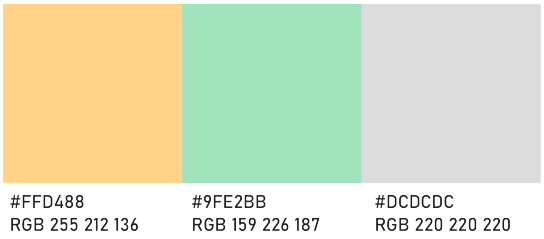

Figura 4.5: Paleta de colores de Alf

- **Formas redondeadas** (ver Figura 4.6), estas formas se repiten a lo largo de la página web ya sea con la elección de tipografía con bordes suavizados como con la elección de círculos como elementos constructivos, por ello al realizar la ventana del chat se ha optado por continuar con este motivo.

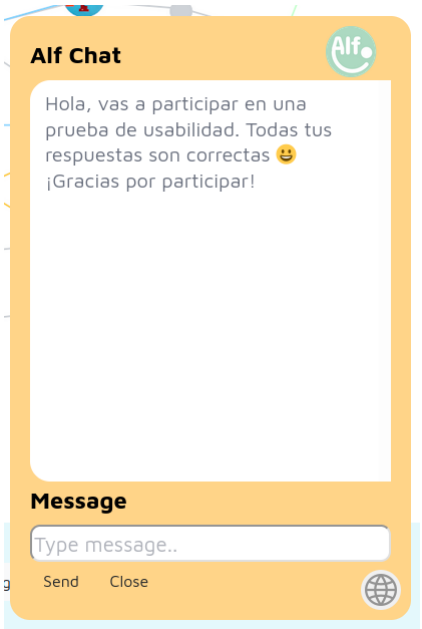

Figura 4.6: Chat abierto

- **Bocadillo**, este elemento es una imagen *.png* vectorizada en *Adobe Ilustrator*, inspirada en PAU (el bot de la página web de la UPV), con el que hemos tratado de dar solución a un problema de percepción que encontramos, que tan solo con el icono del chat, al usuario no le quedaba claro que el círculo situado en la parte inferior

derecha fuese un botón que abriese la ventana de chat. Por tanto, hemos tratado de darle un elemento que le diese presencia y personalidad a **Alf** (ver Figura 4.7).

Figura 4.7: Chat cerrado con bocadillo

4.2.2. Diseño responsive

Pese a que la librería de *Bootstrap* es *responsive* al hacer uso de una plantilla totalmente personalizada para lograr esta característica, hemos tenido que implementar nuestro propio código de CSS. Decimos que una web es responsive cuando, en función de las dimensiones del dispositivo desde el que se consulta, dicha web adapta su diseño para que mantenga coherencia respecto al propio dispositivo.

Hemos divido las dimensiones de la plantilla del chat en tres contenedores siendo el ordenador el dispositivo por defecto en el que se va a usar. Luego hay dos media queries que actúan como contenedores de las reglas de los tamaños (ver Listing 4.10) . Así pues, tenemos unos atributos de estilo específicos para cada uno de los tres dispositivos disponibles: ordenador, tableta y móvil.

```
/*MEDIA QUERIES*/

/*MEDIA QUERY TABLETS*/
@media (min-width: 481px) and (max-width:768px) {

  .labelChat {
    font-size: 1.8rem;
  }

  .chatContainer {
    right: 3rem;
  }

  .open-chat-button {
    width: 4.5rem;
    height: 4.5rem;
    background-size: 6.7rem;
  }

  .alftext {
    right: 5rem;
    bottom: 3rem;
    width: auto;
    height: auto;
  }

  .presentation {
    width: 19rem;
    height: 4.5rem;
  }

  .presemtationTextSize {
    font-size: 1rem;
  }

```

```
36    .balloon {
37      width: 20rem;
38      height: 6rem;
39    }
40
41  }
```

Listing 4.10: Media query para dispositivos de tipo Tablet

4.2.3. Logo

El último apartado del diseño corresponde al logo de **Alf**. Para la creación del logo del *chatbot* se tuvieron en cuenta varias ideas. La primera fue crear un diseño antropomórfico [7] (en este artículo se detalla la importancia de elegir este tipo de diseño), un diseño que emulase una forma humana, por ello creamos un icono similar a un emoticono con una sonrisa (ver Figura 4.8). Este proceso de creación se ha llevado a cabo con diferentes pruebas de logos, primero creamos bocetos en papel y posteriormente se vectorizaron haciendo uso de *Adobe Ilustrator*.

Figura 4.8: Logo Alf primer diseño simulando emoticono de sonrisa

Sin embargo, después de varias pruebas se comprobó que al reducir el tamaño no funcionaba bien, debido al alto grado de detalle, por tanto optamos por un diseño más simplificado y sintetizado. Así se unió también la segunda idea, que es la de conexión y cercanía, inspirada en el fondo de la página principal, donde hay varios puntos interconectados. Dando pie a todos estos diseños (ver Figuras 4.9, 4.10 y 4.11).

Figura 4.9: Logo Alf diseño alternativo 1

Figura 4.10: Logo Alf diseño alternativo 2

Figura 4.11: Logo Alf diseño alternativo 3

Tras varias pruebas de color, para ver cual era la formula que mejor funcionaba, se optó por la decisión final (ver Figura 4.12) . Para el icono hemos usado la tipografía Concert One, la cual se trata de una tipo San Serif con los bordes redondeados pero con carácter más geométrico para diferenciarse de la tipografía de *Aleph*.

Figura 4.12: Logo Alf diseño definitivo

Lo que hemos buscado con esta diferenciación es dejar intuir al usuario que se trata de un trabajo independiente a *Aleph* pero, que a su vez, mantiene una estrecha relación con él, manteniendo unos colores similares que preservan esa conexión.

CAPÍTULO 5
Backend

En este capítulo trataremos todos los engranajes de **Alf**, desde que el usuario envía el mensaje a través de una **API de REST** hasta que el bot devuelve una respuesta, pasando por la interacción con la base de datos y el procesamiento del mensaje. Este capítulo puede considerarse el más importante, aquí hemos desarrollado la mayor parte del código y la funcionalidad principal del módulo, recogiendo la información proporcionada por los usuarios para su posterior análisis y mejora de *Aleph*.

Para llevar a cabo esta tarea hemos creado una **API de REST**, usando el *framework* **FastAPI** [12], una base de datos, en nuestro caso una **MySQL 8.0.26** [17], un *Object-Relational Mapper (ORM)* para interaccionar con la base de datos, hemos elegido **SQLAlchemy** [13] y, finalmente, la máquina de estados escrita en *Python 3*. Cabe decir que tanto **FastAPI** como **SQLAlchemy** usan *Python* como lenguaje, este factor común se ha buscado para no ampliar la cantidad de lenguajes ya usados y mantener un mismo lenguaje en la implementación del *Backend*.

Además, como ya mencionamos anteriormente, es aquí donde se encuentran las tareas, preguntas y cuestionarios. Tanto las preguntas como cuestionarios se encuentran en varios ficheros .txt (**1task1.txt, 2task2.txt, 3task3.txt, 4sus.txt** y **5ueq.txt**) y se hallan en la carpeta *questions*. Existe una carpeta diferente para cada idioma posible, inglés, valenciano y español, con los ficheros traducidos. Las **tareas y los comentarios**, como las aclaraciones de rangos aceptables de una respuesta, se hallan en la carpeta *strings* en forma de fichero .csv, también uno por cada idioma (**eng.csv**, **esp.csv** y **val.csv**).

En esta misma carpeta *strings* se encuentran los ficheros **affirmations.txt** y **negations.txt**, donde se recogen todas las formas posibles de responder a una pregunta de tipo afirmativo o negativo: *s, sí, si, y, yes, n* y *no*. Fuera de esta carpeta junto con el resto de los ficheros *python* encontramos el fichero **ranges.csv**, el cual contiene los rangos de las respuestas aceptables para cada pregunta (ver Listing 5.1).

```
task,question,type,low,high,comment
1,1,affirmation,0,1,
1,2,text,0,280,
1,3,affirmation,,1,
2,1,affirmation,0,1,
2,2,text,0,280,
2,3,affirmation,0,1,
3,1,affirmation,0,1,
3,2,text,0,280,
3,3,affirmation,0,1,
4,1,int,1,5,
```

```
4,2,int,1,5,
4,3,int,1,5,
4,4,int,1,5,
4,5,int,1,5,
4,6,int,1,5,
4,7,int,1,5,
4,8,int,1,5,
4,9,int,1,5,
4,10,int,1,5,
5,1,int,1,7,
5,2,int,1,7,
5,3,int,1,7,
5,4,int,1,7,
5,5,int,1,7,
5,6,int,1,7,
5,7,int,1,7,
```

Listing 5.1: Fichero con rangos aceptables de respuesta ranges.csv

Aquí podemos ver la estructura de archivos de la capa **Service** o **Backend** (ver Figura 5.1):

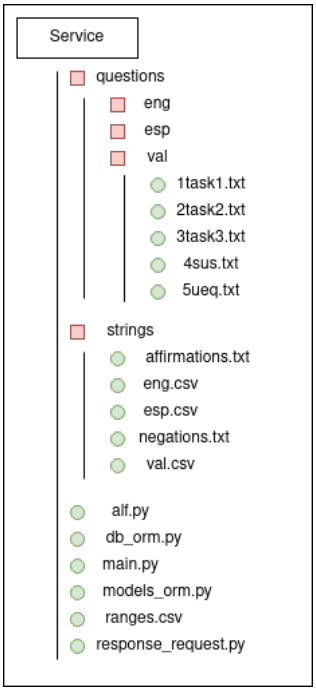

Figura 5.1: Estructura de ficheros Backend

5.1 API de REST

En esta sección desarrollamos el API de REST, implementado mediante el *framework* web **FastAPI** [12], este se compone de dos clases, **main.py** y **response_request.py**. Hemos elegido este *framework* debido a que nos permite hacer una implementación rápida y sencilla escrita en *Python*. Ahora bien, antes de pasar a explicar estas clases debemos entender qué es una API de REST.

Una API de transferencia de estado representacional (REST), o API de RESTful, es una interfaz de programación (API) que permite interaccionar con los servicios web de RESTful; su uso es muy común hoy en día. Permite separar los componentes de una aplicación dejando en un lado la interfaz web y en el otro toda la lógica de la aplicación, en

nuestro caso del módulo chatbot.

Las solicitudes a través de nuestra API de REST se entregan por medio de HTTP en formato JSON. FastAPI usa *Uvicorn* (una implementación de servidor ASGI muy rápida) para el despliegue en el servidor.

