Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Implementación de una Herramienta para Hacer Búsquedas Usando Lógica Difusa en Bases de Datos

Autor: Rubén García-Patos Benito
Tutor: Ángel Herranz Nieva
Cotutor: Susana Muñoz Hernández

Madrid, Enero 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Implementación de una Herramienta para Hacer Búsquedas Usando Lógica Difusa en Bases de Datos

Enero 2025

Autor: Rubén García-Patos Benito
Tutor: Ángel Herranz Nieva
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Cotutor: Susana Muñoz Hernández
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

Este trabajo fin de grado se centra en la implementación de una herramienta que haga búsquedas con lógica difusa en una base de datos. Esta aplicación ya fue implementada y este trabajo hace una migración a un entorno más moderno.

Buscar, aprender y aprovechar la información, es una tarea que se realiza desde que nuestra especie existe. Se busca el conocimiento en la naturaleza, en las personas, en los libros y, hoy en día de forma ágil, a través de las nuevas tecnologías. Pero muchas veces esta información no es fácil de encontrar o interpretar.

El objetivo de este proyecto es permitir encontrar información utilizando un lenguaje natural, sin requerir aprendizaje previo, solo la comprensión de los conceptos de un lenguaje natural y la capacidad de utilizar una aplicación web.

Nuestros criterios a la hora de buscar información en el entorno que nos rodea no son estrictos. Normalmente no búscamos información repetida, o un objeto especifico y concreto, sino características ambiguas que se ajusten a un patrón. Siguiendo esta forma de búsqueda se emplea un razonador sobre lógica difusa.

Este es el componente clave del trabajo. En torno a él giran la librería y la aplicación web a desarrollar. Buscan aprovechar las ventajas de estas búsquedas desde entornos más amigables.

Abstract

This final project focuses on the development of a tool that performs fuzzy logic searches within a database. The project introduces innovative and powerful tools and environments.

Searching for, learning from, and utilizing information has been a fundamental task since the dawn of our species. Knowledge is sought in nature, from people, in books, and today, very easily, through new technologies. However, this information is often difficult to find or interpret.

The goal of this project is to enable information retrieval using natural, everyday language, without requiring prior learning—just an understanding of the concepts of a natural language and the ability to use a web application. This is the core idea behind the work.

Our criteria when searching for information in our surroundings are not strict. We generally do not look for repeated information or a specific, concrete object, but rather for ambiguous characteristics that fit a pattern. This search approach relies on a reasoning system based on fuzzy logic.

This is the key component of the work. Around it revolve the library and the web application to be developed. Their goal is to leverage the advantages of such searches from more user-friendly environments.

Tabla de contenidos

1. Introducción **1**

1.1. Antecedentes del trabajo . . . 1

1.2. Motivación del trabajo . . . 2

1.3. Objetivos del trabajo . . . 2

1.4. Estructura del documento . . . 3

2. Preliminares **5**

2.1. Lógica difusa . . . 5

2.2. Prolog . . . 6

2.2.1. Lenguaje Prolog . . . 6

2.2.2. Bases de datos Prolog . . . 7

2.2.3. RFuzzy y otras librerías . . . 8

2.3. Lenguajes funcionales . . . 9

2.3.1. Elixir . . . 9

2.3.2. Phoenix . . . 10

3. Análisis **11**

3.1. Conexión Prolog-Elixir . . . 11

3.1.1. La consola de Ciao Prolog . . . 11

3.1.2. Principales mandatos y predicados . . . 14

3.1.3. Las salidas de Ciao Prolog . . . 15

3.2. UfleSe . . . 19

3.2.1. Estructura IU . . . 19

3.2.2. Funcionalidades . . . 23

4. Diseño **29**

4.1. Prolixir . . . 30

4.1.1. Funciones de interacción con la consola . . . 30

4.1.2. Parser . . . 31

4.2. UfleSe . . . 32

4.2.1. *Backend* . . . 32

4.2.2. *Frontend* . . . 34

5. Implementación **37**

5.1. Uso de *GenServer* para la conexión . . . 37

5.2. UfleSe con Phoenix . . . 40

5.2.1. Controladores 40
5.2.2. Enrutamiento 40
5.2.3. UI 41

6. Conclusiones y trabajo futuro **47**
6.1. Resultados 47
6.1.1. Objetivos técnicos no resueltos 48
6.2. Nuevos objetivos 48
6.3. Conclusión final 49

Bibliografía **51**

Anexos **55**

A. Conceptos de Prolog **55**

B. Gramatica de salida Ciao Prolog **57**

Capítulo 1

Introducción

En este capítulo vamos a ver que precede a este trabajo, cual es su motivación y que objetivos persigue. Exite un punto adicional que describe la estructura del trabajo y ayuda a leerlo.

1.1. Antecedentes del trabajo

UfleSe, *User-Friendly Parametric Framework for Expressive Flexible Searches*, es una herramienta ya implementada anteriormente. Fue un trabajo llevado a cabo por Mohammad Halim Deedar y Susana Muñoz-Hernández.[1] Por tanto, la creación de esta herramienta tiene precedentes. UfleSe ya posee una implementación que se quiere imitar y mejorar.

Dicha herramienta es un marco paramétrico para hacer consultas estricas o difusas sobre una base de datos desde una aplicación web. Sin ser técnicos, la aplicación se compone de las siguientes funcionalidades:

1. Inicar sesión en la apliación para guardar datos propios de consultas y bases de datos con información relevante para el usuario.
2. Subir datos de multiples maneras. En formato json, csv y fichero prolog.
3. Hacer búsquedas estrictas o con lógica difusa.
4. Crear relaciones de similitud. Personalizar parámetros difusos.
5. Crear nuevas reglas y relaciones de manera sencilla.
6. Crear consultas y obtener un índice(no necesariamente debe saber ser interpretado por el usuario) sobre la cercanía de su consulta borrosa a la base de datos.

Todo ello lo debe poder hacer un usuario sin conocimiento alguno. Simplemente debe ser capaz de interactuar con la herramienta desde un buscador web y conocer términos del lenguaje natural como caro, calor, temprano, lejos, etc.

1.2. Motivación del trabajo

En nuestro día a día vivimos constantemente tomando decisiones: que camino tomar para llegar al trabajo, que restaurante elegir para una comida, que máster estudiar... Para tomar estas decisiones nos basamos en nuestra experiencia o conocimiento.

Con las nuevas tecnologías, los datos han crecido y por tanto el conocimineto. Saber gestionarlo es fundamental para tomar decisiones más acertadas. Para las grandes empresas o personas con un conocimiento a priori, esta toma de decisiones se puede hacer con herramientas potentes pero más complejas. Este proyecto intenta simplificar la búsqueda de datos a partir de una herramienta que utiliza un lenguaje natural.

Estos motivos fueron los impulsores de la anterior versión de UfleSe, y por tanto, los que también impactan en este trabajo. Pero, además, a ese trabajo se le añaden algunos fallos de implementación y el uso de tecnologías más antiguas y poco adecuadas para el desarrollo de este tipo de aplicación web.

Por tanto, nos motiva el mismo objetivo que el UfleSe primitivo y se recrea para su corrección y mejora.

1.3. Objetivos del trabajo

Por lo descrito anteriormente se busca alcanzar los siguientes objetivos académicos:

- Objetivo 1: Estudiar a fondo Prolog y las bibliotecas que implementan la lógica difusa. En concreto, Rfuzzy y Fuzzy Prolog.
- Objetivo 2: Dominar el lenguaje funcional Elixir y conocer las mejores herramientas para la implementación de UfleSe. Conocer algunos de sus frameworks como Phoenix.
- Objetivo 3: Crear una librería en Elixir que conecte Elixir con una consola de Prolog.
- Objetivo 4: Analizar los requisitos y diseñar una arquitectura software que soporte UfleSe. Este punto incluye discutir las funcionalidades exactas de la aplicación.
- Objetivo 5: Implementar de forma correcta con métodos y herramientas de desarrollo, como git, asdf, git flow, ect.
- Objetivo 6: Realizar una documentación de usuario y desarrollador competente.
- Objetivo 7: Realizar una documentación del trabajo de fin de grado más profesional usando Latex.
- Objetivo 8: Poner el práctica los conocimientos y aptitudes adquiridos a lo largo de una carrera y demostrar la capacidad de aprendizaje.

Con estos objetivos el alumno pretende dominar herramientas que no son tan usuales o están en vías de crecimiento. Las novedades del trabajo como el uso de una base de datos en Prolog, los conceptos de lógica difusa, el uso de un lenguaje funcional con cada vez más popularidad y reconcimiento como Elixir y el desarrollo en un framework que quiere llegar a la altura de otros como React de JavaScript o Ruby on Rails, motivan al alumno a completar y cumplir los objetivos que se detallan. Es evidente, que la suma de los objetivos da lugar al ya mencionado objetivo 8.

Haciendo énfasis en los objetivos principales, tenemos dos grandes partes en este trabajo a las que se hará referencia constantemente:

1. Librería conexión Prolixir-Elixir. *Descrito en el capítulo 4.1*
2. Nuevo UfleSe. *Descrito en el capítulo 4.2*

1.4. Estructura del documento

Para la lectura del trabajo se especifican en esta sección los principales puntos de cada parte y apuntes para facilitar su comprensión.

Para entender bien todos los conceptos es conveniente tener las siguientes aptitudes o conocimientos:

- Conocimientos básicos de matemáticas.
- Conocimientos fundamentales de programación.
- Saber usar minimamente una terminal de sistema operativo.
- Haber tenido contacto con una consola de Prolog.

En primer lugar, hay que ser conscientes de que detrás del TFG y sus objetivos hay dos herramientas que han sido desarrolladas y sobre los que esta memoria va a tratar. Las secciones irán marcadas por el diseño y desarrollo de los dos componentes. El primero trata de la conexión entre Elixir y Prolog. Recibe el nombre de *Prolixir*. A partir de este momento cuando se hable de Prolixir, nos referiremos a esta librería de conexión. Por otro lado, tenemos a *UfleSe*. Nos referiremos a esta herramienta como tal.

Para comenzar con la lectura, es conveniente que el lector refresque, si es necesario, algunos conceptos o deficiones de Prolog. En el anexo A existen definiones básicas para ayudar al seguimiento.

Seguidamente podrá continuar con la lectura de forma natural, teniendo en cuenta el contenido de cada capítulo.

- En el Capítulo 2 se habla del contexto científico de conceptos que se han requerido en la realización del trabajo o herramientas que se han empleado. Es recomendable que el lector revise las secciones cuyo tema no conozca.
- En el Capítulo 3 se analizan la consola de Ciao y la antigua aplicación web de UfleSe. Este análisis es fundamento para plantear el diseño de Prolixir

y UfleSe.

- En el Capítulo 4 se estudia y planifica de forma general como deben ser los componentes de Prolixir y UfleSe.
- En la Implementación, el Capítulo 5, se muestra que herramientas se han escogido para la implementación de los requisitos. Se muestran la finalización de algunos componentes de la aplicación web.
- Los objetivos alcanzados, los objetivos no implementados y el trabajo futuro se detallan en el Capítulo 6.

Capítulo 2

Preliminares

En este capítulo vamos a presentar elementos importantes en la elaboración y entendimiento del trabajo. Algunos puntos son conceptos teóricos que sostienen resultados, tanto de este trabajo, como de herramientas involucradas. Otros puntos son información acerca de lenguajes, herramientas y su contexto. Todo ello da una idea general de en que ambiente se va a desarrollar el trabajo.

2.1. Lógica difusa

En el contexto de la lógica, asociado a una proposición o predicado existe un valor de verdad. Este valor de verdad, en lógica proposicional se evalúa como verdadero o falso, true o false, 0 ó 1...

$$ev : P \rightarrow \{verdadero, falso\} \tag{2.1}$$

En cambio en el día a día, utilizamos constantemente términos que no se pueden computar de manera correcta como verdadero o falso, o incluso que cada persona puede otorgar un resultado distinto. La proposición "Juan es bajo" toma un valor verdadero de forma evidente si mide 1 metro y 20 centímetros. En cambio, si mide 1.70 metros no está del todo claro.

Por ello, dentro de la lógica difusa o borrosa la evaluación de una proposición comprende un valor del 0 al 1, e incluso, en diferentes ámbitos del -1 al 1.

$$ev : P \rightarrow [0, 1] \tag{2.2}$$

Para este trabajo tomaremoms como valor de verdad en que comprende del 0 al 1.

En la lógica proposicional, defeníamos operaciones o aplicaciones n-arias entre proposiciones. Las más típicas son la negación, conjunción, la disyunción, la condición, la doble condición, el xor, ect. Este tipo de aplicaciones son las que dan a la Lógica teórica valor para ser estuadiada. En la ampliación de la lógica difusa, este tipo de aplicaciones se abstraen y siguen devolviendo valores borrosos o difusos.

$$f : [0, 1]^n \rightarrow [0, 1] \tag{2.3}$$

Ahora podemos traducir a un ámbito más riguroso matemáticamente, las proposiciones compuestas como “el restaurante es caro y grande”, atribuyendo a cada proposición un valor y aplicando una aplicación binaria hacia un nuevo valor borroso de verdad. El ejemplo resulta: A la proposición “el restaurante es caro” se la evalúa $0{,}8$ y a “el restaurante es grande” con $0{,}6$. La conjunción se podría definir como el valor mínimo quedando entonces la proposición compuesta “el restaurante es caro y grande” con valor $0{,}6$.

Para la caracterización del valor de verdad de una proposición se da una función que parte del dominio propio de la proposición al intervalo [0,1].Por ejemplo, a la proposición “hace buen tiempo” se le atribuye una función del tipo:

$$f : (Temperatura)\mathbb{R} \to [0, 1] \tag{2.4}$$

Esta función de verdad puede variar según la persona. Y un conjunto de personas pueden tener una similitud en su función debido a la cultura o cercanía. Esto infiere conjuntos de conjuntos de personas con diferentes funciones de verdad. Dentro del objetivo del este trabajo no está en estudio minucioso de esto. Por ello, como más adelante se verá, nos centraremos en la función de verdad de un usuario en concreto y esta será personal.