En primer lugar tenemos la clase **response_request.py** (ver Listing 5.2), en esta clase auxiliar encontramos el modelo **ResponseRequest**, este incluye los parámetros que debe contener la solicitud, si recordamos envuelve los mismo parámetros que definíamos en el método **getBotResponse()** del *Frontend*.

```
from pydantic import BaseModel

class ResponseRequest(BaseModel):
    idSession: str
    text: str
    comment: str
    language: str
```

Listing 5.2: Clase response_request.py

En segundo lugar encontramos la clase **main.py** (ver Listing 5.3), la instanciación de la API de REST. Aquí también se instancian las tablas de la bases de datos (método **create_tables**) y se inicializa la tabla **QUESTIONS** mediante la llamada al método **load_questions**. Finalmente, encontramos el encabezado de la llamada, aquí llega un JSON enviado por el usuario, que contiene: el id, el texto de la respuesta, el idioma y el campo de los comentarios vacío. Acto seguido, llamamos al método **process_message()** y le pasamos como parámetro el ítem de tipo **ResponseRequest**. Una vez procesado el mensaje en la clase **alf.py**, este ítem será devuelto, habiendo sido modificado de tal forma que el campo del comentario puede incluir un mensaje y el campo del texto incluirá la pregunta a realizar.

```
from typing import Optional

from sqlalchemy.sql.sqltypes import VARCHAR

from fastapi import FastAPI
from pydantic import BaseModel
from fastapi.middleware.cors import CORSMiddleware
from alf import process_message, load_questions, create_tables
from response_request import ResponseRequest

app = FastAPI()

origins = ["*"]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

create_tables()

load_questions()



@app.post("/bot/")
async def process_response(item: ResponseRequest):
    print(item.text)
```

```
32     response = process_message(item)
33     return response
```

Listing 5.3: API de REST clase main.py

Para desplegar el API de REST en el servidor. Hacemos uso de esta instrucción en la terminal:

```
1  $ uvicorn main:app --reload --7777
```

Listing 5.4: Instrucción para levantar la API de REST

5.2 Tratamiento de base de datos

El objetivo de este fragmento de *software* es guardar las preguntas, las respuestas y el estado en el que se encuentra el usuario en cada momento en una base de datos en función de las contestaciones de el usuario. A continuación enunciaremos los pasos en los que se estructura y se interacciona con la base de datos.

5.2.1. Estructura de la base de datos

A la hora de elegir una base de datos nos para almacenar toda esta información no hemos decantado por **MySQL 8.0.26** [17] ya que es una base de datos potente y portable. Su uso está ampliamente reconocido, se trata de uno de los motores de bases de datos relacionales más usados. Además es compatible con **MariaDB** o **sqlite3** por lo que es fácilmente reemplazable. Cabe mencionar que los desarrolladores de *Aleph* usan MariaDB por lo que aunque en el proyecto se haya usado una MySQL los desarrolladores podrán cambiarla sin problema en caso de preferir el otro motor.

Para la implementación de la base de datos se ha hecho uso de la imagen oficial de **MySQL** (ver Apéndice C.1) de **Docker Hub** [19]. Una vez ejecutamos el fichero Docker-Compose en el que se encuentra la imagen se crea el contenedor con nuestra propia base de datos.

Se han diseñado tres tablas SQL para poder almacenar los datos necesarios de la máquina de estados. **STATUS**, contiene la información de cada usuario que está realizando el cuestionario. **QUESTIONS**, almacena todas las preguntas a realizar. Y por último, tenemos la tabla **RESPONSES**, esta guarda todas las respuestas generadas por el usuario. Aquí vemos su estructura (ver Figura 5.2):

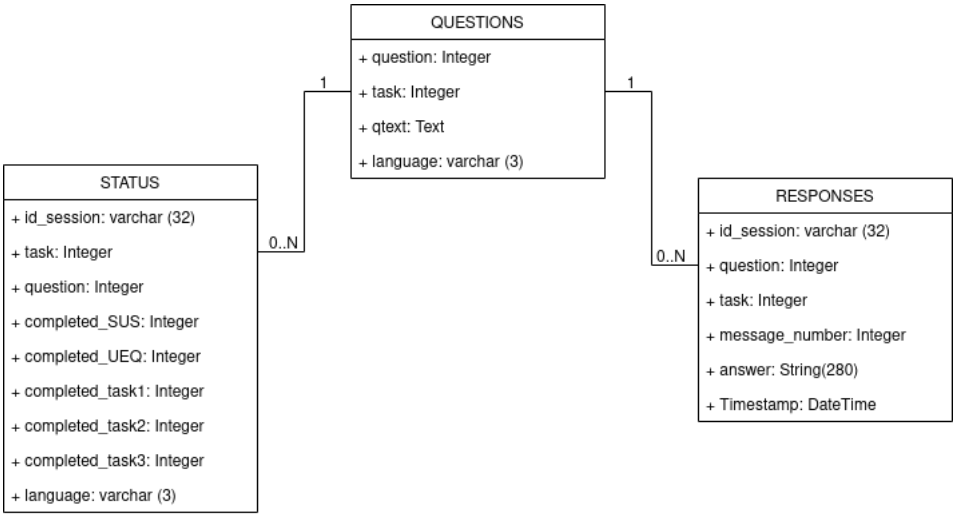

Figura 5.2: Diagrama de UML de la base de datos

5.2.2. ORM

Para interaccionar con la base de datos teníamos dos opciones, hacer uso de las librerías por defecto de *Python* para manejar consultas sobre las bases de datos o utilizar una **ORM** (de sus siglas en inglés *object-relational mapper*). Un ORM es una biblioteca de código que automatiza la transferencia de datos almacenados en tablas de bases de datos relacionales a objetos que se usan más comúnmente en el código de la aplicación. En este apartado veremos cómo se implementa y en la siguiente sección, la máquina de estados, cómo se usa.

Resulta de interés ya que permite una mayor abstracción en el código al no tener que escribir la consulta SQL y da un acabado más profesional al código. Ha resultado extremadamente útil ya que gracias a esto no hemos necesitado crear métodos exclusivos en los que se lancen consultas concretas, sino que con una sentencia escrita en *Python* obteníamos el mismo resultado ahorrando lineas y métodos del código.

Las principales ventajas de usar un ORM son:

- Podemos acceder a las tablas y filas de la base de datos como clases y objetos.
- No es necesario usar el lenguaje SQL. El ORM se encarga de traducir el código a SQL, aún así se pueden seguir ejecutando consultas dentro del propio código de la ORM.
- Independencia de la base de datos. Y que es posible cambiar de motor de base de datos modificando mínimamente el código de la aplicación.

En nuestro caso nos hemos decantado por **SQLAlchemy** [13], ya que es compatible con la mayoría de bases de datos relacionales conocidas: MySQL, MariaDB, Sqlite3, PostgreSQL, etc., e implementa múltiples patrones de diseño que te permiten desarrollar aplicaciones rápidamente, además de abstrarte de ciertas tareas, como manejar el pool de conexiones a la base de datos.

En el siguiente código *software* vemos la creación de este ORM, lo primero es la creación del *engine*, que es la entrada a la base de datos, dicho de otra forma, es lo que permite a SQLAlchemy establecer la conexión con la base de datos, en nuestro caso la MySQL. Otra ventaja de hacer uso de esta ORM es que cada motor de SQL tiene su propio dialecto, es decir, tiene variaciones sobre el lenguaje SQL. El *engine* configura esto por nosotros de forma automática.

Lo siguiente que debemos crear es una sesion, que vendría a ser un conjunto de operaciones sobre la base de datos que se ejecutan de forma atómica, en el caso de que falle una o varias, estas no se ejecutarían. Por último, queda implementar los modelos, que son las clases, lo que vendrían a ser las tablas en la base de datos, sin embargo, estas deben heredar de la clase Base, intanciada gracias al meétodo **declarative_base()**. Todo este código se encuentra en la clase **db_orm.py** (ver Listing 5.5).

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine('mysql://user:example@0.0.0.0:3306/db')

Session = sessionmaker(bind=engine)
session = Session()

Base = declarative_base()
```

Listing 5.5: Clase instanciación ORM db_orm.py

Una vez tenemos la clase Base podemos comenzar a crear los modelos. Las clases **Status** (ver Listing 5.6), **Questions** (ver Listing 5.7) y **Responses** (ver Listing 5.8) representan las tablas vistas en la sección anterior. Crear una clase modelo lo primero que hay que especificar es la tabla a la que hace referencia mediante el atributo de clase "**__tablename__**".

Cada una de las **columnas** de la tabla se corresponden en la clase a través de atributos de tipo Column, en estos atributos se declaran los tipos y los atributos que debe tener la columna a la que hace referencia. Así pues, el atributo de la clase Status **id_session** se corresponde con la columna homónima, cuyo tipo de datos es de tipo VARCHAR con un límite de 32 caracteres y que es la clave primaria.

```
from sqlalchemy.sql.schema import ForeignKeyConstraint
from sqlalchemy.sql.sqltypes import VARCHAR
import db_orm

from sqlalchemy import Column, Integer, String, Float, VARCHAR, Text, DateTime,
    PrimaryKeyConstraint, UniqueConstraint

#### TABLA STATUS ####
class Status(db_orm.Base):

    __tablename__ = 'STATUS'

    id_session = Column(VARCHAR(32), primary_key=True)
    task = Column(Integer, nullable=False)
    question = Column(Integer, nullable=False)
    completed_SUS = Column(Integer)
    completed_UEQ = Column(Integer)
    completed_task1 = Column(Integer)
    completed_task2 = Column(Integer)
    completed_task3 = Column(Integer)
    language = Column(VARCHAR(3))

```

```
    __table_args__ = (
        ForeignKeyConstraint(['question', 'task'], ['Questions.question', 'Questions.
    task'])
    )

    def __init__(self, id_session, task, question, language, completed_SUS=0,
    completed_UEQ=0, completed_task1=0, completed_task2=0, completed_task3=0):
        self.id_session = id_session
        self.task = task
        self.question = question
        self.completed_SUS = completed_SUS
        self.completed_UEQ = completed_UEQ
        self.completed_task1 = completed_task1
        self.completed_task2 = completed_task2
        self.completed_task3 = completed_task3
        self.language = language
```

Listing 5.6: Modelo ORM para tabla STATUS

En nuestro caso, las tablas **QUESTIONS** y **RESPONSES** deben hacer uso de una **clave primaria compuesta** por una tupla de ítems, esto es así porque, por ejemplo, en la tabla QUESTIONS van a existir varias preguntas con valor question igual a uno, pero no van a existir más de una pregunta numero uno, de la tarea uno con idioma español. Esta característica, y si existe alguna tupla que no pueda tener valor repetido, se le proporciona en el atributo "**__table_args__**".

```
        return f'Status({self.id_session}, {self.task}, {self.question}, {self.
    completed_SUS}, {self.completed_UEQ}, {self.completed_task1}, {self.completed_task2
    }, {self.completed_task3}, {self.language})'

    def __str__(self):
        return self.id_session

#### TABLA QUESTIONS ####
class Questions(db_orm.Base):

    __tablename__ = 'QUESTIONS'

    question = Column(Integer)
    task = Column(Integer)
    qtext = Column(Text)
    language = Column(VARCHAR(3))

    __table_args__ = (
        PrimaryKeyConstraint('question', 'task','language'),
        UniqueConstraint('question', 'task', 'language'),
    )

    def __init__(self, question, task, qtext, language):
        self.question = question
        self.task = task

        self.qtext = qtext
        self.language = language
```

Listing 5.7: Modelo ORM para tabla QUESTIONS

Antes de continuar con el uso de la ORM en la siguiente sección, debemos pararnos a estudiar el caso de la tabla **RESPONSES**, como vemos existe un atributo de la clase llamado **message_number** (ver Listing 5.8), que pertenece a la tupla de ítems que conforman la clave primaria. Este atributo está creado a conciencia para el caso de la pregunta de respuesta abierta. Cuando ideamos esta pregunta sabíamos que requeriría de una implementación especial solo para este caso. Existe la posibilidad que el usuario en vez de enviar los errores en una sola respuesta quiera enviarlos en diferentes mensajes, por ello no servía la tupla de sesión, pregunta y tarea, tendríamos varias respuestas con misma clave primaria y valor único. Necesitábamos crear este atributo para dar solución al pro-

blema, este incrementará en función de la cantidad de respuestas reciba.

```
        return f'Questions({self.question}, {self.task}, {self.qtext}, {self.language})'

    def __str__(self):
        return self.question

#### TABLA RESPONSES ####
class Responses(db_orm.Base):

    __tablename__ = 'RESPONSES'

    id_session = Column(VARCHAR(32))
    question = Column(Integer)
    task = Column(Integer)
    message_number = Column(Integer)
    answer = Column(String(280))
    Timestamp = Column(DateTime)


    __table_args__ = (
        PrimaryKeyConstraint('id_session','question', 'task', 'message_number'),
        ForeignKeyConstraint(['question', 'task'], ['Questions.question', 'Questions.
    task'])
    )

    def __init__(self, id_session, question, task, message_number, answer, Timestamp):
        self.id_session = id_session
        self.question = question
        self.task = task
        self.message_number = message_number
        self.answer = answer
        self.Timestamp = Timestamp
```

Listing 5.8: Modelo ORM para tabla RESPONSES

5.3 Maquina de estados

Este apartado comprende todo el código de **alf.py**, se trata de la clase donde está implementada la máquina de estados, además veremos el uso de la ORM.

5.3.1. Métodos auxiliares

En este apartado se encuentra el método para crear las tablas, los métodos de lectura, es decir, los que se encargan de cargar las preguntas, ítems o comentarios y los métodos con las funciones principales de la máquina de estados, no confundir con la propia máquina de estados, estos últimos métodos son los encargados de comprobar si la respuesta emitida por el usuario es valida, incrementar el contador de la tabla STATUS y almacenar la respuesta en la base de datos.

En primer lugar tenemos el método **create_tables** (ver Listing). Si recordamos este método aparecía en el API de REST, esto es así porque una vez la levantamos debemos crear las tablas. Este sencillo método ejecuta una simple sentencia de la ORM, la cual crea, en caso de no estar creadas las tablas de la base de datos en función a la clases de modelos creados en **models_orm.py**.

```
# Metodo inicializacion #
```

```
def create_tables():
    db_orm.Base.metadata.create_all(db_orm.engine)
```

Listing 5.9: Método para inicializar las tablas

Ahora bien, antes de continuar definiendo los métodos, debemos comentar las variables globales. Para el correcto funcionamiento de la máquina de estados hemos creado cuatro variables globales o, mejor dicho, cuatro diccionarios globales: **rules**, **n_questions_per_answer**, **multiple_answer_cache** y **comm**.

- **rules**: se trata del diccionario que almacena los datos del archivo **ranges.csv**
- **n_questions_per_answer**: es un diccionario que contiene el número de preguntas por tarea y cuestionario.
- **multiple_answer_cache**: diccionario cuya clave primaria es el identificador de sesión del usuario, y cuyo valor será el atributo message_number de la clase Responses (columna message_number de la tabla RESPONSES).
- **comm**: este diccionario que guarda los datos de los archivos **eng.csv**, **esp.csv** y **val.csv**. Es decir, los archivos contenedores de comentarios.

Los siguientes métodos son los denominados de lectura: **load_questions**, **get_rules**, **get_comment** y en ultima instancia **n_questions_per_task**. El primero, **load_questions** (ver Listing 5.10), también se encuentra en la API de REST, este método se encarga de guardar las preguntas en la tabla QUESTIONS. Este accede al directorio en el que se encuentran los ficheros .txt, los recorre y almacena cada una de sus preguntas.

```
def load_questions():
    for lang in listdir('questions'):
        try:
            for i, task in enumerate(sorted(listdir('questions/' + lang))):
                with open('questions/' + lang + '/' + task, 'r', encoding='utf-8') as
    file:
                    b = 0
                    lines = file.read().split('\n')
                    for j, line in enumerate(lines):
                        if line.strip(): #Elimina espacios al principio y al final del
    string
                            arroz = Questions(j + 1 - b, i + 1, line.strip(), lang)
                            db_orm.session.add(arroz)
                            db_orm.session.commit()
                        else:
                            b += 1
        except:
            db_orm.session.rollback()
        finally:
            db_orm.session.close()
```

Listing 5.10: Método para cargar las preguntas en la base de datos load_questions

Los métodos **get_rules** y **get_comment** (ver Listing 5.11) funcionan de manera similar, hacen uso de la biblioteca de *software pandas* y del módulo CSV para la lectura de los ficheros .csv. En el primero vemos que solamente tiene que acceder a un fichero, y con un bucle, lo recorre e indexa el diccionario. Sin embargo, en el segundo hemos implementado otro bucle para que sea capaz de recorrer los ficheros que contienen los comentarios en todos los idiomas. Finalmente, ambos métodos devuelven un diccionario, diccionarios globales vamos a inicializar con estos métodos.

```python
#cargamos reglas csv
def get_rules():
    rules_ = {}
    df_ = read_csv('ranges.csv', sep=',')
    for _, row in df_.iterrows():
        question = (row['task'], row['question'])
        aux_ = {'type': row['type'], 'low': row['low'], 'high': row['high']}
        rules_[question] = aux_
    return rules_

# cargamos commentarios csv
def get_comment():
    comments_ = {}
    for f in listdir('strings'):
        dict_ = {}
        with open('strings/' + f, 'r', encoding='utf-8') as csvfile:
            # si no es un fichero csv lo saltamos
            if not f.endswith('.csv'):
                continue
            csv_ = csv.reader(csvfile, delimiter=';')
            for row in csv_:
                dict_[row[0]] = row[1]
        comments_[f.split('.')[0]] = dict_
    return comments_
```

Listing 5.11: Método para cargar archivos .csv

El último de los métodos de los que apodamos métodos de lectura tenemos **n_questions_per_task** (ver Listing 5.12), este sencillo método que hemos desarrollado es capaz de crear un diccionario en el que cada tarea tiene asociado su número total de preguntas. Para lograr esta tarea ejecuta una query sobre la tabla QUESTIONS, la cual devuelve la cantidad de ítems que contiene la tarea.

```python
def n_questions_per_task(n_tasks=5, default_lang='esp') -> dict:
    n_tasks_dic = {}
    for i in range(1, n_tasks+1):
        n_questions = db_orm.session.query(Questions).filter(Questions.task==i).filter(
    Questions.language==default_lang).count()
        n_tasks_dic[i] = n_questions
    return n_tasks_dic
```

Listing 5.12: Método para calcular el número de preguntas por tarea

Ahora vamos a tratar de estudiar los métodos que aportan las principales funcionalidades de la máquina de estados, el primero de estos es **check_answer** (ver Listing 5.13), lo que hace este método es comprobar si la respuesta del usuario es válida, para ello estudia en cada caso si se trata de un entero, un texto o una respuesta categórica, hace uso de los ficheros **affirmations.txt**, **negations.txt** y, en especial, **ranges.csv**.

```python
def check_answer(string_, task, question):

    # Afirmacion / negacion
    negations = [el for el in open('strings/negations.txt', 'r').read().split('\n') if
    el]
    affirmations = [el for el in open('strings/affirmations.txt', 'r').read().split('\n'
    ) if el]

    global rules
    if rules is None:
        rules = get_rules()

    if len(string_) > MAX_LENGTH_RESPONSE:
        return False

    ranges = rules[(task, question)]
    # valores numericos
```

```
    if ranges['type'] == 'int':
        try:
            val = int(string_)
            return ranges['low'] <= val <= ranges['high']
        except:
            return False

    # preguntas si/no
    elif ranges['type'] == 'affirmation':
        if string_.lower() in affirmations:
            return True
        elif string_.lower() in negations:
            return True
        else:
            return False

    # no text restrictions
    elif ranges['type'] == 'text':
        return True
```

Listing 5.13: Método para comprobar si el mesnaje del usuario es correcto check_answer

El método **increment_counter** (ver Listing 5.14) recibe como parámetros, la tarea y la pregunta actuales y el id de sesión, devuelve un booleano. Este booleano sirve para indicar si nos encontramos ante la ultima pregunta de todo el cuestionario o no. Entonces, nos encontramos ante tres posibles casos. Caso primero, se ha llegado al final, devolvemos True. Caso segundo, nos encontramos ante la ultima pregunta de una tarea, incrementamos el atributo task y question lo inicializamos a uno; todo esto sobre la tabla STATUS. Finalmente, caso último, incrementamos el contador de preguntas y devolvemos False.

```
def increment_counter(actual_task, actual_question, msession) -> bool:
    global n_questions_per_answer
    global multiple_answer_cache
    d_time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    # TODO 5 deberia ser una constante durante el programa
    if actual_task == 5 and actual_question == n_questions_per_answer[5]:
        # caso base: se ha completado la ultima pregunta
        return True
    elif actual_question == n_questions_per_answer[actual_task]:
        # ultima pregunta de una tarea
        db_orm.session.query(Status).filter(Status.id_session==msession).update({Status.
    task: actual_task+1,

    Status.question: 1})
        return False
    else:
        # simplemente incrementa el contador de question
        db_orm.session.query(Status).filter(Status.id_session==msession).update({Status.
    question: actual_question+1})
        return False
```

Listing 5.14: Método para incrementar el estado increment_counter

El último método de este apartado es **store_correct_answer** (ver Listing 5.15), este cumple la función de guardar la respuesta del usuario de forma correcta, para ello necesita comprobar el caso especial, caso de la respuesta múltiple. Si se trata de este pueden pasar dos cosas, el usuario haya acabado de enviar errores por lo nos encontraríamos que el usuario ha enviado *fin*, *finish* o *fi*, en este caso limpiará la caché con el contador del numero de respuestas múltiples y devolvería True. En el caso contrario lo único que se realiza es incrementar la caché con el contador de respuesta múltiple. Acto seguido almacena la respuesta, esto se hace mediante los métodos de SQLAlchemy, primero creamos un objeto de la clase Responses, después lo añadimos mediante el método **add** y finalmente se guarda el cambio en la base de datos ejecutando sobre la session de la ORM el método **commit**.