2.2. Prolog

La programación nació con el objetivo de interactuar y dominar una máquina tan potente como era el computador. A lo largo de la historia han nacido diferentes lenguajes de programación a partir de las formas de pensar de los programadores y las necesidades que se demandaban.

Las diferencias entre los lenguajes formaron diferentes formas de trabajo y surgieron los paradigmas de programación. Entre los paradigmas principales encontramos el paradigma imperativo, centrado en la secuencia de instrucciones y basado en la modificación de estados (como en C o Python); el paradigma funcional, que se basa en la aplicación de funciones y la inmutabilidad de datos (como Elixir y Haskell); y el paradigma orientado a objetos, enfocado en la organización de datos y comportamiento en *objetos* que representan entidades del mundo real o de un sistema (como en Java o Ruby).

Un paradigma específico, pero muy potente, es el conocido como lógico. Este se basa en la lógica matemática para resolver problemas. Prolog, desarrollado en la década de 1970 por Alain Colmerauer y Robert Kowalski, es uno de los lenguajes más representativos de este paradigma.

2.2.1. Lenguaje Prolog

Prolog es un lenguaje dentro del paragdigma de la progrmación lógica. [2] Fue desarrollado en la década de 1970 por Alain Colmerauer y Robert Kowalski[2] con el objetivo de crear un lenguaje que no dijese como hacer una cosa, sino, como es. Buscar su esencia o definición.

Prolog, y también su paradigma, es comúnmente visto como más complejo que los demás y no ha tenido una buena acogida en el ámbito comercial. Además, es complejo escribir programas eficientes y no tiene un gestor de módulos grandes que facilite desarrollar proyectos grandes de software. A pesar de ello, si es conocido y ha abierto grandes caminos hacía la investigación de los lenguajes de programación.

Las principales aplicaciones de Prolog son :

- Procesamiento del lenguaje natural: Con un pequños análisis del mundo que nos rodea, se pueden encontrar patrones lógicos fáciles de implementar y explotar en Prolog.
- Problemas de carácter matemático teórico: Su estructura lógica puede facilitar el entendimiento y análisis de ciertos conceptos o desarrollos matemáticos.
- Bases de datos: Se puede utilizar para almacenar datos y estructurarlos de forma lógica. Las consultas pueden ser complejas y estar unidas logicamente por restricciones o predicados.

Implementaciones de Prolog

Existen diversas implementaciones de Prolog. A lo largo de los años desde su creación se fueron implementando diferentes Prolog que diferían en la sintaxis. Fue en 1990 cuando se creo en estándar ISO, llamado ISO-Prolog. Cada una de estas implementaciones tiene sus propias fortalezas y se adapta mejor a diferentes tipos de problemas y necesidades de programación, pero todas intentan seguir el estándar.

Algunos de los dialectos más conocidos son:

1. Ciao Prolog
2. SWI-Prolog
3. GNU Prolog
4. YAP Prolog

En este trabajo nos centraremos en Ciao Prolog. La elección se debe a que en la Universidad Politécnica de Madrid se imparte docencia relacionada con este dialecto y además, las librerías implementadas son algunas de las herramientas que vamos a utlizar (como RFuzzy[3]).

2.2.2. Bases de datos Prolog

Para este trabajo nos interesa especialmente la novedad que conlleva una base de datos prolog y usar la lógica y las librerías de prolog para sacar resultados más útiles y enriquecidos que simplemente una base de datos usual.

Prolog es una excelente herramienta para crear y manejar bases de datos debido a su capacidad para representar y manipular datos de manera lógica y declara-

tiva. Esto hace que sea especialmente útil para bases de datos complejas donde no solo es importante almacenar datos, sino también establecer relaciones entre ellos y realizar inferencias automáticas. Además, Prolog nos puede ayudar a inferir datos ausentes y trabajar con ambiguedades. Nos permite almacenar conocimiento. Para este proyecto nos interesa y aporta las siguientes facilidades:

- Representación de hechos y relaciones: Se pueden almacenar diferentes tipos de datos al igual que en una base de datos usual. Estos tipos de datos o clases, tendrán unos atributos definidos. Se podrán relacionar de forma lógica. Esto presenta una novedad, ya que en bases de datos usuales se necesitan más tablas o columnas para guardar relaciones. Esto garantiza el esquema general que entendemos por base de datos.
- Consultas: La sencillez de las consultas Prolog nos dan flexiblidad y la funcionalidad mínima de una base de datos. Pero incluso, nos ofrece la posibilidad de hacer consutas más complejas, de forma sencilla y con lenguaje natural gracias a predicados y reglas lógicas.
- Backtracking: El tipo de búsqueda implementado por Prolog nos permite encontrar todas las soluciones de una consuta o solo las que nos sean necesarias. Su tipo de búsqueda nos ayuda en problemas orientados a la optimización o búsqueda exhaustiva.
- Incertidumbre y conocimiento: Las propias funcionalidades de Prolog al no ser solo una base de datos puede atribuir nuevos rasgos a los datos que almacenamos. En el trabajo que nos implica, es fundamental para encontrar datos ajustados a las peticiones difusas que se hagan.

En definitiva, este tipo de bases de datos son utiles cuando no solo nos interesa guardar datos sino también conocimientos.

2.2.3. RFuzzy y otras librerías

La librería RFuzzy de Prolog es un herramienta que mejora la representación lógica difusa en Prolog. [3] Sus principales características se describen a continuación:

1. Sintaxis sencilla para gestionar información ambigua o difusa: Esta librería está especializada en la lógica difusa, por lo que trae consigo una sintaxis sencilla y una implementación que no requiere más esfuerzos en el programador por desarrollar lógica difusa.
2. Números Reales: La librería da soporte para utilizar números reales de forma eficiente.
3. Combinación de consultas crips o difusas: La librería permite hacer consultas sobre hechos trabajando con verdades asociadas a una relación matemática, como por ejemplo *"temperatura >5 ºC"* o con términos difusos, como por ejemplo *"no hace frio"*.
4. Lógica multi-adjunta: Permite gestionar valores de verdad con operaciones

lógicas básicas como negación, conjunción, disyunción,etc. Y hacer operaciónes puramente aritméticas como suma, resta, producto, promedio, etc.

5. Permite utilizar valores predeterminados: En caso de no tener un dato se puede seguir haciendo un razonamiento.
6. Tipos: Permite distinguir los dominios dentro de una proposición que será evaluada con un valor difuso. Los predicados así, se podrán ajustar a deificiones más robustas.

2.3. Lenguajes funcionales

En este trabajo vamos a desarrollar herramientas con Elixir. Es fundamental comprender que este lenguaje pertece a otro paradigma. Es esta sección veremos que características tiene y en que nos puede ayudar.

Los lenguajes de programación funcionales son un paradigma que se centra en el uso de funciones como unidades básicas de organización y manipulación de datos. A diferencia de los lenguajes imperativos, donde se especifican pasos secuenciales para modificar el estado de un programa, los lenguajes funcionales buscan describir qué debe hacerse en lugar de como hacerlo, enfatizando el uso de funciones puras y eliminando posibles efectos secundarios sobre otras partes de un programa.

Las principales carácterísticas son:

1. Declarativo: Pretende mostrar qué debe hacerse en vez de cómo hacerlo.
2. Abstracción de funciones: Las funciones son una unidad mínima y pueden ser tratadas incluso como variables. Las funciones son fácilmente reutilizables.
3. Inmutabilidad: Los datos inmutables garantizan que el estado de un programa no cambie inesperadamente. Esto reduce errores y problemas temporales.
4. Funciones de orden superior y expresiones lambda.

La programación funcional es utilizada en sistemas que requieren alta concurrencia, como servidores de telecomunicaciones (Erlang) y procesamiento de grandes volúmenes de datos (Scala). Su enfoque en funciones puras y la inmutabilidad mejora la confiabilidad y seguridad del software, facilita el paralelismo, y reduce la probabilidad de errores. Los más conocidos son algunos como Haskell, Erlang, Scala y Elixir.

2.3.1. Elixir

Elixir es un lenguaje de programación [4] funcional que ha ganado popularidad en los últimos años, especialmente en sistemas de alta concurrencia, sistemas distribuidos y desarrollo de aplicaciones web.

Las principales carácterísticas de Elixir son:

- Escalabilidad: La ejecución se lleva a cabo en multiples hilos llamados procesos. Estos son muy ligeros y están implementados para mantener una buena comunicación. El lenguaje es muy apto para la alta concurrencia.
- Gestión de errores: Ante las circunstancias de la producción de un programa, Elixir es capaz de volver a un estado inicial para solucionar fallos.
- Adaptable y extensible a dominios: Elixir ha sido diseñado para ser ajustado a las necesidades de un dominio concreto.
- Paradigma funcional: Contiene las principales carácterísticas de los lenguajes funcionales.

2.3.2. Phoenix

En los grandes proyectos un desarrollador debe organizar correctamente su trabajo. Realizar un pequeño programa puede ser sencillo sin ayuda de herramientas de desarrollo, pero cuando el producto crece debemos tener una estructura en el proyecto. Para trabajar con Elixir tenemos la ventaja de desarrollar con Phoenix. Para la implementación de UfleSe se ha utilizado. (Capítulo 5.2)

Phoenix es un *framework web* para Elixir,[5] diseñado para construir aplicaciones web rápidas, escalables y mantenibles. Se inspira en otros frameworks modernos como Ruby on Rails, pero se aprovecha de las ventajas del lenguaje Elixir.

Las principales características de Phoenix:

- Alta concurrencia: Phoenix puede manejar miles de conexiones concurrentes sin perder rendimiento.
- Escalabilidad: Elixir y Phoenix están diseñados para distribuirse fácilmente en múltiples servidores, lo que permite escalar aplicaciones de manera eficiente.
- Alta fiabilidad: Al estar basado en la máquina virtual de Erlang, Phoenix hereda su capacidad de manejar fallos y mantener la aplicación funcionando sin interrupciones.
- Soporte para *websockets*: Permite construir aplicaciones en tiempo real que requieren comunicación bidireccional entre el servidor y el cliente.
- *Framework full-stack*: Phoenix proporciona herramientas tanto para la parte del servidor como para la interfaz de usuario, utilizando tecnologías como LiveView para crear interfaces interactivas sin JavaScript.

Capítulo 3

Análisis

En este capítulo se estudia a fondo las características que tienen las herramientas con las que vamos a tener que trabajar. En primer lugar se hace un análisis de los aspectos más significativos de la consola de Ciao Prolog y sus salidas. Por otro lado, se analiza el aspecto de la aplicación web de UfleSe y sus funcionalidades.

3.1. Conexión Prolog-Elixir

La idea de esta librería es aprovecharse de los beneficios de Prolog. Para ello se desea que desde Elixir, se pueda interartuar con una consola de Ciao Prolog como si se estuviese en la misma.

3.1.1. La consola de Ciao Prolog

Para entender el comportamiento y las interacciones que nuestra librería debe gestionar, es importante comprender las propiedades de la consola de Ciao Prolog. A continuación, se describen las características, componentes y estados de esta consola.

Componentes

La consola de Ciao Prolog es una consola sencilla, similar a la de un sistema operativo, con una entrada de texto y una salida en pantalla(comunmente conocida como REPL), que lee una instrucción, la ejecuta, escribe la salida y vuelve al bucle. A continuación, se detallan los componentes principales de esta consola:

- *Prompt*: Es el símbolo que aparece cuando la consola espera una nueva consulta. Normalmente es "`?- `".
- Consulta o mandato: Es la entrada que introduce el usuario. Va a continuación del *prompt*.
- Salida: Es la respuesta (si existe y es computable) que devuelve el sistema a una consulta o entrada. Puede ser un error o una respuesta.

- Error: Es un tipo de salida que indica al usuario un error. Es una cadena de texto.
- Respuesta: La respuesta es la salida cuando no hay error. Puede ser una solución o el valor "*yes*" o "*no*" para indicar si la consulta fue exitosa o no.
- Solución: Es un conjunto de restricciones en las que hay una variable, una relación y un término.
- Variable: Es un nombre al que se le puede relacionar un valor. Empiezan por mayúscula.
- Relación: Es un operador que establece una relación entre dos elementos. Ejemplos son `=`, `.>.`, `.<.` entre otros.
- Término: Puede ser un átomo, lista o estructura.

Estados

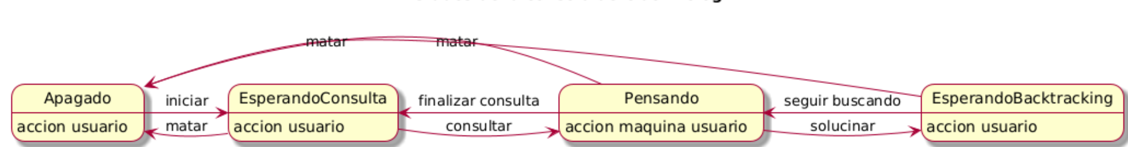
Es fundamental conocer en que estado se encuentra una consola y que tipos hay. Depende del estado van a existir distintos tipos de mandatos coherentes.

El primer estado trivial es el estado "apagado", en el que la consola no está activa. Inicialmente no tiene sentido, pero es importante saber que en este estado estará nuestra consola si la forzamos a la detención. En este estado la única acción lógica es iniciar la consola.

Por otro lado, recien iniciada la consola, tenemos el estado principal. En este trabajo lo llamaremos "esperando consulta". En este estado existen diferentes acciones. Realizar una consulta o reiniciar la consola. La segunda opción es teóricamente idempotente.

Cuando se realiza una consulta la consola entra en el estado de "pensando". Este estado es interesante y se analizarán en el siguiente punto algunas propiedades referentes a él. Generalmente, la consola vive este estado como una transición de recibir una consulta y dar una respuesta. Dependiendo de la respuesta, la consulta se resolverá con un error o con una respuesta. Si la respuesta no ofrece más soluciones se volverá al estado de "espera consulta". Sin embargo, si existe solución la consola frenará en ella y quedará en un nueuvo estado.