```
def store_correct_answer(mtext, mlang, msession, current_question, current_task):
    global multiple_answer_cache
    time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    # Pregunta multiples respuestas
    if current_task <= 3 and current_question == 2:
        global comm
        if mtext.lower() == comm[mlang]['done'].lower():
            # borra la entrada de la cache
            multiple_answer_cache.pop(msession)
            return True
        else:
            # aumenta el contador en cache
            multiple_answer_cache[msession] += 1

    respon = Responses(id_session=msession, question=current_question, task=current_task,
                       message_number=multiple_answer_cache[msession], answer=mtext, Timestamp=time)
    db_orm.session.add(respon)
    db_orm.session.commit()

    return not(current_task <= 3 and current_question == 2)
```

Listing 5.15: Método para almacenar la respuesta store_correct_answer

5.3.2. El método process_message

En esta subsección hemos implementado el método **process_message**, constituye la máquina de estados que procesa la respuesta del usuario, este método hace uso de todos los métodos vistos hasta ahora de forma eficiente. El método podemos verlo completo en el apéndice B.1. Además, podemos ver un resumen del funcionamiento en el diagrama de flujo (ver Figura 5.3).

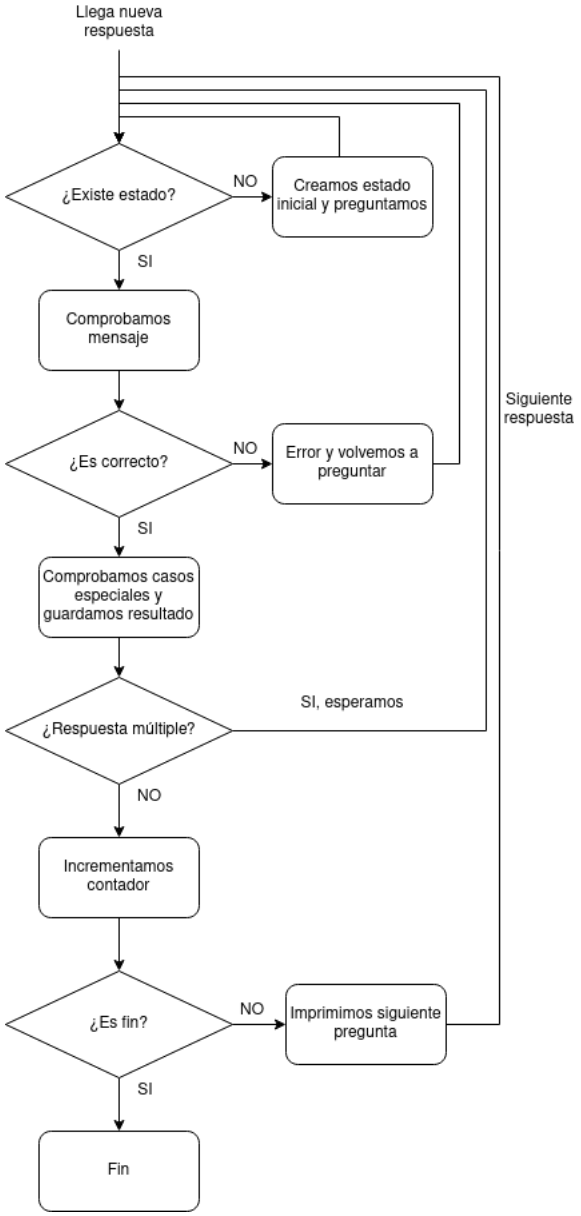

Figura 5.3: Diagrama de flujo de la máquina de estados

El primer paso cuando llega un mensaje es comprobar si los diccionarios **comm** y **n_questions_per_answer** están inicializados, en caso negativo se inicializan.

Las siguientes funciones, que podemos ver en el apéndice (ver Apéndice B.1), son unas funciones auxiliares a las que se accedía mediante palabras clave y que en la versión definitiva han sido eliminadas. No obstante, por el uso que se les ha dado durante toda la fase de prueba cabe mencionarlas. La primera a comentar es la comprobación */help*

o */h*, que devuelve un listado con todas las palabras clave. Como tan solo se implementó la otra función **restart**, la comprobación de ayuda tan solo devolvía "*/restart*" como argumento.

La función **restart** automatizaba el proceso de eliminar las tablas de la base de datos y levantarlas de nuevo cargando las preguntas. Esta función resultaba de mucha utilidad ya que si se efectuaba un cambio sobre la máquina de estados no pasaba nada, sin embargo, cuando el cambio se producía en los modelos de la ORM, estos no se implementaban hasta que no se volvían a crear las tablas. El caso más notable fue cuando tratamos de dar solución al problema de las preguntas de múltiple respuesta en la que se modificó la clase Responses. Además, cada vez que introducíamos cambios y queríamos volver a probar la máquina de estados desde el estado cero tocaba reiniciar las tablas de RESPONSES y STATUS, por lo que esta función también daba solución a este problema.

Con todo esto visto, podemos pasar a comentar la implementación que hemos realizado de la máquina de estados (ver Listing 5.16). El primer estado que nos encontramos es cuando el usuario se ha conectado por primera vez, en este momento no existe estado, es decir, la tabla STATUS no contiene el estado del usuario. Por lo que el estado cero consisten en inicializar esta tabla con la cookie sessionid como identificador de la tabla y devolver la primera tarea junto con su primera pregunta, se lleva a cabo modificando el objeto item. Todo esto se efectúa mediante la primera comprobación en base a una consulta sobre la tabla STATUS, en la que se filtra por id_session.

```
    else:

        # Intenta recuperar la entrada para esa sesion en la tabla status
        query = db_orm.session.query(Status).filter_by(id_session = msession).first()

        # Instanciamos nuevo dato en Status
        if query is None:
            #Pregunta a realizar
            firstquestion = db_orm.session.query(Questions.qtext).filter_by(question =
1, task = 1, language = mlang).first()
            item.comment = comm[mlang]['task1']

            # anyadidos argumentos por defecto y nombre en los argumentos para que el
codigo quede mas claro
            firstSession = Status(id_session=msession, task=1, question=1, language=
mlang)
            db_orm.session.add(firstSession)
            db_orm.session.commit()
            item.text = firstquestion.qtext
            return item
```

Listing 5.16: Estado inicial de la máquina

Cabe recordar que el uso de la cookie como identificador de usuario se debe a que cuando varias personas estén realizando los cuestionarios y traten de acceder al estado, este puede devolver el estado de otro usuario. Por ello resulta de vital importancia tener este identificador como clave primaria con tal de no mostrar otro estado que no sea el que le corresponde a cada usuario.

Ahora bien, en caso de que exista el estado (ver Listing 5.17), la consulta devuelve una fila de la tabla. La siguiente comprobación a efectuar es si el mensaje del usuario es correcto. Caso negativo, no es correcto, tratamos el error, en función de la pregunta y tarea en la que nos encontramos devuelve un error acorde con el estado en el que se

encuentra y repite la pregunta sin alterar el estado. Errores:

- Error en el SUS y el UEQ, devuelve un comentario en el que recuerda al usuario que solo acepta respuestas de tipo numérico y los rangos aceptables.
- Error en el caso de múltiples respuestas, este se da si el usuario excede el límite de caracteres aceptables, por lo que recuerda el límite aceptable para que el usuario reformule el error o lo envíe en varios mensajes diferentes.
- Error en las preguntas con respuestas afirmativas y negativas, el usuario ha contestado otra cosa, Alf le recuerda el tipo de respuestas posibles para esa pregunta.

Si la respuesta ha sido verificada correctamente pasamos a la siguiente comprobación. Aquí tratamos en caso especial, si es el caso de respuesta abierta, la máquina de estados no hace nada más ya que el método **store_correct_answer** realiza las acciones pertinentes descritas en la explicación de este método en la subsección anterior. Si **store_correct_answer** devuelve **True**, nos encontramos ante un caso normal, por lo que proseguimos a la última comprobación.

Esta última comprobación determina si se trata del estado final o un estado intermedio, como esta comprobación se realiza mediante el método **increment_counter**, la tabla STATUS se actualizará al estado correspondiente automaticamente. En el caso de un estado intermedio la única acción que se debe realizar es si se trata de la pregunta uno, ya que habrá que añadir el mensaje de inicial de cada tarea. En el caso de que sea el estado final, **Alf** devolverá un mensaje dando la gracias por haber realizado la tarea y la máquina de estados habrá concluido.

```
1         else:
2             get_task = query.task
3             get_question = query.question
4             correct = check_answer(mtext, get_task, get_question)
5 
6             # OJO si la respuesta es correcta la guardo y lanzo la siguiente
7 
8             if correct:
9 
10                cont_up = store_correct_answer(mtext, mlang, msession, get_question,
    get_task)
11                if cont_up:
12 
13                    is_finished = increment_counter(actual_task=get_task,
    actual_question=get_question, msession =query.id_session)
14                    if is_finished:
15                        item.text = comm[mlang]['end']
16                        return item
17                    else:
18 
19                        actual_status = db_orm.session.query(Status).filter_by(
    id_session = msession).first()
20                        next_question = db_orm.session.query(Questions.qtext).filter_by(
    question = actual_status.question, task = actual_status.task, language = mlang).
    first()
21                        item.text = next_question.qtext
22                        if actual_status.question == 1:
23                            item.comment = comm[mlang][translate_task[actual_status.task
    ]]
24                        #elif actual_status.question == 1:
25                        return item
26                else:
27 
28                    # Cuando se trata de un respuesta con multiples mensajes
```

```
29                      pass
30
31              else:
32
33                  # ha respondido mal, debo indicar porque y repetir la pregunta
34                  if get_task == 4:
35                      item.comment = comm[mlang]["error_num_sus"]
36                  elif get_task == 5:
37                      item.comment = comm[mlang]["error_num_ueq"]
38                  else:
39                      if get_question == 2:
40                          item.comment = comm[mlang]["error_text"]
41                      else:
42                          item.comment = comm[mlang]["error_proposition_cat"]
43
44                  next_question = db_orm.session.query(Questions.qtext).filter_by(question
    = get_question, task = get_task, language = mlang).first()
45                  item.text = next_question.qtext
46                  return item
```

Listing 5.17: Máquina de estados sin comprobación inicial

CAPÍTULO 6
Orquestación y despliegue

6.1 Orquestación

En esta sección vamos a dar una visión global de cómo funciona en conjunto los diferentes módulos, **Alf**. En el capitulo de tareas y cuestionarios hemos definido las preguntas y hemos visto que los archivos se encuentran en el *Backend*, en el capitulo del **Frontend** hemos desarrollado el diseño y su implementación y finalmente en el backend hemos creado la lógica, es decir, todas las funcionalidades para procesar un mensaje. Además se encuentra la base de datos, recordemos que se trata de un contenedor Docker creado a partir de esta imagen de MySQL (ver Apéndice C.1).

Así pues, en el frontend obtenemos la *cookie* **sessionid**, el mensaje del usuario y el idioma, con todo esto generamos un JSON y lo enviamos a la maquina de estados a través del puerto 7777 en el que se encuentra el servidor de *FastAPI*. La API de REST inicializa la base de datos, en caso de no estarlo, y llama al método **process_message** de la máquina de estados, esta comprueba que el mensaje sea correcto, y si lo es, lo procesa. Procesar el mensaje es: guardarlo en la tabla **RESPONSES** de la base de datos, incrementar el contador de la tablas **STATUS** y obtener la siguiente pregunta consultando la tabla **QUESTIONS**. Entonces, cuando la máquina de estados concluye esta devuelve el JSON con el **sessionid**, un comentario, en caso de ser necesario, la pregunta y el idioma. En resumen, solo modifica dos campos del mensaje con formato JSON, el campo text y el comment.

La *cookie* **sessionid**, como hemos mencionado en capítulos anteriores es necesaria para evitar inconsistencias en el sistema. Finalmente, los accesos de lectura y escritura a la base de datos los realizamos a través del **ORM SQLAlchemy** que traducía sentencias *Python* a consultas de lenguaje *SQL*. En el diagrama siguiente (ver Figura 6.1) vemos el resultado final del funcionamiento de **Alf**:

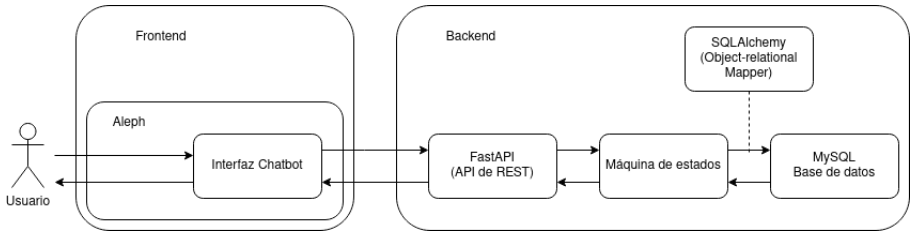

Figura 6.1: Diagrama de flujo de Alf

6.2 Despliegue

Una vez terminado el desarrollo de **Alf** es necesario migrarlo al entorno de producción, es decir, tenemos que añadir la parte del *Frontend* a la última versión de *Aleph*, así pues, la versión de la aplicación estará actualizada con el chatbot integrado. Recordemos que la parte del *Backend* funciona de forma independiente al chatbot, en otras palabras, se trata de un microservicio modular que no va integrado junto al *Frontend*.

El entorno de producción de *Aleph* es un servicio dockerizado, *Docker* es un servicio que permite automatizar el despliegue de aplicaciones, estas se encuentran en contenedores *software*. En consecuencia, hemos creado dos nuevas imágenes de *Docker*, un servicio de *Aleph* con la interfaz de **Alf** y otro servicio con el *Backend* del módulo, estos servicios se encuentran en el archivo **docker-compose.yml** (ver Listing 6.1). Finalmente, tenemos un fichero **Dockerfile** (ver Listing 6.2) que contiene las instrucciones necesarias, para crear la imagen del servicio *Backend*. Una vez ejecutemos estos archivos tenemos un entorno de producción de *Aleph* totalmente desplegado, en el que se encuentra **Alf** integrado.

```
version: '3.1'
services:

  alephdemodb:
    image: mariadb:latest
    container_name: mariadbdev
    command: --default-authentication-plugin=mysql_native_password
    volumes:
      - ${HOME}/docker_volumes/aleph_demo:/var/lib/mysql
    restart: always
    ports:
      - "4409:3306"
    environment:
      - MYSQL_ROOT_PASSWORD=${MYSQL_SERVER_ROOT_PASS}
      - MYSQL_DATABASE=${MYSQL_SERVER_DBNAME}
      - MYSQL_PASSWORD=${MYSQL_SERVER_PASS}
      - MYSQL_USER=${MYSQL_SERVER_USER}

  alephdemo:
    depends_on:
      - alephdemodb
    image: aleph_try
    container_name: alephdemo
    ports:
          - "8000:8000"
    environment:
      - MYSQL_SERVER_DBNAME=${MYSQL_SERVER_DBNAME}
      - MYSQL_SERVER_USER=${MYSQL_SERVER_USER}
      - MYSQL_SERVER_PASS=${MYSQL_SERVER_PASS}
      - MYSQL_SERVER_HOST=alephdemodb
      - MYSQL_SERVER_PORT=${MYSQL_SERVER_PORT}
      - DJANGO_INADVANCE_KEY=${DJANGO_INADVANCE_KEY}
    entrypoint: /entrypoint.sh

  alphchat:
    depends_on:
        - alephdemodb
    build:
      context: .
      dockerfile: Dockerfile
    container_name: alphbot
    ports:
          - "7777:7777"
    environment:
        - MYSQL_SERVER_DBNAME=${MYSQL_SERVER_DBNAME}
        - MYSQL_SERVER_USER=${MYSQL_SERVER_USER}
        - MYSQL_SERVER_PASS=${MYSQL_SERVER_PASS}
        - MYSQL_SERVER_HOST=alephdemodb
        - MYSQL_SERVER_PORT=${MYSQL_SERVER_PORT}
```

```
50          - DJANGO_INADVANCE_KEY=${DJANGO_INADVANCE_KEY}
```

Listing 6.1: Fichero para el despliegue final docker-compose.yml

```
1  FROM python:3.7
2
3  RUN apt-get update -y
4
5  RUN pip install --upgrade pip
6  RUN pip install pandas
7  RUN pip install mysql.connector
8  RUN pip install sqlalchemy
9  RUN pip install pydantic
10 RUN pip install fastapi
11 RUN pip install uvicorn
12 RUN pip install mysqlclient
13
14 ADD Service /opt/backend
15 WORKDIR /opt/backend
16 EXPOSE 7777
17
18 CMD uvicorn main:app --port 7777
```

Listing 6.2: Fichero dockerfile

CAPÍTULO 7
Conclusiones

7.1 Resumen general

Este trabajo ha comprendido por una parte, las fases de creación, diseño, desarrollo, orquestación y despliegue del chatbot y por otra, las fases caracterización del problema, investigación, elaboración de tareas y modificación de los cuestionarios atendiendo a los principios de usabilidad y experiencia de usuario.

Así pues, la última versión de **Alf** es un módulo *chatbot* con personalidad capaz de evaluar la usabilidad y la experiencia de usuario de forma interactiva de la página web de *Aleph*. En cuanto a la interfaz visual. Se han aplicado sobre esta las experiencias y los conocimientos obtenidos, en base al trabajo realizado sobre la usabilidad y la experiencia de usuario, para crear una interfaz altamente usable, interactiva y agradable. En cuanto a la máquina de estados hemos obtenido un resultado eficiente que realiza las tareas y cuestionarios de forma secuencial almacenando los resultados de manera rápida y eficaz. Es menester recalcar que sus tareas y cuestionarios son altamente modificables, por lo que pueden ser adaptados a las necesidades de *Aleph* conforme este proyecto vaya evolucionando, bien alterando la tareas o usando otros cuestionarios.

En resumen, hemos creado un *chatbot* funcional y altamente adaptable con una interfaz eficaz, eficiente, fácil de usar y agradable al usuario. Lo que diferencia a este trabajo de otros *chatbots*, es haber usado esta herramienta para evaluar la usabilidad y la experiencia de usuario de una página web y no conforme con esto, haber aplicado los conocimiento extraídos para desarrollar el propio chatbot.

Finalmente, si realizásemos algunas adaptaciones podríamos incrustar este módulo en cualquier página web en la que quisiésemos evaluar estos atributos, además, al poder modificar los cuestionarios, estos podrían adaptarse a las necesidades y experiencias de cada página web dando pie a un chatbot altamente versátil.

7.2 Posibles mejoras y trabajo futuro

A nivel técnico, la primera mejora que se le podría realizar a fin de lograr una mayor independencia de la aplicación sería desincrustar el código html del fichero **base.html** perteneciente al proyecto *Aleph*. Con esto obtendríamos tres fragmentos de códigos que

se podrían añadir a cualquier página web que quisiese hacer uso de este chatbot.

Otra posible mejora a nivel técnico sería crear algún método o función en las que se calculase, de forma automática, el grado de usabilidad y experiencia de usuario en base a los cuestionarios y a la base de datos, sin embargo, esto solo tendría sentido si no se modificasen los cuestionarios, hacer uso de otros cuestionarios inhabilitara esta función.

Bien, como trabajo futuro hay varias propuestas que no se han llevado a cabo por limitaciones, especialmente de tiempo, aunque por su extensión, bien servirían para un trabajo independiente. Estas tareas comprenderían la validación de las tareas y de las modificaciones de los cuestionarios. La validación se debe hacer según una metodología igual o similar a la empleada en las traducciones de los cuestionarios [8] y [9].

Una vez validadas estas tareas y modificaciones se entraría en el periodo de pruebas, en este habría que hacer un seguimiento de los usuarios, la metodología empleada debería ser similar a la empleada en esta misma fase en [3]. En esta fase ya se evaluaría la usabilidad de *Aleph*, adicionalmente, durante esta fase se deberían realizar las mejoras de esta aplicación en paralelo (hay que entender que esta fase se llevo a cabo durante tres meses). Finalmente, habría que realizar una entrevista con los usuarios encuestados como bien se realiza en el mismo estudio.

Con el objetivo de poder mejorar y ampliar el trabajo, todo el *software* se ha diseñado atendiendo a la idea de conseguir tanta modularidad como fuese posible a fin de poder sustituir o añadir piezas del *software* sin demasiada dificultad y sin que ello afecte al resto de componentes.

7.3 Relación del trabajo desarrollado con los estudios cursados

En este último apartado voy a desarrollar el impacto que han tenido estos años de carrera cómo han confluido en la elaboración de este trabajo, además de añadir también los conocimientos que he adquirido durante el proceso.

Asignaturas como IPC (Interfaces Persona Computador) han influido en la manera de elegir un diseño, conocer qué causas producen un efecto negativo en la experiencia de usuario. Gracias a haber cursado la rama de computación tenía experiencia suficiente en programar con *Python*. En este trabajo también he podido aplicar los conocimientos adquiridos en bases de datos, aunque las consultas no han sido tan complicadas, el conocer el funcionamiento de las bases de datos relacionales ha sido bastante útil. Las prácticas de empresa me aportaron el conocimiento para trabajar con APIs de REST y el uso *Docker* y *Docker-Compose*.

Quizás la parte más complicada o que más me ha costado, debido a la poca experiencia, ha sido la interfaz web, fue lo primero con lo que me encontré cuando empecé este trabajo, y menos *JavaScript*, lenguaje que ya había usado en asignaturas como TSR (Tecnología de Sistemas de Información en la Red) aunque no con el mismo propósito, tanto de *HTML* como de *CSS* tenía poca o nula experiencia. En este apartado he tenido que ser bastante autodidacta, y he hecho uso de todo lo que he tenido a mano para aprender a

usar estos lenguajes.

Vídeos, documentación oficial o páginas web, en especial *W3Schools* [20], han sido, entre otros, elementos de aprendizaje. En conjunto, este trabajo ha resultado en una experiencia positiva, en el que he aprendido nuevas dimensiones de la informática como es el campo del diseño web, he aprendido a hacer uso de herramientas como las API de REST o los ORM. En especial siento que he aprendido a hacer un programa funcional desde cero e integrando diversos lenguajes.

Bibliografía

[1] Vicent Blanes-Selva, Ascensión Doñate-Martínez, Gordon Linklater, Juan M. García-Gómez. Development of quantitative frailty and mortality prediction models on older patients as a palliative care needs assessment tool. medRxiv, 2021. https://doi.org/10.1101/2021.01.22.21249726.

[2] Página actual con la demo de Aleph.
Consultado el 16 de agosto de 2021.
https://demoiapc.upv.es/.

[3] Biduski, Daiana and Bellei, Ericles Andrei and Rodriguez, João Pedro Mazuco and Zaina, Luciana Aparecida Martinez and De Marchi, Ana Carolina Bertoletti, Assessing long-term user experience on a mobile health application through an in-app embedded conversation-based questionnaire. Computers in Human Behavior,2020,vol. 104, p. 106169. https://doi.org/10.1016/j.chb.2019.106169.

[4] ASENSIO-CUESTA, Sabina, et al. How the Wakamola chatbot studied a university community's lifestyle during the COVID-19 confinement. Health Informatics Journal, 2021, vol. 27, no 2, p. 14604582211017944. https://journals.sagepub.com/doi/10.1177/14604582211017944.

[5] Schrepp, Martin and Hinderks, Andreas and Thomaschewski, Jörg, Design and evaluation of a short version of the user experience questionnaire (UEQ-S). International Journal of Interactive Multimedia and Artificial Intelligence, 4 (6), 103-108., 2017. http://dx.doi.org/10.9781/ijimai.2017.09.001.

[6] BROOKE, John, et al. SUS-A quick and dirty usability scale. IUsability evaluation in industry, 1996, vol. 189, no 194, p. 4-7. http://www.tbistafftraining.info/smartphones/documents/b5_during_the_trial_usability_scale_v1_09aug11.pdf.

[7] ASENSIO-CUESTA, Sabina, et al. A user-centered chatbot (Wakamola) to collect linked data in population networks to support studies of overweight and obesity causes: design and pilot study. JMIR Medical Informatics, 2021, vol. 9, no 4, p. e17503. https://medinform.jmir.org/2021/4/e17503.

[8] SEVILLA-GONZALEZ, Magdalena Del Rocio, et al. SEVILLA-GONZALEZ, Magdalena Del Rocio, et al. Spanish Version of the System Usability Scale for the Assessment of Electronic Tools: Development and Validation. JMIR Human Factors, 2020, vol. 7, no 4, p. e21161. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7773510/.

[9] RAUSCHENBERGER, Maria, et al. Efficient measurement of the user experience of interactive products. How to use the user experience questionnaire (UEQ). Example:

Spanish language version. 2013. https://reunir.unir.net/handle/123456789/9661.

[10] Página principal de wakamola.
Consultado el 27 de marzo de 2021.
https://wakamola.webs.upv.es/index.php/publicaciones-cientificas-2/.

[11] Documentación oficial de Django con información sobre el framework.
Consultado el 27 de marzo de 2021.
https://www.djangoproject.com/start/.

[12] Página oficial de FastAPI con tutorial sobre el framework para construir APIs.
Consultado el 15 de julio de 2021.
https://fastapi.tiangolo.com/tutorial/.

[13] Documentación oficial de Django con información sobre el framework.
Consultado el 24 de julio de 2021.
https://docs.sqlalchemy.org/en/14/orm/.

[14] Página con información sobre el cuestionario System Usability Scale.
Consultado el 21 de julio de 2021.
https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html.

[15] Página con información sobre la realización de preguntas.
Consultado el 21 de agosto de 2021.
https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html.

[16] Página oficial sobre la UNE-EN ISO 9241-11:1998.
Consultado el 21 de agosto de 2021.
https://www.une.org/encuentra-tu-norma/busca-tu-norma/norma?c=N0013840.

[17] Documentación oficial sobre MySQL 8.0.
Consultado el 15 de mayo de 2021.
https://dev.mysql.com/doc/refman/8.0/en/.

[18] Documentación oficial sobre Docker-Compose.
Consultado el 16 de mayo de 2021.
https://docs.docker.com/compose/.

[19] Imagen oficial de MySQL en Docker Hub.
Consultado el 16 de mayo de 2021.
https://hub.docker.com/_/mysql.

[20] Página oficial de w3schools con información sobre JavaScript, HTML y CSS.
Consultado el 3 de junio de 2021.
https://www.w3schools.com/.

APÉNDICE A
Código Frontend

A.1 HTML del Chatbot