En el estado de "esperando *backtraking*", la consola solo permitirá dos acciones: *next*, que calcula la siguiente solución; o stop, que finalizará el *backtracking* y volverá al estado principal.



En la figura anterior se ven los estados comentados anteriormente y sus transiciones. Además aparece que tipo de acción se ha realizado para cambiar de estado. Es importante señalar que todas las acciones son humanas excepto las

que se llevan a cabo en el estado “pensando” a no ser que el usuario decida matar la consola.

Sincronía

Es importante conocer el funcionamiento y el tratamiento de las consultas. La consola, a un ritmo humano (salvo casos de consultas costosas), no presenta problemas para realizar tareas muy seguidas o escribir mientras la consola está “pensando”. Pero al crear un librería que pueda mandar consultas a Ciao a una gran latencia, es importante tener el cuenta si Ciao hace una cola de peticiones y las gestiona síncronamente o desprecia todas aquellas consultas que son realizadas mientras está “pensando”.

En la consola de Ciao se puede ver gracias al predicado `pause/1` el comportamiento que tiene ante este tipo de situaciones. La consola de Ciao acumula en un *buffer* las entradas y las ejecuta en orden. Esto quiere decir que mientras la consola está en el estado “pensando”, el usuario puede introducir mandatos. Estos mandatos tendrán mayores riegos de tener respuestas inesperadas.

```
Ciao 1.24.0 [LINUXx86_64]
?- pause(5).
X is 5.

yes
?-
X = 5 ?
```

Después de finalizar con éxito la pausa, la consola devuelve el valor `yes`, que índica la correcta terminación de `pause/1`, muestra el *prompt* y por último ejecuta el mandato en cola.

```
?- x(6,7) = x(U, W), pause(5).
;
;
;

U = 6,
W = 7 ?
no
?- .
{SYNTAX ERROR: (lns 16-23) . cannot start an expression
; ; .
** here **
}
```

Este ejemplo puede llegar a darse cuando hay gran frecuencía de escritura, o pensamos lo que devolverá una consulta. Es interesante ver que hay que escribir un punto para que salte el error sintáctico ya que la consola en caso de error sintáctico espera a que se remedie y te permite dar saltos de línea.

Por otro lado, la evaluación de una consulta puede generar loops o ciclos infinitos si no se establece una condición de salida clara. La consola no impone un límite de tiempo o un control automático sobre la profundidad de búsqueda de la consulta. Si una consulta entra en un ciclo infinito, la consola permanecerá esperando una respuesta durante un tiempo indefinido, lo cual puede hacer que el sistema se vuelva no interactivo hasta que se interrumpa el proceso. Esto significaría tener que matar el proceso actual y relanzar la consola de Ciao Prolog.

```
?- assertz((loop :- loop.)).
{SYNTAX ERROR: Malformed query}
?- assertz((loop :- loop)).

yes
?- loop.
```

En conclusión, la consola ejecuta los mandatos o consultas de forma sícrona, pero los mandatos pueden ser escritos en cualquier momento. Estos serán ejecutados en orden.

3.1.2. Principales mandatos y predicados

En el uso de Ciao Prolog, nososotros utilizamos unos mandatos escritos o señales para interactuar con la consola. Estos mandatos son los que deben ser implementados en una librería de conexión con Prolog.

- Consulta: Es el concepto de poder escribir una entrada.

  ```
  ?- t(X, 5) = t(4, Y).
  ```

- *Next*: Es el mandato que permite continuar con una búsqueda en *backtracking*.

  ```
  X = 4,
  Y = 5 ? ;
  ```

- *Consult*: Permite abrir módulos y ficheros para hacer sobre ellos consultas.

  ```
  ?- consult('natural_numbers.pl').
  ```

- *Halt*: Este mandato es el que finaliza una sesión de consola.

  ```
  ?- halt.
  ```

- Señales: Sin utilizar el comando “halt” se puede terminar una consulta a través de señales del propio sistema operativo.

  ```
  Ciao 1.24.0 [LINUXx86_64]
  ?- ^C
  Ciao interruption (h for help)? ^C
  {ERROR: read_term/3 - No handle found for sent signal control_c}
  ERROR: {Program ended with failure}
  ```

- *Aserta*: Añade una regla al principio.

```
?- asserta(university(upm)).

yes
?- university(X).

X = upm ? ;

no
```

- *Asertz*: Añade una regla al final.

```
?- assertz(alumn(upm, rgpb)).

yes
?- alumn(U, N).

N = rgpb,
U = upm ? ;

no
```

- *Retrac*: Elimina una regla.

```
?- retract(university(upm)).
```

- *Retracall*: Elimina todas las reglas de un predicado concreto.

```
?- retractall(alumn/2).
```

Estos predicados y mandatos son los mínimos que se deben implementar dentro de una biblioteca de conexión con la consola de Ciao Prolog.

3.1.3. Las salidas de Ciao Prolog

Como se ha analizado en el apartado anterior de componentes de la consola existen diferentes tipos de respuestas. Estas van a estar relacionadas con las posibles salidas que nos puede ofrecer el sistema. Con el fin de caracterizar y realizar una traducción de estas salidas vamos a ver que propiedades tienen.

Respuestas

Una *salida* es todo aquello que la consola de Ciao escribe por pantalla. Esto significa que todo lo que devuelve la consola es texto plano. Este texto plano sigue muchas veces patrones, aunque las salidas de Ciao pueden ser manipuladas por el usuario y son totalmente abiertas.

Dependiendo de lo introducido por el usuario o el estado en el que esté la consola, estas respuestas se podrán clasificar. Es importante resaltar que las respuestas con las que se va a trabajar son las de Ciao Prolog. Estas respuestas

pueden variar según el tipo de Prolog. Si el sistema Prolog que se está utilizando está muy ligado al estándar ISO Prolog, entonces las respuestas serán parecidas con algunas diferencias.

Errores

En primer lugar encontramos los errores. Los errores van encerrados entre llaves. Siguen el siguiente formato regular:

```
{ ERROR: <Causa del error> }
```

Soluciones

Las soluciones son otro tipo de salida. En primer lugar encontramos las soluciones triviales:

```
yes

no
```

Estas soluciones indican la finalización correcta de un mandato, la veracidad de una consulta o el tipo de final que ha tenido una búsqueda en *backtracking*.

Cuando la solucion no es trivial, se devuelve una solución. Las soluciuones son devueltas de forma natural de una en una, parando la búsqueda cuando se encuentra una. (Existen casos en los que a la salida se le da un formateo, ver en el el punto "Salidas con formato-3.1.3) Las soluciones suelen llevar la siguiente estructura:

```
Variable = termino , Variable = termino , ... ?
```

A esta estructura típica se le suele llamar “restricción”, como se puede observar una solución involucra a muchas restricciones. La variable de cada restricción fue especificada por el usuario cliente. Mínimo debe de existir una restricción y el número que haya se preserva en las siguientes soluciones para la misma consulta. Salida total a una consulta:

```
X = 4,
Y = 7,
Z = 9 ? ;

X = 5,
Y = 8,
Z = 10 ?

yes
```

Solución:

```
X = 4,
Y = 7,
Z = 9 ?
```

Restricción:

```
X = 4
```

Variables

Las variables son la primera parte de una restricción. Nativamente se escriben en mayúsculas y son introducidas por el usuario cuando escribe la consulta. Tiene asociada en cada respuesta un valor distinto o igual, pero la variable siempre posee el mismo nombre. En la figura anterior se puede ver que las variables son X, Y y Z.

Término

Los términos son el resultado real o bruto de una consulta. Salvo que la consulta sea para comprobar un predicado o la existencia de solución lo más importante es esta parte de la respuesta. Los términos pueden ser de tres tipos principalmente:

- Átomos: Los átomos son valores internos fáciles de parsear. Suelen ser constantes, números, cadenas de texto...

  ```
  ?- X is 5.

  X = 5 ?

  yes
  ?- asserta(nat(zero)).

  yes
  ?- nat(X).

  X = zero ?

  yes
  ?- asserta(predicado(h014)).

  yes
  ?- predicado(X).

  X = h014 ?

  yes
  ?-
  ```

- Listas: Entre corchetes lleva dentro de si otros términos.

```
?- X = [2, hola, t(X), [[]], [3,5, 7, 11]].

X = [2,hola,t(X),[[]],[3,5,7,11]] ? ;

no
```

- Estructuras: Las estructuras son más complejas. Son una cadena de texto seguida de unos paréntesis con términos. Este tipo de salidas nos da la idea de la libertad que pueden llegar a tener las respuestas de Prolog.

```
?- X = t(x, y(X, [0,1,2])).

X = t(x,y(X,[0,1,2])) ?

yes
```

Salidas con formato

Es muy importante tener en cuanta que no se puede definir una gramática estricta de salida, ya que el cliente puede modificar la salida incluso de una consulta usual. Con los predicados format/1 y write/n se pueden modificar. En una consulta normal:

```
?- low(X, s(s(s(0)))).

X = 0 ? ;

X = s(0) ? ;

X = s(s(0)) ? ;

no
```

Mientras, editando la salida:

```
?-  low(X, s(s(s(0)))), format("Un numero menor es ~w", [X]).
Un numero menor es 0
X = 0 ? ;
Un numero menor es s(0)
X = s(0) ? ;
Un numero menor es s(s(0))
X = s(s(0)) ? ;

no
```

También se pueden quitar el *backtracking* y dar todos los resultados a la vez:

```
?- low(X, s(s(s(0)))), format("Un numero menor es ~w~n", [X]), fail.
Un numero menor es 0
Un numero menor es s(0)
Un numero menor es s(s(0))

no
```

Evidentemente, estas salidas pueden ser diseñadas por el usuario y el deben saber manejarlas posteriormente.

```
?- low(X, s(0)), write('Esto es ruido para distorsionar la salida').
Esto es ruido para distorsionar la salida
X = 0 ? ;

no
```

3.2. UfleSe

Por otro lado, en este trabajo se plantea el desarrollo de una aplicación web reescribiendo una ya existente. La diferencia de nuestro desarrollo es el lenguaje y el *framework* con el que se va a desarrollar. Para el análisis debemos de fijarnos en las diferentes características y funcionalidades de la aplicación actual. El trabajo intentará simular y trabajar en su reimplementación.

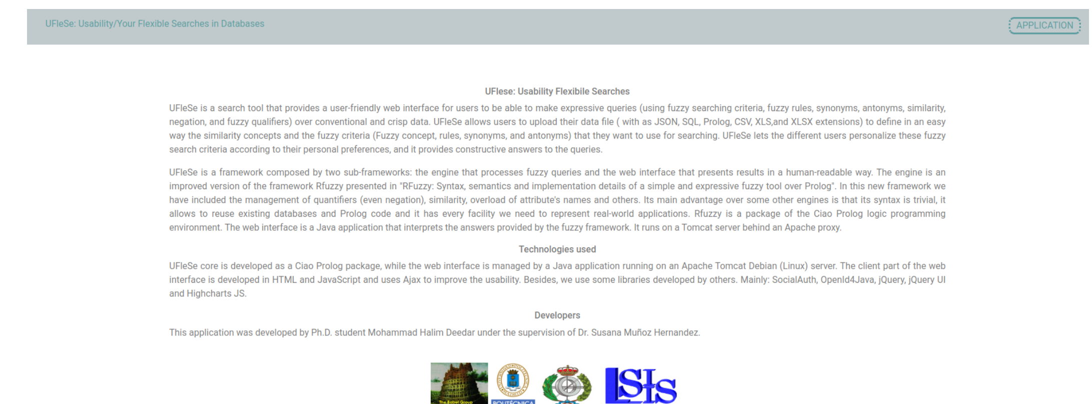
3.2.1. Estructura IU

La aplicación cuenta con varias pantallas separadas por las funcionalidades. Vamos a analizar sus componentes.

Entrada

En primer lugar tenemos la entrada la aplicación. Cuenta con dos pantallas:

1. Bienvenida:



2. Inicio de sesión:

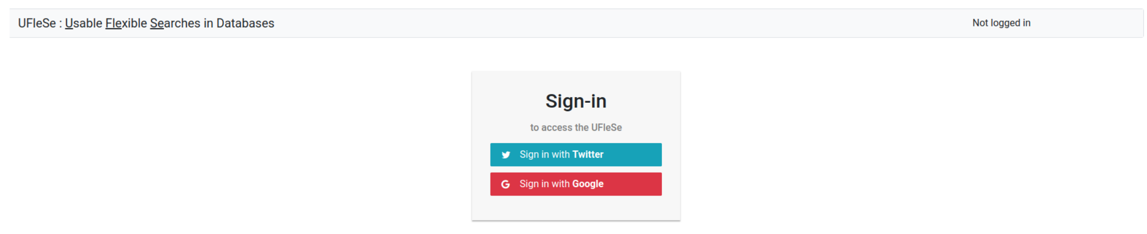


Figura 3.1: Pantalla de incio de sesión con Google o Twitter.

Pantalla Principal

En la pantalla principal se pueden ver tres opciones en la barra de navegación superior. Esta barra separa las pantallas o principales partes de la aplicación web. Además existe una breve introducción escrita de la aplicación.

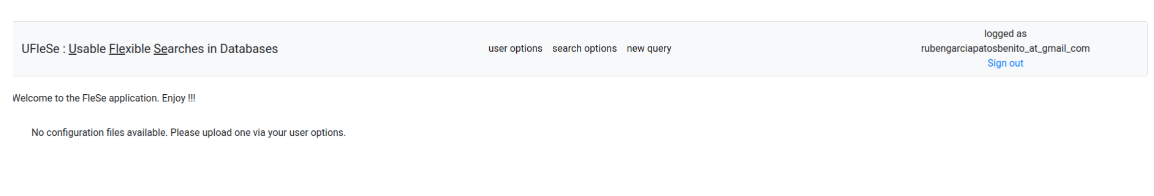


Figura 3.2: Pantalla de inicio.