```
<!DOCTYPE html>
{% load static %}

<html lang="en">
<head>
    <!-- Bootstrap core CSS -->
    <!-- css/bootstrap.min.css -->
    <link href="{% static 'basic/assets/css/bootstrap.min.css'%}" rel="stylesheet">
    <!-- Custom styles for this template -->
    <!-- CHATBOT CSS-->
    <link href="{% static 'basic/assets/css/chat-widget.css'%}" rel="stylesheet">

</head>
<!-- He anyadido este tipo de body para que el footer quede abajo con su clase footer-->
<body class="d-flex flex-column min-vh-100">

    <div class="chatContainer">
        <button id="alfBtn" class="open-chat-button" onclick="openForm()" onmouseover="
    balloonChat(this)" aria-activedescendant=""></button>
        <div id="balloonChatId" class="alftext">
            <img src="/static/basic/images/Balloon2.png" class="balloon">
            <div class="presentation"><p class="presemtationTextSize"></p></div>
        </div>
        <div id="myForm" class="chat-popup">
            <form class="form-chat-container">
                <label class="labelChat"><b>Alf Chat</b></label>
                <div class="alfIcon"> <img src="/static/basic/images/alf4.png" alt="Alf"
     width="30%" style="float: right;"></div>
                <div id="chatBox" class="chatBox">

                    <p id="presentationText" class="botFirstText"></p>

                </div>
                <div>
                    <label class="labelChat"><b>Message</b></label>
                    <input id="textInput" class="input-box" type="text" name="msg"
    placeholder="Type message..">
                </div>
                <div>
                    <button type="button" id="sendBtn" class="btn send" onclick="
    sendButton()">Send</button>
                    <button type="button" class="btn cancel" onclick="closeForm()">Close
    </button>
                </div>
                <div class="selectLanguage">
                    <div id="langSelector" class = langSelector>
                        <button type="button" id="langSelectorEng" class = "
    langSelectorBtn" onclick="changeLangButtonEng()">
                            <img src="/static/basic/images/eng_flag.png" class="
    langImage" width="40px" height="40px" alt="Eng">
                        </button>
```

```
                        <button type="button" id="langSelectorEsp" class = "
    langSelectorBtn" onclick="changeLangButtonEsp()">
                            <img src="/static/basic/images/esp_flag.png" class="
    langImage" width="40px" height="40px" alt="Esp">
                        </button>
                        <button type="button" id="langSelectorVal" class = "
    langSelectorBtn" onclick="changeLangButtonVal()">
                            <img src="/static/basic/images/valencian_flag.png" class="
    langImage" width="40px" height="40px" alt="Val">
                        </button>
                    </div>

                    <button type="button" id="langButton" class="selectLanguageButton"
    onclick="openSelectorButton()">
                        <img src="/static/basic/images/global.png" alt="Lang">
                    </button>
                </div>
            </form>
        </div>
    </div>

    <header>

    </footer>
    <!-- Chat core JavaScript
    ================================================== -->
    <!-- js/chat.js -->
    <script src="{% static 'basic/assets/js/chat.js'%}"></script>
</body>

</html>
```

Listing A.1: Fichero base.html

A.2 CSS del Chatbot

```
@font-face {
  font-family: Maven;
  src: url('/static/basic/assets/fonts/MavenPro-VariableFont_wght.ttf');
}

@font-face {
  font-family: Maven;
  font-weight: bold;
  src: url('/static/basic/assets/fonts/MavenPro-Bold.ttf');
}

@font-face {
  font-family: Maven;
  font-weight: 600;
  src: url('/static/basic/assets/fonts/MavenPro-SemiBold.ttf');

}

.chatContainer{
  z-index: 10;
  position: fixed;
  bottom: 2.5rem;
  right: 6.25rem;

}

.open-chat-button {
  position: absolute;
  bottom: 0;
  right: 0;
  width: 5rem;
  height: 5rem;
  border-radius: 50%;
  background: url(/static/basic/images/alf4.png);
```

```
  background-size: 7.5rem;
  border-style: none;
  border: hidden;
}

.open-chat-button:hover {
  box-shadow: 1rem 0.938rem 1.938rem -0.125rem rgba(48,87,130,0.45);
}

.alftext {
  bottom: 3.75rem;
  right: 5rem;
  width: 25rem;
  visibility: visible;
  text-align: center;
  padding: 0.313rem 0;

  /* Position the tooltip */
  position: absolute;
  z-index: 11;
}

.presentation {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
  width: 25rem;
}

.presemtationTextSize {
  font-size: 1.25rem;
}

.chat-popup {
  display: none;
  width: 25rem;
  background-color: #FFD488;
  /*opacity: 0.9; */
  padding: 1.25rem;
  border-radius: 1.25rem;
}

/*.form-chat-container {} amarillo: #ffb259*/

.chatBox {
  height: 25rem;
  border-radius: 1.25rem;
  overflow-y: scroll;
  overflow-x: hidden;
  word-wrap: break-word;
  background-color: white;
  opacity: 1;
}

#chatBox::-webkit-scrollbar {
  width: 1.25rem;
  border-radius: 1.25rem;
  background-color: #bfbfbf;
}

.input-box {
  border-radius: 0.625rem;
  width: 22.5rem;
}

/*CHAT MESSAGES SECTION*/

.botFirstText {
  font-size: 1.25rem;
  margin-left: 0.938rem;
  margin-top: 0.625rem;
  margin-right: 1.25rem;
  line-height: 1.6rem;
```

```
}

.botText{
  font-size: 1.25rem;
  margin-left: 0.938rem;
  margin-right: 1.25rem;
  line-height: 1.6rem;;
}

.userText{
  font-size: 1.25rem;
  margin-right: 0.938rem;
  margin-left: 1.25rem;
  text-align: right;
  padding-left: 1.563rem;
  line-height: 1.6rem;
}

/*I*/

.alfIcon {
  position: absolute;
  right: 0rem;
  top: 0;
  padding: 0.625rem;
}

/*LANGUAGES SECTION*/

.selectLanguageButton {
  position: absolute;
  bottom: 0.625rem;
  right: 0.625rem;
  width: 2.5rem;
  height: 2.5rem;
  border-radius: 50%;
  border-style: none;
}

.langSelector {
  display: none;
}

.langSelectorBtn {
  width: 2.5rem;
  height: 2.5rem;
  border-radius: 50%;
  border-style: none;
}

.langImage {
  left: -0.25rem;
  position: relative;
  top: -0.063rem;
}

.send:hover, .cancel:hover, .selectLanguageButton:hover {
  opacity: 0.5;
}

/*MEDIA QUERIES*/

/*MEDIA QUERY TABLETS*/
@media (min-width: 481px) and (max-width:768px) {

  .labelChat {
    font-size: 1.8rem;
  }

  .chatContainer {
    right: 3rem;
  }

  .open-chat-button {
```

```
    width: 4.5rem;
    height: 4.5rem;
    background-size: 6.7rem;
  }

  .alftext {
    right: 5rem;
    bottom: 3rem;
    width: auto;
    height: auto;
  }

  .presentation {
    width: 19rem;
    height: 4.5rem;
  }

  .presemtationTextSize {
    font-size: 1rem;
  }

  .balloon {
    width: 20rem;
    height: 6rem;
  }

}

/*MEDIA QUERY MOBILES*/
@media (max-width: 480px) {

  .labelChat {
    font-size: 1.8rem;
  }

  .chatContainer {
    right: 0rem;
  }

  .open-chat-button {
    right: 1rem;
    bottom: -1rem;
    width: 4.5rem;
    height: 4.5rem;
    background-size: 6.7rem;
  }

  .chat-popup { /*?????????????????*/
    max-width: 30rem;
  }

  .alftext {
    right: 5rem;
    bottom: 3rem;
    width: auto;
    height: auto;
  }

  .presentation {
    width: 19rem;
    height: 4.5rem;
  }

  .presemtationTextSize {
    font-size: 1rem;
  }

  .balloon {
    width: 20rem;
    height: 6rem;
  }
}
```

Listing A.2: Fichero chat-widget.css

A.3 JavaScript del Chatbot

```
var lang = "esp";

function openForm() {

  if (document.getElementById("balloonChatId").style.visibility == "visible") {
    document.getElementById("balloonChatId").style.visibility = "hidden";
  }

  document.getElementById("myForm").style.display = "block";
  document.getElementById("alfBtn").style.display = "none";
}

function closeForm() {
  document.getElementById("myForm").style.display = "none";
  document.getElementById("alfBtn").style.display = "block";
}

function firstBotMessage() {
  return null;
}

function getHardResponse(userText) {

  let response = getBotResponse(userText);

  let botComment = response['comment'];
  if (botComment != null) {
    let botComHtml = '<p class="botText">' + botComment + '</p>';
    $("#chatBox").append(botComHtml);
  }

  let botResponse  = response['text'];
  let botHtml='<p class="botText">' + botResponse + '</p>';

  $("#chatBox").append(botHtml);

  updateScroll();

}

function getRespuesta() {

  let userText = $("#textInput").val();
  let userHtml = '<p class="userText">' + userText + '</p>';

  $("#textInput").val("");
  $("#chatBox").append(userHtml);

  updateScroll();

  setTimeout(()=> {
    getHardResponse(userText);
  }, 1000)

}

function sendButton() {
  getRespuesta();
}

$("#textInput").keypress(function(e) {
  if(e.which == 13) {
    e.preventDefault();
    document.getElementById("sendBtn").click();
  }
});

/**  TIEMPO  **/
function getTime() {
  let today = newDate();
  hours = today.getHours();
```

```
72      minutes = today.getMinutes();
73
74      let time = hours + ":" + minutes;
75      return time;
76    }
77
78
79    function updateScroll(){
80      var element = document.getElementById("chatBox");
81      element.scrollTop = element.scrollHeight;
82    }
83
84    /**  Respuesta BOT  **/
85    function getBotResponse(input) {
86
87      var sessionId = getSessionId();
88      console.log(sessionId);
89
90      let answer = { "idSession": sessionId, "text": input, "comment": "", "language":
        lang};
91
92      var xhr = new XMLHttpRequest();
93      var url = "http://127.0.0.1:7777/bot/";
94      xhr.open("POST", url, false);
95      xhr.setRequestHeader("Content-Type", "application/json");
96      var data = JSON.stringify(answer);
97      xhr.send(data);
98      var response = JSON.parse(xhr.responseText);
99      console.log(response);
100     return response;
101   }
102
103   /**  Get cookie sessionid  **/
104   function getSessionId() {
105     var jsId = document.cookie.match(/sessionid=[^;]+/);
106     if(jsId != null) {
107       if (jsId instanceof Array)
108           jsId = jsId[0].substring(10);
109       else
110           jsId = jsId.substring(10);
111     }
112     console.log(jsId);
113     return jsId;
114   }
115
116   /** Hover bocadillo chatbot */
117
118   if (document.getElementById("balloonChatId").style.visibility == "visible") {
119     setTimeout(function(){ document.getElementById("balloonChatId").style.visibility = "
        hidden"; }, 5000);
120   }
121
122   document.getElementById("alfBtn").addEventListener("mouseover", balloonChat);
123
124   function balloonChat() {
125     document.getElementById("balloonChatId").style.visibility = "visible";
126     setTimeout(function(){ document.getElementById("balloonChatId").style.visibility = "
        hidden"; }, 5000);
127   }
128
129   /** Seleccionar Idioma */
130
131   function openSelectorButton() {
132     if(document.getElementById("langSelector").style.display == "block") {
133       document.getElementById("langSelector").style.display = "none";
134     }
135     else {
136       document.getElementById("langSelector").style.display = "block";
137     }
138   }
139
140   function changeLangButtonEng() {
141     lang = "eng";
142     document.getElementById("langSelector").style.display = "none";
```

```
143     document.getElementById("presentationText").innerHTML = "Hello, you're going to
        participate in a usability test. All your answers are correct Thank you for
        participating!";
144   }
145
146   function changeLangButtonEsp() {
147     lang = "esp";
148     document.getElementById("langSelector").style.display = "none";
149     document.getElementById("presentationText").innerHTML = "Hola vas a participar en
        una prueba de usabilidad. Todas tus respuestas son correctas Gracias por participar!
        ";
150   }
151
152   function changeLangButtonVal() {
153     lang = "val";
154     document.getElementById("langSelector").style.display = "none";
155     document.getElementById("presentationText").innerHTML = "Hola, participaras en una
        prova d'usabilitat. Totes les teues respostes son correctes Gracies per participar!"
        ;
156   }
```

Listing A.3: Fichero chat.js

APÉNDICE B
Código Backend

B.1 Maquina de estados/Alf

```
#### MAQUINA DE ESTADOS ####
from sqlalchemy.sql.expression import true
import db_orm
import json
import csv
import time

from models_orm import Status, Questions, Responses
from sqlalchemy import update, delete
from os import listdir
from pandas import read_csv
from datetime import datetime
from collections import defaultdict

global rules
global n_questions_per_answer
global multiple_answer_cache
global comm

rules = None
n_questions_per_answer = None
comm = None
multiple_answer_cache = defaultdict(int)
translate_task = {1:"task1", 2:"task2", 3:"task3", 4:"sus", 5:"ueq"}


# movido aqui como constante para tener que modificarlo solo una vez
MAX_LENGTH_RESPONSE = 280

# Metodo inicializacion #
def create_tables():
    db_orm.Base.metadata.create_all(db_orm.engine)

# Metodos cargar preguntas#
def load_questions():
    for lang in listdir('questions'):
        try:
            for i, task in enumerate(sorted(listdir('questions/' + lang))):
                with open('questions/' + lang + '/' + task, 'r', encoding='utf-8') as
    file:
                    b = 0
                    lines = file.read().split('\n')
                    for j, line in enumerate(lines):
                        if line.strip(): #Elimina espacios al principio y al final del
    string
                            arroz = Questions(j + 1 - b, i + 1, line.strip(), lang)
                            db_orm.session.add(arroz)
                            db_orm.session.commit()
                        else:
                            b += 1
        except:
```

```
            db_orm.session.rollback()
        finally:
            db_orm.session.close()

#cargamos reglas csv
def get_rules():
    rules_ = {}
    df_ = read_csv('ranges.csv', sep=',')
    for _, row in df_.iterrows():
        question = (row['task'], row['question'])
        aux_ = {'type': row['type'], 'low': row['low'], 'high': row['high']}
        rules_[question] = aux_
    return rules_

# cargamos commentarios csv
def get_comment():
    comments_ = {}
    for f in listdir('strings'):
        dict_ = {}
        with open('strings/' + f, 'r', encoding='utf-8') as csvfile:
            # si no es un fichero csv lo saltamos
            if not f.endswith('.csv'):
                continue
            csv_ = csv.reader(csvfile, delimiter=';')
            for row in csv_:
                dict_[row[0]] = row[1]
        comments_[f.split('.')[0]] = dict_
    return comments_

#### Funciones principales ####

#comprobar respuesta de usuario
def check_answer(string_, task, question):

    # Afirmacion / negacion
    negations = [el for el in open('strings/negations.txt', 'r').read().split('\n') if
    el]
    affirmations = [el for el in open('strings/affirmations.txt', 'r').read().split('\n'
    ) if el]

    global rules
    if rules is None:
        rules = get_rules()

    if len(string_) > MAX_LENGTH_RESPONSE:
        return False

    ranges = rules[(task, question)]
    # valores numericos
    if ranges['type'] == 'int':
        try:
            val = int(string_)
            return ranges['low'] <= val <= ranges['high']
        except:
            return False

    # preguntas si/no
    elif ranges['type'] == 'affirmation':
        if string_.lower() in affirmations:
            return True
        elif string_.lower() in negations:
            return True
        else:
            return False

    # no text restrictions
    elif ranges['type'] == 'text':
        return True

def increment_counter(actual_task, actual_question, msession) -> bool:
    global n_questions_per_answer
    global multiple_answer_cache
    d_time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    # TODO 5 deberia ser una constante durante el programa
```

```
122     if actual_task == 5 and actual_question == n_questions_per_answer[5]:
123         # caso base: se ha completado la ultima pregunta
124         return True
125     elif actual_question == n_questions_per_answer[actual_task]:
126         # ultima pregunta de una tarea
127         db_orm.session.query(Status).filter(Status.id_session==msession).update({Status.
    task: actual_task+1,
128 
    Status.question: 1})
129         return False
130     else:
131         # simplemente incrementa el contador de question
132         db_orm.session.query(Status).filter(Status.id_session==msession).update({Status.
    question: actual_question+1})
133         return False
134 
135 # Este metodo solo se llama cuando la respuesta es correcta
136 def store_correct_answer(mtext, mlang, msession, current_question, current_task):
137     global multiple_answer_cache
138     time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
139     # Pregunta multiples respuestas
140     if current_task <= 3 and current_question == 2:
141         global comm
142         if mtext.lower() == comm[mlang]['done'].lower():
143             # borra la entrada de la cache
144             multiple_answer_cache.pop(msession)
145             return True
146         else:
147             # aumenta el contador en cache
148             multiple_answer_cache[msession] += 1
149 
150     respon = Responses(id_session=msession, question=current_question, task=current_task
    ,
151                        message_number=multiple_answer_cache[msession], answer=mtext,
    Timestamp=time)
152     db_orm.session.add(respon)
153     db_orm.session.commit()
154 
155     return not(current_task <= 3 and current_question == 2)
156 
157 
158 # OJO Calcular el numero de praguntas para cada fase para generalizar el codigo
159 def n_questions_per_task(n_tasks=5, default_lang='esp') -> dict:
160     n_tasks_dic = {}
161     for i in range(1, n_tasks+1):
162         n_questions = db_orm.session.query(Questions).filter(Questions.task==i).filter(
    Questions.language==default_lang).count()
163         n_tasks_dic[i] = n_questions
164     return n_tasks_dic
165 
166 
167 
168 #Inicio de la maquina de estados#
169 def process_message(item):
170 
171     msession = item.idSession
172     mtext = item.text
173     mlang = item.language
174 
175     global comm
176     if comm is None:
177         comm = get_comment()
178 
179     global n_questions_per_answer
180     if n_questions_per_answer is None:
181         n_questions_per_answer = n_questions_per_task()
182 
183     # TODO esta parte del codigo debe ser eliminada cuando acaben las pruebas
184     if item.text.lower() == "/restart":
185         #eliminamos las tablas
186         try:
187             Status.__table__.drop(db_orm.engine)
188             Questions.__table__.drop(db_orm.engine)
189             Responses.__table__.drop(db_orm.engine)
```