Pantalla Opciones de Usuario

Primera pantalla de interacción con la aplicación. En ella aparece una tabla con los ficheros o bases de datos subidos por el usuario y un botón para subirlos.

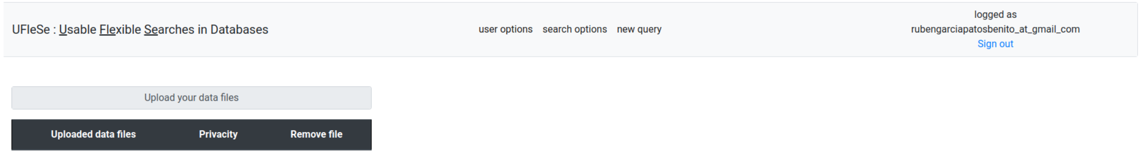


Figura 3.3: Pantalla de opciones de usuario sin interacción previa con la aplicación.

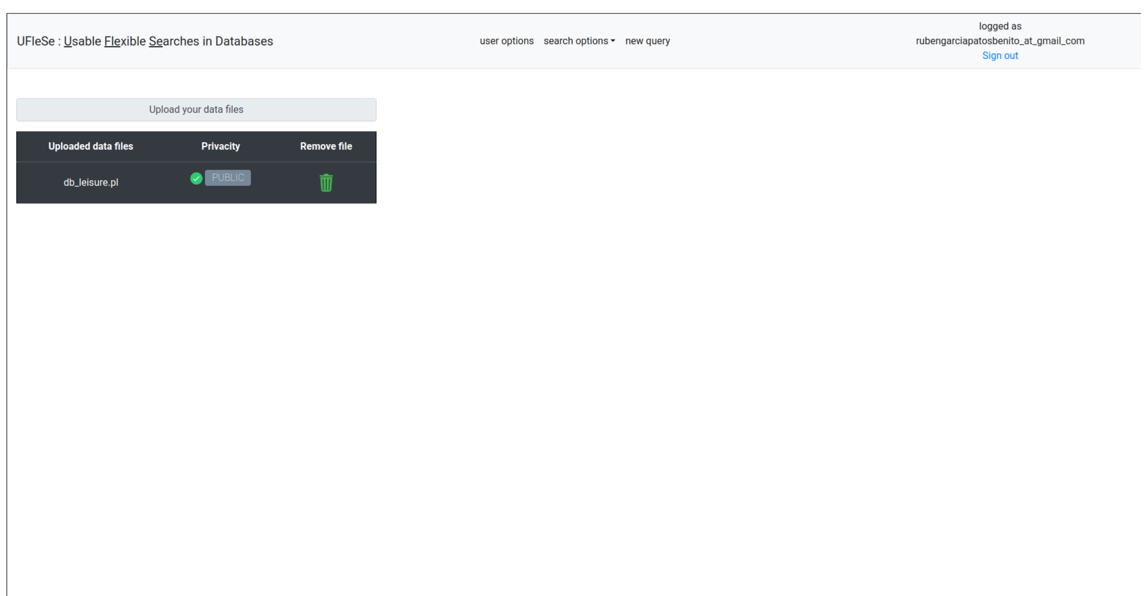


Figura 3.4: Pantalla de opciones de usuario con un fichero subido.

Configuración de búsquedas

Es una ventana con tres funcionalidades.

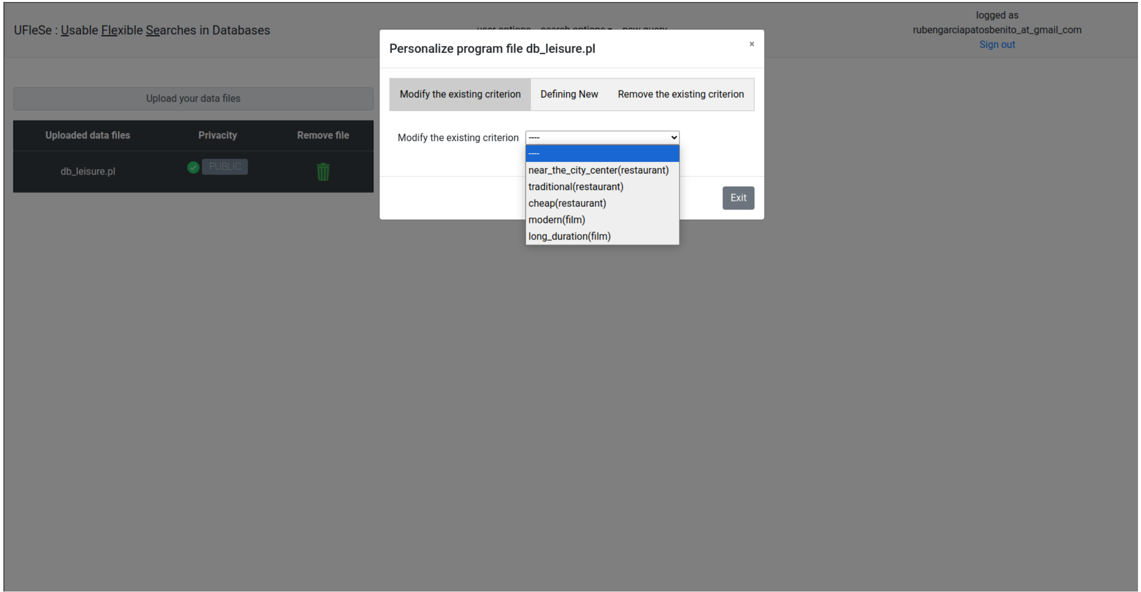


Figura 3.5: Pantalla de opciones de búsqueda. Crear, modificar y eliminar.

Pantalla Búsquedas

En esta pantalla van apareciendo los elementos de forma secuencial.

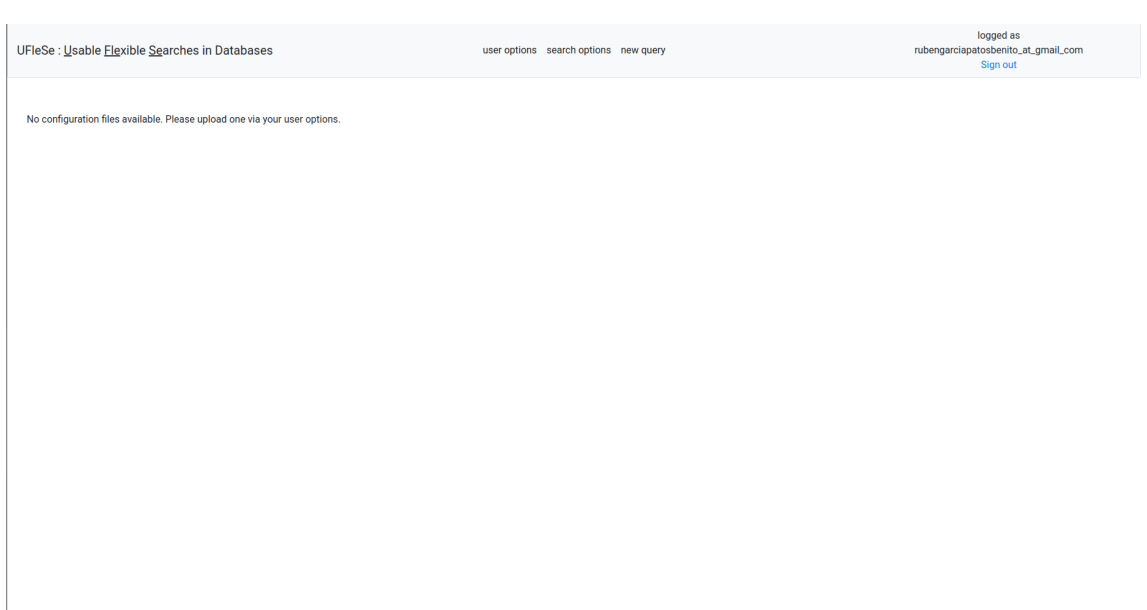


Figura 3.6: Pantalla de consultas. Sin interacción previa.

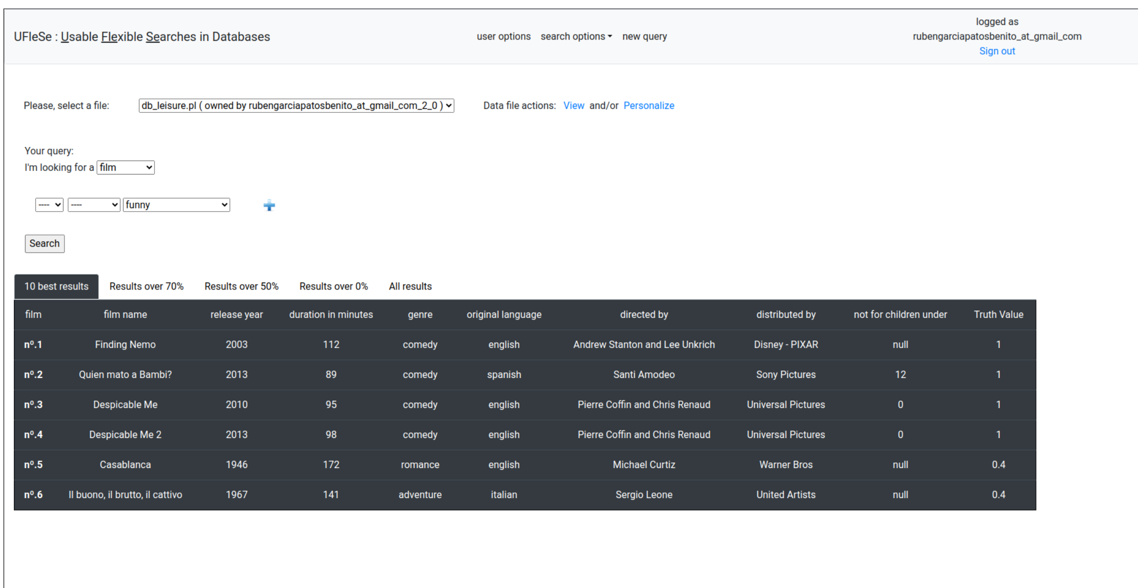


Figura 3.7: Pantalla de consultas. Consulta realizada.

3.2.2. Funcionalidades

El flujo de la aplicación es claro. Subida de archivos, configuración de filtros personales y consultas o búsquedas. Estas funcionalidades se ven separadas de forma muy sencilla gracias a la separación entre pantallas.

En primer lugar, en la pantalla de opciones de usuario se intuyen las siguientes funcionalidades:

- Subida de ficheros: Se pueden subir ficheros para su posterior consulta.

- Visualización de ficheros: Se pueden visualizar en una tabla todos los ficheros subidos, junto con su privacidad y la opción de borrado.

- Eliminación de ficheros: Se pueden eliminar los ficheros subidos. Estos ficheros ya no podrán ser tratados.

Mientras, en las opciones de búsqueda existen las siguientes posibilidades, dentro de cada fichero:

- Modificar un criterio existente: Se pueden modificar un criterio difuso en función de un criterio estricto. Te da opciones de configuración para todos los criterios existentes.

- Crear un criterio: Se pueden crear de diferentes tipos y sobre diferentes criterios ya existentes.

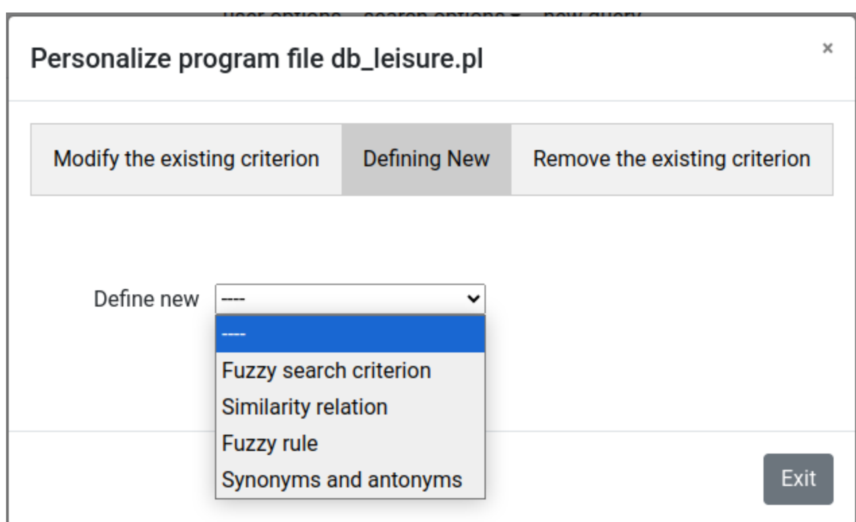


Figura 3.8: Opciones de nueva definición.