```
190         except:
191             pass
192 
193         #volvemos a crear las tablas
194         create_tables()
195         #volvemos a cargar las preguntas
196         load_questions()
197         item.text = "Done"
198         return item
199 
200     elif item.text == "/help" or item.text == "/h":
201         item.text = comm[mlang]['help']
202         return item
203 
204     ##################################################
205     ############     PROCESAMOS MENSAJE     ############
206     ##################################################
207     else:
208 
209         # Intenta recuperar la entrada para esa sesion en la tabla status
210         query = db_orm.session.query(Status).filter_by(id_session = msession).first()
211 
212         # Instanciamos nuevo dato en Status
213         if query is None:
214             #Pregunta a realizar
215             firstquestion = db_orm.session.query(Questions.qtext).filter_by(question =
    1, task = 1, language = mlang).first()
216             item.comment = comm[mlang]['task1']
217 
218             # anyadidos argumentos por defecto y nombre en los argumentos para que el
    codigo quede mas claro
219             firstSession = Status(id_session=msession, task=1, question=1, language=
    mlang)
220             db_orm.session.add(firstSession)
221             db_orm.session.commit()
222             item.text = firstquestion.qtext
223             return item
224 
225         # existe una entrada de status en la tabla
226         else:
227             get_task = query.task
228             get_question = query.question
229             correct = check_answer(mtext, get_task, get_question)
230 
231             # OJO si la respuesta es correcta la guardo y lanzo la siguiente
232 
233             if correct:
234 
235                 cont_up = store_correct_answer(mtext, mlang, msession, get_question,
    get_task)
236                 if cont_up:
237 
238                     is_finished = increment_counter(actual_task=get_task,
    actual_question=get_question, msession =query.id_session)
239                     if is_finished:
240                         item.text = comm[mlang]['end']
241                         return item
242                     else:
243 
244                         actual_status = db_orm.session.query(Status).filter_by(
    id_session = msession).first()
245                         next_question = db_orm.session.query(Questions.qtext).filter_by(
    question = actual_status.question, task = actual_status.task, language = mlang).
    first()
246                         item.text = next_question.qtext
247                         if actual_status.question == 1:
248                             item.comment = comm[mlang][translate_task[actual_status.task
    ]]
249                         #elif actual_status.question == 1:
250                         return item
251                 else:
252 
253                     # Cuando se trata de un respuesta con multiples mensajes
254                     pass
```

```
255
256             else:
257
258                 # ha respondido mal, debo indicar porque y repetir la pregunta
259                 if get_task == 4:
260                     item.comment = comm[mlang]["error_num_sus"]
261                 elif get_task == 5:
262                     item.comment = comm[mlang]["error_num_ueq"]
263                 else:
264                     if get_question == 2:
265                         item.comment = comm[mlang]["error_text"]
266                     else:
267                         item.comment = comm[mlang]["error_proposition_cat"]
268
269                 next_question = db_orm.session.query(Questions.qtext).filter_by(question
    = get_question, task = get_task, language = mlang).first()
270                 item.text = next_question.qtext
271                 return item
```

Listing B.1: Clase alf.py

B.2 API de REST

```
1  from typing import Optional
2
3  from sqlalchemy.sql.sqltypes import VARCHAR
4
5  from fastapi import FastAPI
6  from pydantic import BaseModel
7  from fastapi.middleware.cors import CORSMiddleware
8  from alf import process_message, load_questions, create_tables
9  from response_request import ResponseRequest
10
11 app = FastAPI()
12
13 origins = ["*"]
14
15 app.add_middleware(
16     CORSMiddleware,
17     allow_origins=origins,
18     allow_credentials=True,
19     allow_methods=["*"],
20     allow_headers=["*"],
21 )
22
23 create_tables()
24
25 load_questions()
26
27
28
29 @app.post("/bot/")
30 async def process_response(item: ResponseRequest):
31     print(item.text)
32     response = process_message(item)
33     return response
```

Listing B.2: Clase main.py

B.3 Response Request

```
1  from pydantic import BaseModel
2
3  class ResponseRequest(BaseModel):
4      idSession: str
5      text: str
```

```
    comment: str
    language: str
```

Listing B.3: Clase response_request.py

B.4 ORM

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine('mysql://user:example@0.0.0.0:3306/db')

Session = sessionmaker(bind=engine)
session = Session()

Base = declarative_base()
```

Listing B.4: Clase db_orm.py

B.5 Modelos de ORM

```
from sqlalchemy.sql.schema import ForeignKeyConstraint
from sqlalchemy.sql.sqltypes import VARCHAR
import db_orm

from sqlalchemy import Column, Integer, String, Float, VARCHAR, Text, DateTime,
    PrimaryKeyConstraint, UniqueConstraint

#### TABLA STATUS ####
class Status(db_orm.Base):

    __tablename__ = 'STATUS'

    id_session = Column(VARCHAR(32), primary_key=True)
    task = Column(Integer, nullable=False)
    question = Column(Integer, nullable=False)
    completed_SUS = Column(Integer)
    completed_UEQ = Column(Integer)
    completed_task1 = Column(Integer)
    completed_task2 = Column(Integer)
    completed_task3 = Column(Integer)
    language = Column(VARCHAR(3))

    __table_args__ = (
        ForeignKeyConstraint(['question', 'task'], ['Questions.question', 'Questions.
    task'])
    )

    def __init__(self, id_session, task, question, language, completed_SUS=0,
    completed_UEQ=0, completed_task1=0, completed_task2=0, completed_task3=0):
        self.id_session = id_session
        self.task = task
        self.question = question
        self.completed_SUS = completed_SUS
        self.completed_UEQ = completed_UEQ
        self.completed_task1 = completed_task1
        self.completed_task2 = completed_task2
        self.completed_task3 = completed_task3
        self.language = language

    def __repr__(self):
        return f'Status({self.id_session}, {self.task}, {self.question}, {self.
    completed_SUS}, {self.completed_UEQ}, {self.completed_task1}, {self.completed_task2
    }, {self.completed_task3}, {self.language})'

```

```
40      def __str__(self):
41          return self.id_session
42
43  #### TABLA QUESTIONS ####
44  class Questions(db_orm.Base):
45
46      __tablename__ = 'QUESTIONS'
47
48      question = Column(Integer)
49      task = Column(Integer)
50      qtext = Column(Text)
51      language = Column(VARCHAR(3))
52
53      __table_args__ = (
54          PrimaryKeyConstraint('question', 'task','language'),
55          UniqueConstraint('question', 'task', 'language'),
56      )
57
58      def __init__(self, question, task, qtext, language):
59          self.question = question
60          self.task = task
61
62          self.qtext = qtext
63          self.language = language
64
65      def __repr__(self):
66          return f'Questions({self.question}, {self.task}, {self.qtext}, {self.language})'
67
68      def __str__(self):
69          return self.question
70
71  #### TABLA RESPONSES ####
72  class Responses(db_orm.Base):
73
74      __tablename__ = 'RESPONSES'
75
76      id_session = Column(VARCHAR(32))
77      question = Column(Integer)
78      task = Column(Integer)
79      message_number = Column(Integer)
80      answer = Column(String(280))
81      Timestamp = Column(DateTime)
82
83
84      __table_args__ = (
85          PrimaryKeyConstraint('id_session','question', 'task', 'message_number'),
86          ForeignKeyConstraint(['question', 'task'], ['Questions.question', 'Questions.
    task'])
87      )
88
89      def __init__(self, id_session, question, task, message_number, answer, Timestamp):
90          self.id_session = id_session
91          self.question = question
92          self.task = task
93          self.message_number = message_number
94          self.answer = answer
95          self.Timestamp = Timestamp
96
97      def __repr__(self):
98          return f'Responses({self.id_session}, {self.question}, {self.task}, {self.answer
    }, {self.Timestamp})'
99
100     def __str__(self):
101         return self.id_session
```

Listing B.5: Clase models_orm.py

APÉNDICE C
Código Docker

C.1 Docker Compose

```
# Use root/example as user/password credentials
version: '3.3'

services:

  db:
    image: mysql:8.0.26
    restart: always
    environment:
      MYSQL_DATABASE: 'db'
      MYSQL_PASSWORD: 'example'
      MYSQL_USER: 'user'
      MYSQL_ROOT_PASSWORD: 'example'
    ports:
      - '3306:3306'
    expose:
      - '3306'
    volumes:
      - my-db:/var/lib/mysql

volumes:
  my-db:
```

Listing C.1: Fichero para el despliegue de la base de datos docker-compose.yml

C.2 Dockerfile

```
FROM python:3.7

RUN apt-get update -y

RUN pip install --upgrade pip
RUN pip install pandas
RUN pip install mysql.connector
RUN pip install sqlalchemy
RUN pip install pydantic
RUN pip install fastapi
RUN pip install uvicorn
RUN pip install mysqlclient

ADD Service /opt/backend
WORKDIR /opt/backend
EXPOSE 7777

CMD uvicorn main:app --port 7777
```

Listing C.2: Fichero dockerfile

C.3 Docker Compose

```
version: '3.1'
services:

  alephdemodb:
    image: mariadb:latest
    container_name: mariadbdev
    command: --default-authentication-plugin=mysql_native_password
    volumes:
      - ${HOME}/docker_volumes/aleph_demo:/var/lib/mysql
    restart: always
    ports:
      - "4409:3306"
    environment:
      - MYSQL_ROOT_PASSWORD=${MYSQL_SERVER_ROOT_PASS}
      - MYSQL_DATABASE=${MYSQL_SERVER_DBNAME}
      - MYSQL_PASSWORD=${MYSQL_SERVER_PASS}
      - MYSQL_USER=${MYSQL_SERVER_USER}

  alephdemo:
    depends_on:
      - alephdemodb
    image: aleph_try
    container_name: alephdemo
    ports:
          - "8000:8000"
    environment:
      - MYSQL_SERVER_DBNAME=${MYSQL_SERVER_DBNAME}
      - MYSQL_SERVER_USER=${MYSQL_SERVER_USER}
      - MYSQL_SERVER_PASS=${MYSQL_SERVER_PASS}
      - MYSQL_SERVER_HOST=alephdemodb
      - MYSQL_SERVER_PORT=${MYSQL_SERVER_PORT}
      - DJANGO_INADVANCE_KEY=${DJANGO_INADVANCE_KEY}
    entrypoint: /entrypoint.sh

  alphchat:
    depends_on:
        - alephdemodb
    build:
      context: .
      dockerfile: Dockerfile
    container_name: alphbot
    ports:
          - "7777:7777"
    environment:
        - MYSQL_SERVER_DBNAME=${MYSQL_SERVER_DBNAME}
        - MYSQL_SERVER_USER=${MYSQL_SERVER_USER}
        - MYSQL_SERVER_PASS=${MYSQL_SERVER_PASS}
        - MYSQL_SERVER_HOST=alephdemodb
        - MYSQL_SERVER_PORT=${MYSQL_SERVER_PORT}
        - DJANGO_INADVANCE_KEY=${DJANGO_INADVANCE_KEY}
```

Listing C.3: Fichero para el despliegue final docker-compose.yml