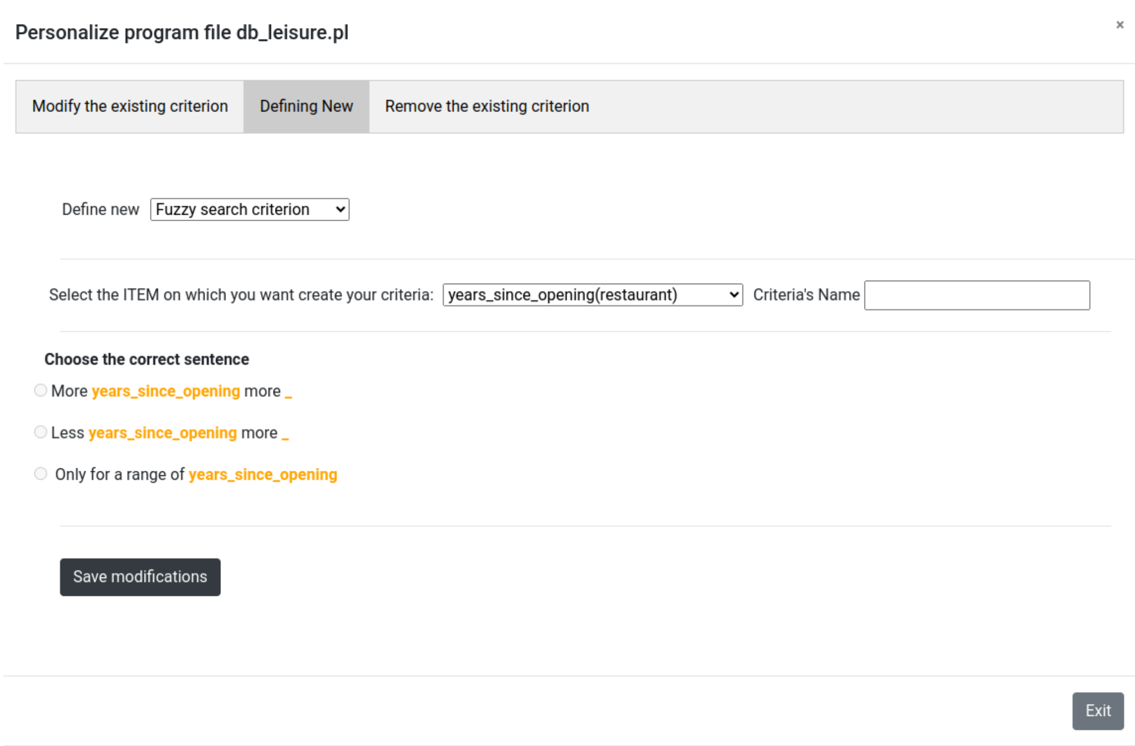


Figura 3.9: Opciones avanzadas para los nuevos criterios.

- Eliminar un criterio: Se elimina un criterio difuso.

Por ultimo tenemos las consultas.

- Selección de base de datos: Se puede seleccionar entre los ficheros que el usuario tiene subidos.

- Selección del dato: Sobre el fichero seleccionado se puede elegir el objeto sobre el que se van a hacer consultas. Suele ser una entidad de la base de datos.

- Creación de filtros y su manipulación: Se podrán poner filtros estrictos o difusos, pudiendo crear, eliminar o editar.

- Realizar la consulta y mostrar los datos: Se muestran los datos conforme a los filtros seleccionados por el usuario.

Desarrollo de una consulta:

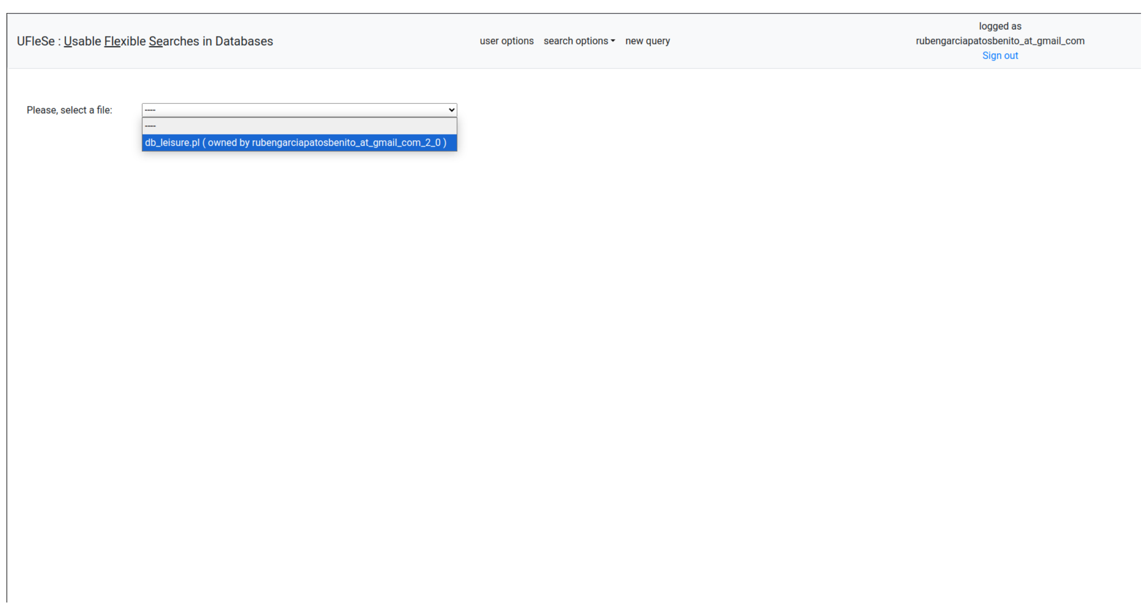


Figura 3.10: Seleccionando base de datos o fichero.

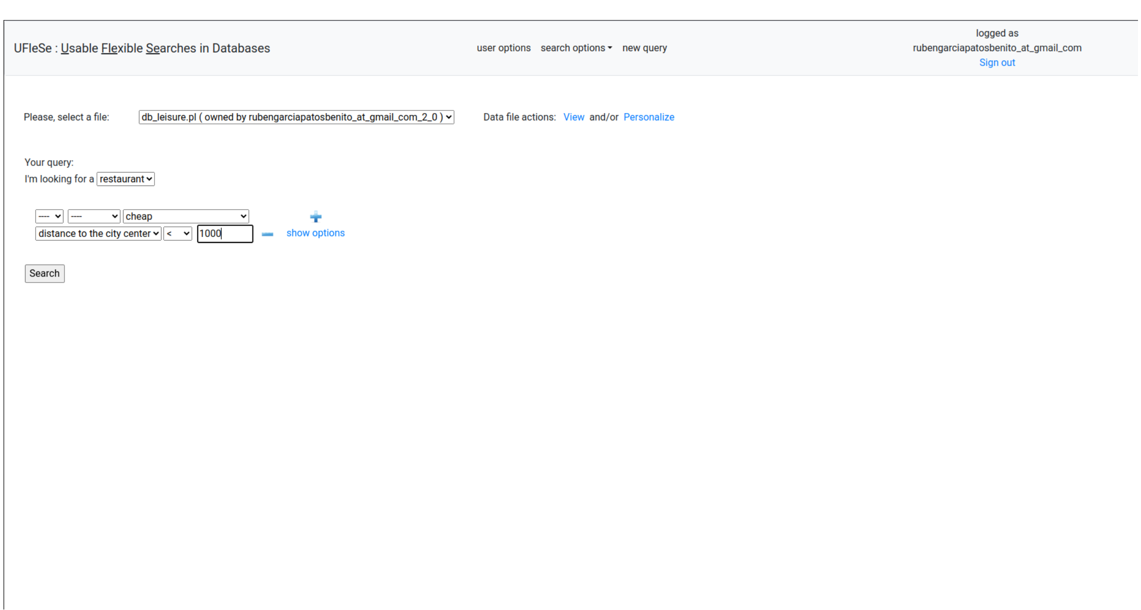


Figura 3.11: Seleccionada entidad. Realizando filtros.

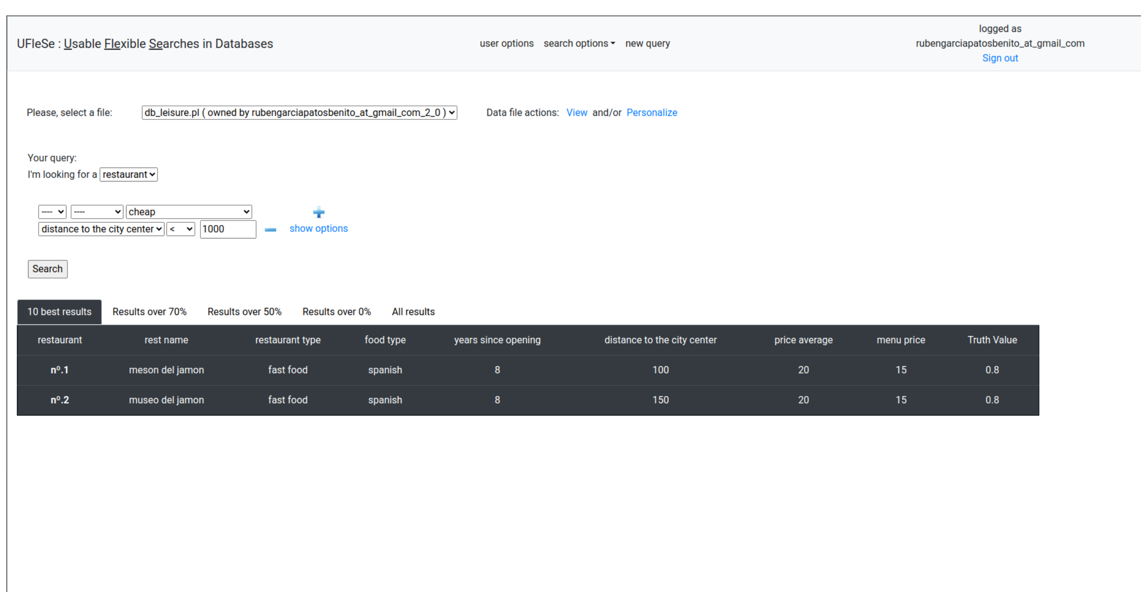


Figura 3.12: Consulta realizada.

Capítulo 4

Diseño

En este capítulo se trata el diseño de las partes a implementar en el trabajo. Primero se diseñan las funciones que se deben poder hacer desde una librería que simula la consola, después las traducciones más relevantes. Posteriormente, diseñamos lo referente a UfleSe, tanto las gestiones internas como lo más cercano al usuario.

- Prolixir-Conexión: Se centra en la satisfacción de los recursos que nos puede ofrecer Ciao. Se diseñan las funciones propias.

- Prolixir-Parser: Se centra en la traducción de los componentes simples estudiados en el análisis.

- UfleSe-*Backend*: Se compone de diferentes módulos para asistir a los requerimientos de la interfaz.

- UfleSe-*Frontend*: Es la interacción más cercana al usuario de la aplicación web desarrollada.

De manera general y para que el usuario entienda desde ya la estructura del proyecto, en la figura 4.1 se agrupan los componentes del proyecto. Los elementos más grandes se refieren a secciones en las que el trabajo se divide. Mientras en trabajo se sigue dividiento. Los elementos más pequeños son módulos que gestionan funcionalidades parecidas. Más adelante se verá que esto conformarán los futuros módulos de Elixir(Véase 4.1). Esta estructura es la diseñada teniendo en consideración las características de sus tripas o seguir un patrón de estructura sencillo. Todo ello se estudia en este capítulo.

Figura 4.1: Estructura/arquitectura proyecto.

4.1. Prolixir

Las funcionalidades que debe de tener la librería se ajustan a lo analizado en el Capítulo 3 y se describe en este.

4.1.1. Funciones de interacción con la consola

La interacción con la consola debe garantizar la implementación de las siguientes funciones:

- `start`: Su símil en consola es iniciarla. Sería como escribir "ciao" en una consola del sistema operativo. No recibe ningún parámetro (se puede implementar para recibir un nombre y que este sea el tipo de Prolog que se quiera utilizar Véase 2.2.1) y devuelve un valor según haya tenido éxito el arranque.

- `restart`: Equivalente a apagar y encender la consola de nuevo. La entrada es equivalente a la del punto anterior. Devuelve un valor según haya tenido éxito el reinicio.

- `consult`: Recibe una lista de nombres de módulos. Develve lo que la consola devolvería.

- `call`: Escribe un input en la consola de ciao. La entrada es la "*query*", la salida es el resultado nativo de la consola.

- `next`: No recibe parámetros. Devuelve la salida de la consola.

- `halt`: No recibe parámetros. Devuelve un indicador de la finalización.

- `asserta`: Recibe la regla a añadir. Devuelve la salida de la consola.
- `assertz`: Recibe la regla a añadir. Devuelve la salida de la consola.
- `retract`: Recibe la regla a eliminar. Devuelve la salida de la consola.
- `retractall`: Recibe el predicado a eliminar. Devuelve la salida de la consola.
- `listing`: No recibe parámetros de entrada. Devuelve la salida de la consola.

En caso de un error extraordinario no vinculado con la consola, las funciones deben devolver un error que indique la causa(este punto es más propio de la implementación 5).

Por otro lado, en el capítulo de análisis se estudiaba la sincronía de la consola (Véase 3). Para el diseño de nuestra librería las llamadas serán síncronas. Si un cliente hace una petición este esperará que se le de la respuesta para realizar la siguiente. Pero diferentes clientes podrán hacer difetentes peticiones a la vez. Estas serán tratadas en orden de llegada, al igual que en la consola. Se almacenarán en una cola y se ejecutarán por orden de llegada dando la respueta al cliente.

A parte de las funciones principales, algunos otras funcinoes pueden ser interesantes cuando se este implemtentado un programa con esta librería.

1. info: Devolver la información de la consola: Donde corre, que estado tiene, consultas realizadas, soluciones, salidas... No recibe parámetros de entrada. Esta función es más general. Las siguientes devuelven elementos más concretos y más típicos.
2. get-internal-state: Ver el estado de la consola. No recibe entrada.
3. get-last-response: Devuelve la última salida de la consola.

4.1.2. Parser

Como se ha estudiado en el capítulo anterior las salidas son muy variadas. Por ello se debe implementar funciones que transformen desde los elementos más sencillos hasta los más generales.

Las funciones que tiene que implementar este módulo de Elixir, debe ser consistente. Para ello se ha hecho una gramática de las salidas de Ciao Prolog. *Veáse anexo B*

Así se deben implementar diferentes módulos de parsers:

1. Paser de términos: Recibe un término en forma de cadena de caracteres devuelve el valor del término en la estructura nativa del lenguaje. Las lista las transforma en listas y sus elementos también los parsea de forma recursiva; las estructuras las deja como texto y por último, los átomos según su tipo los transforma: número, tanto entero como psudo-real, las transforma en número; y las cadenas de texto las deja en texto. En caso de no identificar el tipo, se devuelve la entrada.

2. Parser de restricciones: Se trata como una solución con solo una variable.
3. Parser de soluciones: La entrada es texto y conforma un conjunto de unificaciones. Si no mantiene la estructura esperada devuelve la entrada. Cuando la entrada es correcta se devuelve un mapa con claves los nombres de las variables y valores una tupla compuesta por la relación y el término parseado.
4. Parser de salida: La entrada es una salida total a una consulta. La respuesta de la función es una lista de mapas. Todos tienen las mismas claves. Cada mapa es la traducción de una solución.

Cadena de entrada	**Traducción**
`"54"`	`54`
`"h0l4"`	`"h0l4"`
`"t(64, X)"`	`"t(64, X)"`
`"[32, [], hola, [[0]]]"`	`[32, [], "hola", [[0]]]`
`"X=5"`	`%{"X"=> {"=", 5}}`
`"X=5, Y=3 ?"`	`%{"X"=> {"=", 5}, "Y"=> {"=", 3}}`
`"W=t(X,9) ? W=t(Y,10) ? yes"`	`[%{"W"=> {"=", "t(X,9)"}},` `%{"W"=> {"=", "t(Y,10)"}}]`

Cuadro 4.1: Traducciones

4.2. UfleSe

En esta sección se diseñan la aplicación web del nuevo UfleSe. Es importante conocer las características de la antigua aplicación. Estas fueron analizadas en la sección 3.2.

Como tradicionalmente se ha hecho para la implementación de las aplicaciones web, vamos a separar el diseño en dos partes. En primer lugar, el *Backend*, donde se estudiará todo lo referente a usuarios, ficheros y búsquedas. Mientras diseñaremos una interfaz de usuario, propio de la sección de *Frontend*.

4.2.1. *Backend*

Gestión de usuarios

Se debe tener una base de datos para gestionar las cuentas, los datos y los ficheros de los usuarios. Este trabajo de fin de grado no se centra en este tema, pero entre líneas se puede intuir la estructura de los datos. Se deben guardar los usuarios y sus datos en un tabla de usuarios. Por otro lado, se deben guardar todos los ficheros. Con estos debe haber dos campos más: privacidad y usuario al que pertenece. En caso de ser público, cualquier usuario podría ver el fichero y usarlo. En el caso contrario, solo podrá verlo y usarlo el poseedor.

Debemos tener un módulo que gestione las siguientes acciones o funciones:

- *create_user*: Crear nuevo usuario. Recibe datos de inicio y un correo. Se debe generar un id y almacenar en la base de datos correspondiente. Se debe verificar antes que no existe un usuario con las credenciales y datos.
- *delete_user*: Elimina un usuario. Recibe el *id* del usuario.
- *modify_user*: Modifica los datos de un usuario. Recibe el identificador y los nuevos datos. Debe garantizar que los datos asiciados al usuario que tienen interacción con la aplicación no se pierdan.

Gestión de ficheros

Cuando se tiene seleccionado un fichero sobre este se deben implementar las siguientes acciones.

- *add_file*: Añade un fichero y las opciones con las que se quiere guardar. Pueden ser fichero publicos y privados. Estos están asociados a un usuario.
- *delete_file*: Elimina un fichero gracias a su referencia. Solo se pueden borrar los ficheros personales.

Gestión modificación de ficheros

Cuando se edita un fichero añadiendo predicados, reglas o criterios difusos hay que tener en cuenta de quien es el fichero y que propiedades tiene.

Supongamos que un usuario desea modificar (ya sea añadiendo, eleminando o modifcando un criterio o elemento) un fichero o base de datos. En caso de que el fichero sea suyo y privado el cambio se realizará sobre el existente. Esto no tendrá ninguna repercusión mayor, ya que ningún otro usuario puede acceder al fichero. Mientras si es público y el usuario poseedor quiero editarlo podrá hacerlo y los efectos serán sobre todo el mundo. Mientras si un usuario no poseedor quiere editar un fichero público que no es de su propiedad, se creará un nuevo fichero copia con los campos personalizados y privado.

Esta implementación y la de la base de datos no estaba en la versión anterior, pero nos dan una idea de que necesidades surgirían en caso de avanzar con el desarrollo de esta aplicación.

Se deben gestionar entonces las funciones:

- *manage_edit_file*: Recibe el usuario que edita, la referencia al fichero que esta siendo editado y el nuevo fichero. Según los criterios vistos anteriormente, crea o edita ficheros y sus atributos dentro de la base de datos de ficheros.
- *add_criteria*: Se recibe el fichero en cuestión y el criterio. Se devuelve el fichero editado con el nuevo criterio.
- *add_fuzzy_criteria*: Se recibe el fichero en cuestión y el criterio. Se devuelve el fichero editado con el nuevo criterio difuso.

- *modify_criteria*: Se recibe el fichero en cuestión, el criterio a modificar y las nuevas propiedades del fichero. Se devuelve el fichero editado con el criterio modificado.
- *modify_fuzzy_criteria*: Se recibe el fichero en cuestión, el criterio a modificar y las nuevas propiedades del fichero. Se devuelve el fichero editado con el criterio difuso modificado.
- *delete_criteria*: Se recibe el fichero en cuestión, el criterio a eliminar, tanto extricto como difuso. Se devuelve el fichero editado el nuevo criterio.

El flujo de trabajo de este módulo es el siguiente. Se crea el fichero que se está editando y posteriormente se ve que gestión hacer con el existente. Evidentemente, solo es posibles borrar el anterior o crear uno nuevo.

Gestión de Predicados y Criterios

Para muchas de las funcionalidades futuras se debe tener una buena gestión de que entidades tenemos en cada base de datos, que atributos tiene, que criterios difusos, etc.

Por ello estos métodos son esenciales para el desarrollo de muchas partes de la aplicación web.

- *get_entities*: Recibe un fichero y devuelve las entidades de la base de datos sobre los que se pueden hacer consultas.
- *get_predicates*: Recibe un fichero y la entidad sobre el que actúan los predicados. Devuelve los predicados no difusos.
- *get_fuzzy_predicates*: Recibe un fichero y la entidad sobre la que actúan los predicados. Devuelve los predicados difusos.
- *get_atributes*: Recibe el fichero y la entidad en cuestión. Devuelve los atributos y sus tipos en una lista de tuplas.

Gestión de Consultas

Para hacer búsquedas se debe de tener un módulo que las gestione.

- *select*: Como entrada tiene una entidad y un límite de resultados. Devuelve todas las instancias que existen de la entidad hasta que se llega al límite.
- *select_with_query*: Como entrada recibe la consulta y el límite de resultados. Devuelve los resultados de la consulta mientras que no se exceda el límite.

4.2.2. *Frontend*

Para el diseño de esta parte del proyecto nos guiaremos por las funcionalidades y en que componentes se puede separar. Así se diseñarán las pantallas de la aplicación y que funcionalidades estarán habilitadas en cada una.

Pantalla inicio de sesión

En la pantalla de inicio tendremos la posibilidad de iniciar sesión o crear un usuario. Además se podrá iniciar sesión mediante otros medios. Este punto no es el más relevante y puede ser modificado con facilidad.

Estas funcionalidades están directamente conectadas con el módulo de gestión de usuarios.

Elementos comunes en la aplicación

La aplicación mantendrá una estructura sencilla similar a la vista en la implementación existente. En la parte superior, habrá una cabecera con nombre, logo y enlaces interesantes. Existirá una barra de navegación en la zona izquierda de la pantalla que permitirá navegar por los tres principales sitios de la aplicación: inicio, configuración y búsquedas. A parte, existirá una pantalla extra para el manejo de los usuarios.

Pantalla usuarios

Estrechamente relacionado con el módulo de usuarios. 4.2.1 Permitirá al usuario ver los datos de su cuenta.

Los componentes son:

- Fondo usuarios.
- Tabla información de usuario.

Pantalla de inicio

Es una pantalla sencilla pero ya forma parte de la aplicación. Es una entrada a la aplicación sencilla en la que no agobian las funcionalidades de golpe. En ella habrá información escrita referente al trabajo y a la página web. Figurará información sobre el uso y manejo de las ventajas que ofrece el producto.

Los componentes son:

- Fondo inicio.
- Texto.

Pantalla Configuración

Encierra gran cantidad de funcionalidades. Se divide en dos grandes campos.

1. Gestión de ficheros: En esta sección se puede ver una tabla con todos los ficheros que el usuario tiene. Se puede borrar y editar opciones de los ficheros. Está relacionado profundamente con el módulo de gestión de ficheros del back. 4.2.1
2. Gestión de edición de ficheros: Se podrá seleccionar un fichero para añadir, borrar o editar criterios sobre él. Esta relacionado con el módulo de

gestión de cambios sobre ficheros. Además, para la edición se necesitarán las funciones del módulo de gestión de predicados. 4.2.1

Los componentes son:

- Fondo configuración.
- Componente sección ficheros: contiene la tabla con los datos de los ficheros subidos y el botón para añadir más.
- Componente sección edición: contine selecionador de fichero y ventana de edición.
- Tabla ficheros: Cada fila posee la información de un fichero. Tiene tres columnas: nombre, privacidad y boton de eliminar.
- Botón subida.
- Seleccionador.
- Ventana edición.

Pantalla de búsquedas

Por último en la pantalla de búsquedas se manejarán las funciones del módulo de predicados(para el filtrado en la consulta) y el de gestión de consultas(para conocer las respuestas de Prolog). 4.2.1

Los componentes son:

- Fondo búsqueda.
- Componente búsqueda: Incluye los seleccionadores y los filtros.
- Componente resultado: Incluye la tabla.
- Seleccionador de base de datos.
- Seleccionador de entidad.
- Filtros: Existen de muchos tipos. Son líneas compuestas por despleglables. Llevan un botón para eliminar filtro.
- Filtro.
- Botón añadir filtro.
- Boton de inicio de búsqueda.
- Tabla con resultados.

Capítulo 5

Implementación

El siguiente capítulo estudiamos que decisiones de implementación que se han tomado. En la primera sección se elige la herramienta de `GenServer` para ser el esqueleto de la conexión Prolog-Elixir. Posteriormente, se explica como ha sido la implementación de la aplicación web con Phoenix *framework*[5]. Se describen los elementos utilizados y como han sido empleados como los *controllers*, el enrutamiento y la interfaz de usuario final.

5.1. Uso de *GenServer* para la conexión

Dentro de la librería de Prolixir, se tenía que poner solución a la gestión de procesos y comunicación entre ellos. En primer lugar, en el contexto de la conexión Ciao Prolog, van a existir tres tipos de procesos que interactúan entre si con el fin de implementar las funciones descritas en el capítulo anterior.

El primer proceso importante es el proceso Ciao. En él corre la consola de forma natural. Tiene una entrada y una salida. Estas están redirigidas al segundo proceso de este módulo.

El proceso `Engine`, es un proceso GenServer.[6] Cuando se crea este proceso, automáticamente se levanta el proceso Ciao y se configuran su entrada y sus salidas. Este tipo de proceso nativo de Elixir tiene muchas carácterísticas que nos interesan. Analicémoslas:

- Estado: El proceso posee un estado que nos servirá para guardar información relevante sobre la conexión. Después estudiaremos que datos nos interesa guardar. El estado irá cambiando según vayan realizandose peticiones.
- Sincronía: El GenServer está implementado de tal manera que permite gestionar llamdas síncronas y asíncronas. Esto es ideal, porque los clientes pueden hacer muchas llamadas desde diferentes procesos, pero desde cada uno solo una llamada. Desde el GenServer se gestionará sincronamente sin que se den condiciones de carrera por escribir en la consola. Además, esto garantiza que el estado no sea inesperado.

- Manejo de errores: En caso de errores el GenServer se reinicia solo. Además existen otras muchas opciones y ventajas.

El estado del GenServer será un map de Elixir. En él se guardarán diferentes valores importantes:

- Proceso Ciao: Se guarda la referencia al proceso Ciao, para poder mandarle entradas.

- Estado de la consola: El estado de la consola de Ciao. Como estudiamos en el capítulo de análisis, la consola puede estar en diferentes estados.

- Salidas: Será una lista de pares. Cada par contendrá una query realidaza y su correspodiente salida. Se almacenarán todos los pares generados a lo largo de la sesión.

- Salida actual: en caso de que una salida esté en curso todavía, se almacena en forma de texto.

- Consulta actual: Si hay una consulta no cerrada o en backtracking la contiene.

- Otros: existen otros valores guardados que se necesitan para el correcto funcionamiento de la consola.

Por último están los procesos clientes. Estos procesos son los que quieren tener una conexión con Ciao y los que utilizan la librería de Prolixir para ello. Hacen llamadas al GenServer(proceso "Engine"), estas llamadas son síncronas, es decir si un proceso pide al GenServer, este quedará bloqueado (salvo por un timeout implementado automaticamente por las llamadas a GenServer) hasta que haya una respuesta. Pero por otro lado, diversos clientes pueden hacer llamadas al GenServer, este las gestionará síncronamente.

Es importante saber que cada Engine tiene un proceso Ciao asociado distinto. Esto permite varios procesos separados corriendo simultáneamente.

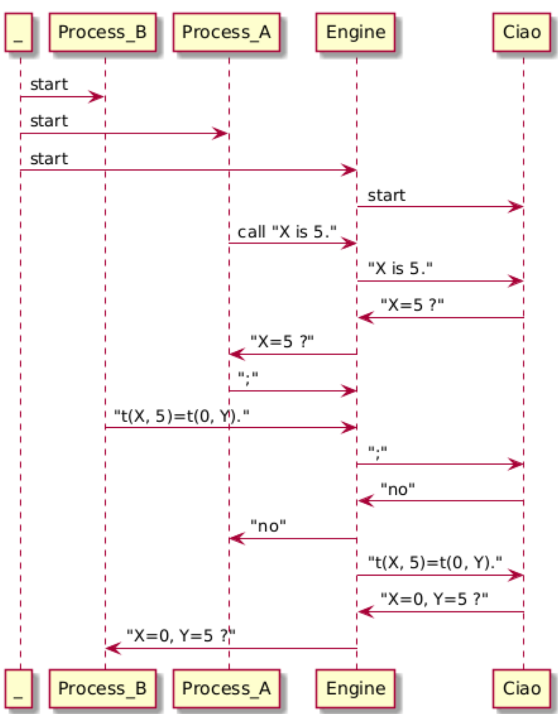


Figura 5.1: Comunicación entre procesos. Conexión con Ciao.

En la figura 5.1 aparece un ejemplo que aclara el funcionamiento de `Engine`. En la imagen aparecen 4 procesos y un proceso "padre". Empecemos por describir que son:

El proceso sin nombre, es un proceso que arranca *A*, *B* y `Engine`. Este proceso figura para remarcar que el proceso *Ciao* lo inicia `Engine`.

El proceso A y B, son procesos cliente de`Engine`. Estos procesos pueden estar realizando tareas irrelevantes para el contexto de Prolixir y en un momento determinado hacer una llamada a `Engine`.

El proceso `Engine` actúa de mediador entre Elixir y el proceso *Ciao*.

El proceso *Ciao* es el proceso de la consola de Ciao Prolog. Su entrada y salida están conectados a `Engine`. Este proceso es arrancado por `Engine` y solo debe tener interacción con él.

En la imagen se muestra como el proceso `Engine` es arrancado y este inicia a *Ciao*. El proceso *A* realiza una petición a `Engine`, y esta se resuelve de la forma usual. Tras esto, el proceso *A* vuelve a realizar una llamada e inmediatamente despues el Proceso *B* también se involucra. Gracias a `Engine` las peticiones no se mandan a *Ciao* si no que se gestiona la primera y la segunda aguarda hasta que la primera se resuelve. Tras terminar `Engine` la primera llamada(y esto incluye enviar la respuesta al proceso *A*), este resuelve la petición pendiente y manda la correspondiente respuesta al proceso *B*.

5.2. UfleSe con Phoenix

La implementación de una aplicación web con Phoenix era desde el principio una de las principales ideas. Pero con Phoenix se podía enfrentar el problema desde dos perspectivas: el desarrollo clásico de *front-end* o el desarrollo moderno de *front-end*.

Para este trabajo se ha optado por una resolución más tradicional pero ajustándose a lo especificado por el Capítulo 4.2.

En esta sección veremos que elementos influyen en la implementación web, como los *controllers* de Phoenix y el enrutamiento, y el aspecto de la interfaz de usuario.

5.2.1. Controladores

La aplicación web se ha divido en tres *controllers*:

1. *home-controller*: Este controlador se encarga de cargar la pantalla de inicio. Cuando este controlador es llamado renderiza una página estática.
2. *configuration-controller*: Es el controlador que se encarga de rederizar cuando el usuario hace una acción relacionada con la gestión de ficheros.
3. *query-controller*: Este controlador renderiza cuando se está realizando una búsqueda. Controla acciones como seleccionar fichero, selecionar dato, añadir filtros y hacer la consulta.

Esta división se ha llevado a cabo ya que cada uno de estos *controller* dirige una pantalla principal y así se gestiona de forma más sencilla desde el punto de vista humano. Además, utilizan distintos módulos del *backend* de UfleSe (Cápitulo 4.2.1). De esta forma un nuevo desarrollador en el proyecto puede entender rápidamente la estructura.

5.2.2. Enrutamiento

Las acciones que el usuario realiace en la aplicación web las gestionan los controladores antes especificados.

En este apartado se muestran las rutas para las acciones y quien se encarga de gestionar o controlar cada acción.

Ruta	“GET `/`”
Descripción	Carga pantalla de inicio
Controlador	*`home_controller`*

Ruta	“GET `/configuration`”
Descripción	Carga la pantalla de configuración, salen ficheros cargados
Controlador	*`configuration_controller`*

Ruta	“POST /configuration/upload”
Descripción	Sube un nuevo fichero
Controlador	*configuration_controller*

Ruta	“POST /configuration/delete”
Descripción	Borra un fichero
Controlador	*configuration_controller*

Ruta	“POST /queries”
Descripción	Carga la pantalla de consultas
Controlador	*queries_controller*

Ruta	“GET /queries/:file”
Descripción	Inicia proceso de busqueda sobre un fichero
Controlador	*queries_controller*

Ruta	“GET /queries/:file/add_filter”
Descripción	Añade un filtro más
Controlador	*queries_controller*

Ruta	“GET /queries/:file/show_results”
Descripción	Muestra los resultados de la búsqueda
Controlador	*queries_controller*

5.2.3. UI

Las pantallas y componentes se han diseñado conforme a lo diseñado en el capítulo 4.2. Manteniendo el flujo de la aplicación ya implementada se ha optado por esta presentación gráfica de los componentes.

Componentes

Todas las pantallas cuentan con una cabecera en la parte superior con el nombre de la aplicación, y referencias a información relacionada con el trabajo.

Asimismo, existe también otro componente que aparace siempre. La barra de navegación permite a los usuarios navegar entre las principales pantallas de la aplicación. Incluye la página de inicio(5.2), la página de configuración(5.3) y la página de consultas(5.5).

Inicio

Configuración

Consultas

Perfil

Entrando en el terreno de las partes principales de la aplicación tenemos la pantalla de inicio. En ella aparece información sobre el trabajo, pequeñas indicaciones de uso y referencias.

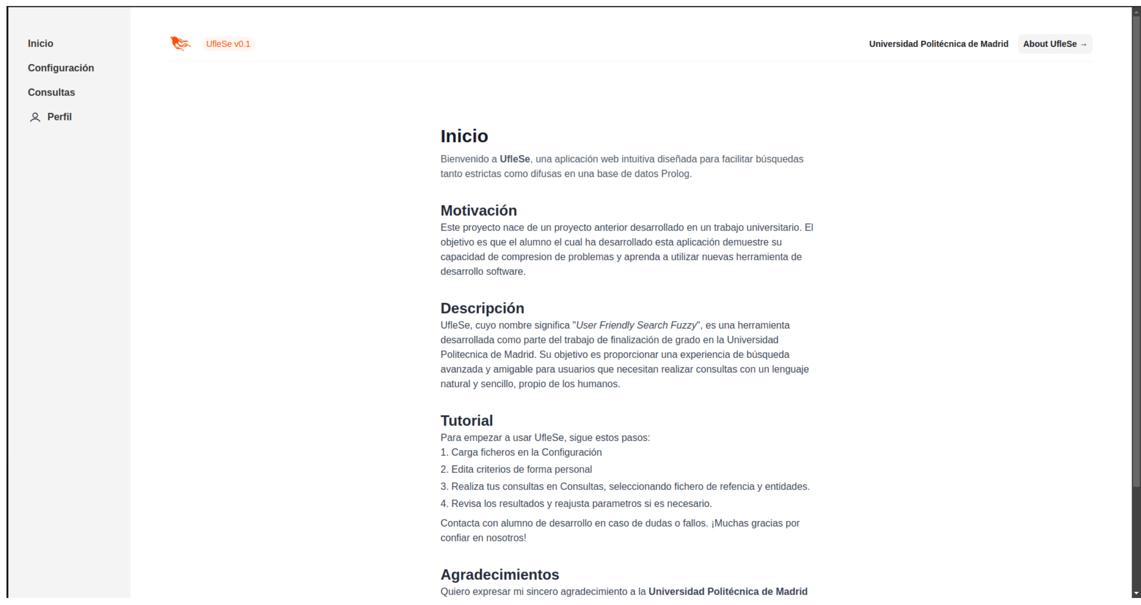


Figura 5.2: Pantalla de inicio.

La pantalla con más densidad de la aplicación en la referente a la configaración. Esta se divide en dos grandes partes: Gestión de subida de ficheros y gestión de edición de ficheros. La figura 5.3 es una configuración previa a añadir ficheros. Mientras en la figura 5.4 se muestran los ficheros subidos, datos y la opción de eliminar.

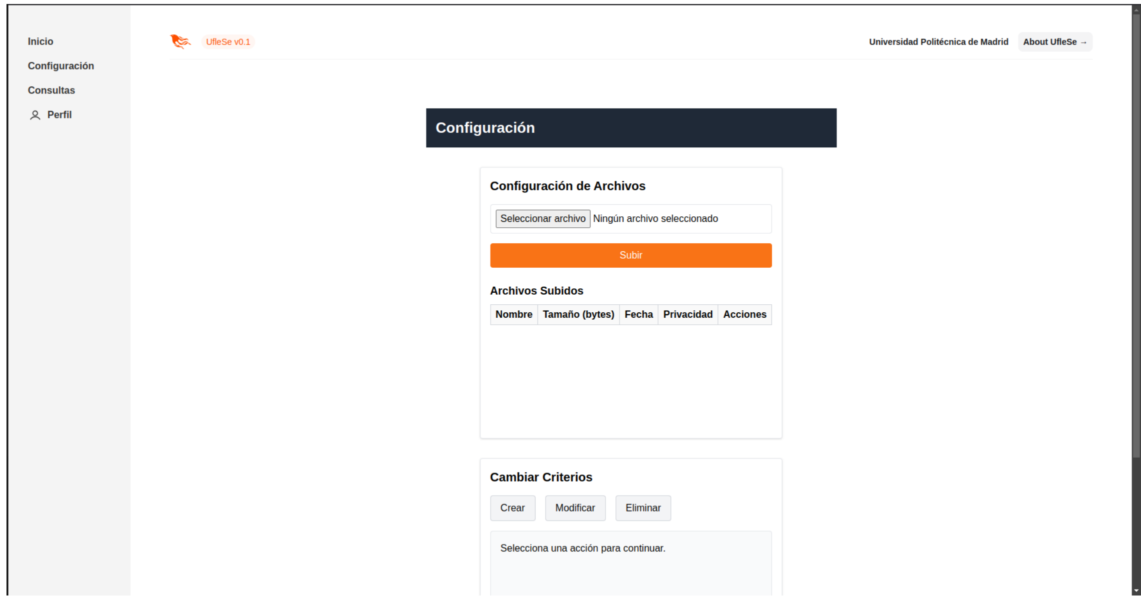


Figura 5.3: Configuración sin iteracción con aplicación.

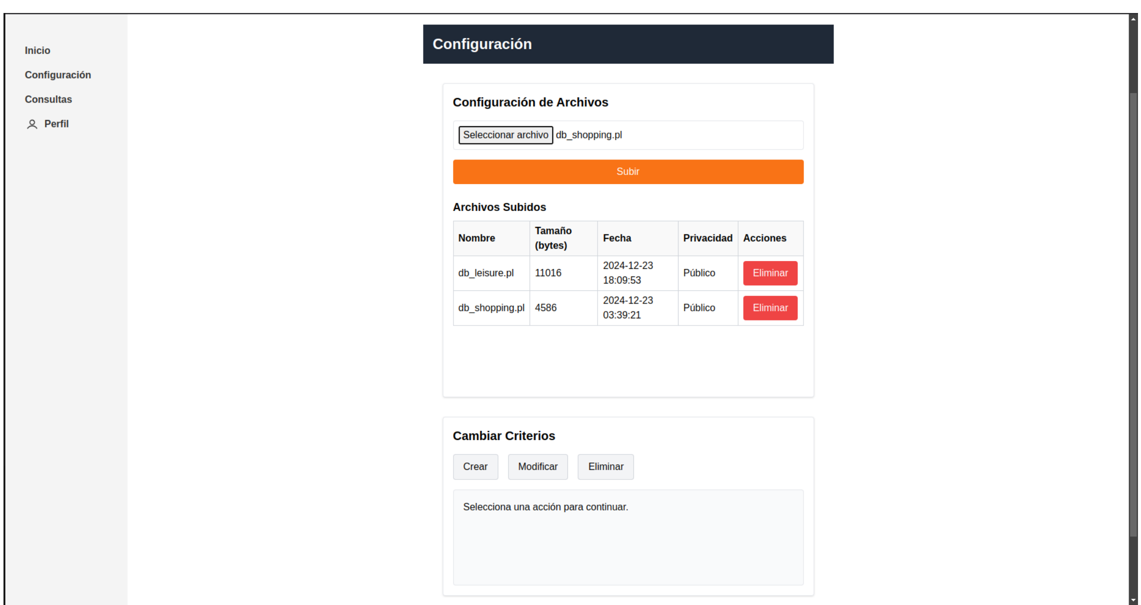


Figura 5.4: Configuración con ficheros cargados.

La pantalla de consulta es la que lleva el título del trabajo. Toda ella tiene se centra en hacer una consulta. En la pantalla se puede apreciar una segunda barra de navegación que permite al usuario seleccionar el fichero sobre el que quiere realizar una búsqueda. En la figura 5.5 se aprecia como todavía no se ha seleccionado ningún fichero. Mientras, en la figura 5.6 la consulta está realizada. El usuario ha ajustado los filtros según lo analizado (figura 3.12) y diseñado (Capítulo 4.2) en el trabajo. El resultado es una tabla con los resultados de la búsqueda.

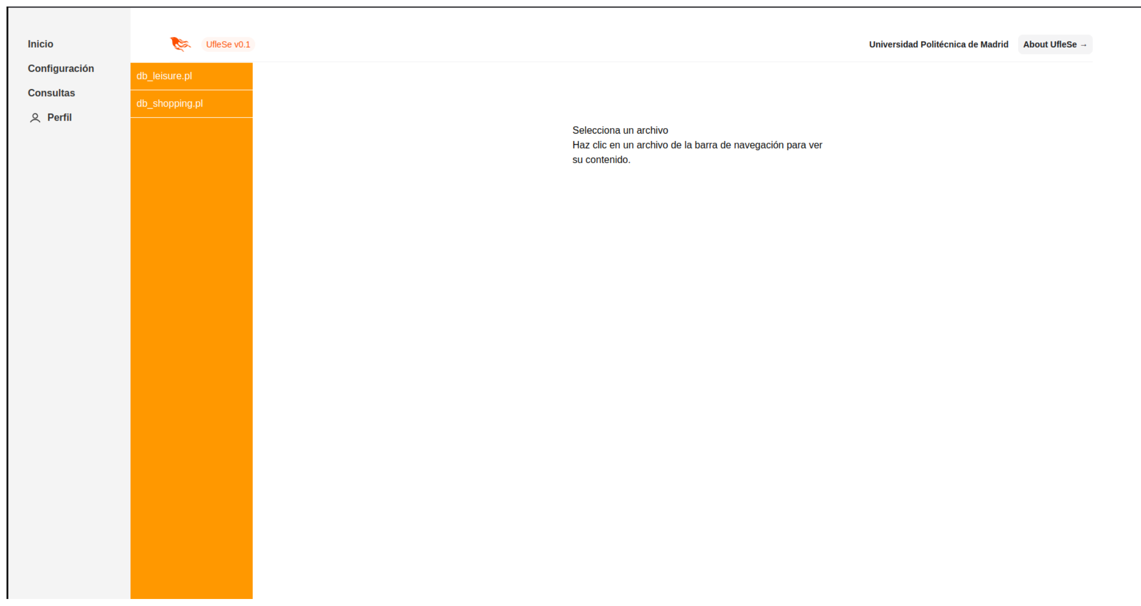


Figura 5.5: Pantalla de Consultas. Iniciando consulta.

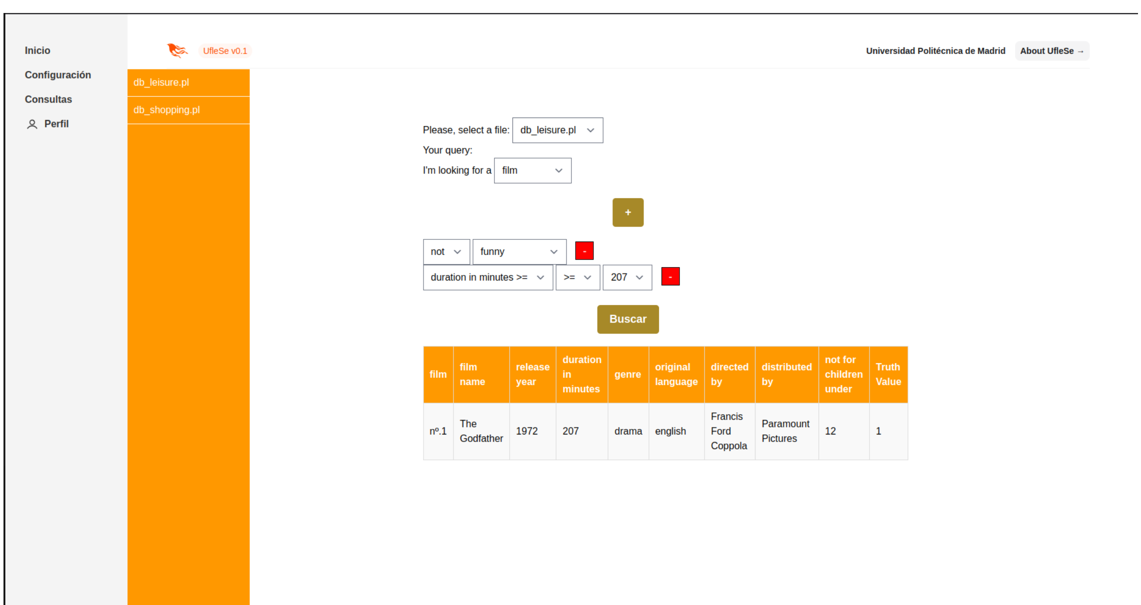


Figura 5.6: Pantalla de Consultas. Resultado de búsqueda.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Resultados

El principal resultado de este TFG está centrado en la implementación de la Herramienta de UfleSe. Este objetivo es muy grande, y por tanto, se han alcanzado en el proceso otros muchos de menor tamaño, pero no menos importantes.

Los resultados más señalados son:

1. Manejo de nuevas herramientas.
2. Conocimiento del paradigma funcional.
3. Prolixir.
4. Realización de la documentación.
5. Desarrollo web.
6. Primera versión del nuevo UfleSe.

Frente a estos objetivos, inicialmente partiamos de unos que han sido alcanzados con éxito en su mayoría.

A nivel personal también se han conseguido pequeños logros pero importantes para el estudiante. En primer lugar, se ha conseguido desarrollar una librería de Elixir. Es una de las primeras aportaciones personales que he realizado. Esto me ha animado a aprender más de este lenguaje y el paradigma funcional. Además, siento ganas de seguir trabajando en ella y publicarla. Por otro lado, me ha gustado darme cuenta de la gran cantidad de herramientas que existen y están disponibles para facilitar muchas tareas. Me insta a ser consciente de lo importante que es una buena documentación para aprovecharnos de los que trabajamos entre todos. Por último, y no menos importante, me he dado cuenta de la importancia de no tener miedo a preguntar cualquier duda. Es un trabajo que siempre me ha costado, pero al igual que trabajamos juntos y compartimos herramientas para desarrollar mejor, siempre son bien recibidas las preguntas cuando van acompañadas de ganas de aprender.

6.1.1. Objetivos técnicos no resueltos

Para ahora comprender los objetivos que quedan pendientes para un posible trabajo futuro hay que tener en cuenta como ha finalizado la primera versión de UfleSe.

De entre las funcionalidades que se habían analizado y exitían de la versión incial de la que partimos, están las más importantes completadas.

Quedan aún así dos módulos importantes:

1. Usuarios: Falta la funcionalidad de inicio de sesión. Manejo de usuarios y sus ficheros.

2. Modificación de criterios: Falta la funcionalidad de edición de criterios y, evidentemente, su gestión dentor de los usuarios.

6.2. Nuevos objetivos

La nueva primera versión de UfleSe se ha alcanzado. La comunicación con Ciao, la estructura, el diseño de aplicación... son todos objetivos alcanzados. La parte que queda es interesante documentar que pasos a seguir se deben dar.

En primer lugar, es importante señalar que la versión de UfleSe anterior a esta no tenía una gestión de usuarios rígida y los ficheros siempre se mantenían públicos. Esto no ha podido ser tratado, pero sería interesante implementarlo como se ha detallado en este mismo trabajo. Esto permitiría al ususario tener una experiencia con la aplicación más cómoda, privada y segura.

Para añadir estas caracterísitcas hace falta analizar los siguientes puntos.

- Base de datos: Qué base de datos se debe seleccionar.

- Formato de datos: Qué datos se van a guardar y cómo.

- Almacenamiento ficheros: Cómo guardar los ficheros. En este trabajo se propuso guardar todos en la misma base de datos y, además del fichero, información sobre privacidad y dueño.

Por otro lado, este tipo de trabajos son interesantes en el ámbito de búsquedas sencillas básicas. Se podrían buscar nuevas características para la aplicación como compartir bases de datos generales entre usuarios, hacer búsquedas de bases de datos públicas. . . Estas ideas son ambiciosas y poco rigurosas, pero se podrían aplicar o quizás añadir ideas de este trabajo a otras herramientas que ya trabajen con datos compartidos.

En el plano de diseño, se puede trabajar en un diseño más profesional de la aplicación, buscando atractividad y mayor facilidad en el uso.

6.3. Conclusión final

Como conclusión final, siento que le debo mucho a este trabajo. Me ha dado más ganas de seguir aprendiendo, me ha enseñado lo mucho que me queda por crecer profesionalmente y las ganas que tengo de ello.

Bibliografía

[1] M. H. D. y Susana Muñoz-Hernández, «UFleSe: User-Friendly Parametric Framework for Expressive Flexible Searches», 2020.

[2] L. S. y Ehud Shapiro, *The Art of Prolog*. The MIT Press, 1994.

[3] S. G. y. C. V. Susana Muñoz-Hernández, «RFuzzy: Syntax, semantics and implementatios details of a simple and expresive fuzzy tool over Prolog», *ELSEVIER-Information Sciences*, 2010.

[4] Elixir. «Documentación oficial Elixir». (), dirección: `https://elixir-lang.org/` (visitado 01-06-2025).

[5] Phoenix. «Documentación oficial Phoenix». (), dirección: `https://hexdocs.pm/phoenix/up_and_running.html` (visitado 01-06-2025).

[6] Elixir. «Documentación oficial Elixir-GenServer». (), dirección: `https://hexdocs.pm/elixir/GenServer.html` (visitado 01-06-2025).

Anexos

Apéndice A

Conceptos de Prolog

Este anexo existe para ayudar al lector en caso de no haber tenido contacto con Prolog o si quiere refrescar algunos conceptos.

Para hacer un analisis del shell de Ciao Prolog primero debemos tener conocimiento de los distintos tipos de datos que tiene Prolog y sus nombres.

Prolog, y algunos dialectos como Ciao Prolog, se estructuran en los siguientes términos:

- Hechos: Son afirmaciones simples que representan datos o relaciones básicas en el programa. Por ejemplo: `hombre(juan).` o `padre(juan, maria).`.
- Regla: Es una cláusula que define la lógica de un predicado usando una cabeza y un cuerpo. Una regla tiene la forma `{Cabeza :- Cuerpo.}`, donde el cuerpo especifica las condiciones necesarias para que la cabeza sea verdadera. Por ejemplo:

  ```
  ancestro(X, Y) :- padre(X, Y).
  ancestro(X, Y) :- padre(X, Z), ancestro(Z, Y).
  ```

- Predicados: Se refiere al concepto lógico que se describe mediante hechos y reglas. Es el núcleo lógico de Prolog y puede estar compuesto por múltiples hechos y/o reglas.
- Consultas: Son preguntas realizadas al sistema para verificar si una cláusula lógica es verdadera o para obtener valores que cumplan una determinada condición. Por ejemplo:

  ```
  ?- padre(juan, maria).
  ```

- Elementos del lenguaje: Operadores lógicos y relacionales, variables, listas, estructuras,
- Clausulas especiales: Propios de librerías especificas. Requiere un estudio especifico.

Hay que diferenciar que es un módulo y que es un fichero con reglas. Un módulo regula la exportacion de predicados. Si no se exportan especificamente los predicados dentro de un modulo son privados. Mientras, en un fichero con predicados y reglas, estas son visibles al cargar el fichero.

Apéndice B

Gramatica de salida Ciao Prolog

No terminales: {Salida, Respuesta, Soluciones, Restriccion, Relacion, Termino, Variable, Atomo, Lista, ElementosLista, Error, LineaError }

Terminales: { yes, no, ‘?’, ‘{’, ‘}’, identificador, cadena, numero, ‘,’, ‘=’, .>. , .<. , .=. , ‘[’, ‘]’ }

Axioma: Salida

Reglas de derivación:

<Salida>::= <Respuesta>| <Error>

<Respuesta>::= ’yes’ | ’no’ | <Soluciones>?

<Soluciones>::= <Restriccion>[’,’ <Soluciones>] <Respuesta>

<Restriccion>::= <Variable><Relacion><Termino>

<Relacion>::= ’=’ | ’.=.’ | ’.<.’ | ’.>.’

<Termino>::= <Atomo>| <Lista>| <Estructura>| <Variable>

<Variable>::= identificador

<Atomo>::= cadena | numero

<Lista>::= ’[’ <ElementosLista>’]’

<ElementosLista>::= <Termino>[’,’ <Termino>] | λ

<Estructura>::= cadena ’(’ <Termino>[’,’ <Termino>] ’)’

<Error>::= ” <LineaError>”

<LineaError>::= cadena

> *Nota: Los símbolos ’[ ]’ significan que los elementos dentro de si, se pueden repetir un numero indeterminado de veces. El símbolo λ indica que el elemento del que ha derivado puede ser vacio.*

Ejemplo de salida:

X = [1,[3]], *Y* = *s*(0, *h014*) ? *yes*

<Respuesta>->

<Soluciones>? ->

<Unificación>[’,’ <Soluciones>] ? <Respuesta>->

<Variable><Relacion><Termino>[’,’ <Soluciones>] ? <Respuesta>->

X = <Lista>[’,’ <Soluciones>] ? ->

X = ’[’ <ElementosLista>’]’ [’,’ <Soluciones>] ? <Respuesta>->

X = ’[’ <Termino>[’,’ <Termino>’]’ [’,’ <Soluciones>] ? <Respuesta>->

X = ’[’ <Atomo>[’,’ <Termino>] ’]’ [’,’ <Soluciones>] ? <Respuesta>->

X = ’[’ 1 [’,’ <Termino>] ’]’ [’,’ <Soluciones>] ? <Respuesta>->

X = ’[’ 1 ’,’ <Termino>[’,’ <Termino>] ’]’ ? <Respuesta>->

X = ’[’ 1 ’,’ <Lista>[’,’ <Termino>] ’]’ ? <Respuesta>->

...

X = [1, [3]] [’,’ <Soluciones>] ? <Respuesta>->

X = [1, [3]] , <Variable><Relacion><Termino>? <Respuesta>->

X = [1, [3]] , Y = <Termino>? <Respuesta>->

X = [1, [3]] , Y = <Estructura>? <Respuesta>->

X = [1, [3]] , Y = s(<Termino>[’,’ <Termino>]) ? <Respuesta>->

X = [1, [3]] , Y = s(<Atomo>[’,’ <Termino>]) ? <Respuesta>->

X = [1, [3]] , Y = s(0 [’,’ <Termino>]) ? <Respuesta>->

X = [1, [3]] , Y = s(0 ’,’ <Termino>) ? <Respuesta>->

X = [1, [3]] , Y = s(0 ’,’ <Atomo>) ? <Respuesta>->

X = [1, [3]] , Y = s(0 ’,’ h014) ? <Respuesta>->

X = [1, [3]] , Y = s(0 ’,’ h014) ? yes

Este documento esta firmado por

	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Tue Jan 14 23:53:58 CET 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)