Escuela
Politécnica
Superior

Diseño e implementación de un sistema software que facilite la utilización de sensores usando ontologías.

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Eduardo Grande Ruiz (alumno)
Tutor/es:
Juan Andrés Montoyo Guijarro (tutor)

Universitat d'Alacant
Universidad de Alicante

Mayo 2022

Diseño e implementación de un sistema software que facilite la utilización de sensores usando ontologías.

Autor
Eduardo Grande Ruiz (alumno)

Tutor
Juan Andrés Montoyo Guijarro (tutor)
Departamento de Lenguajes y Sistemas Informáticos

Grado en Ingeniería Informática

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Mayo 2022

Resumen

A la hora de desarrollar una aplicación, es muy importante plantear una buena arquitectura para así poder tener un buen punto de partida. Las aplicaciones modernas suelen tener una arquitectura muy bien definida, lo que se traduce en un bajo acoplamiento, el uso de microservicios o APIs para la interconexión de datos, o la definición de diferentes capas entre las cuales se reparte el código.

Las ontologías son una de las formas de representación de conocimiento en formato digital más utilizadas. Learning Engine Through Ontologies (LETO) es una aplicación desarrollada conjuntamente por el Grupo de Procesamiento de Lenguaje Y Sistemas de Información de la Universidad de Alicante (GPLSI) y la Escuela de Matemáticas y Computación de la Universidad de la Habana (Cuba). El objetivo de la aplicación es permitir introducir información de diferentes fuentes, tanto estructuradas como no estructuradas, y traducir esa información a ontologías. Una vez se tienen esas ontologías almacenadas, se contará con una base de conocimiento la cual podrá ser representada utilizando diferentes visualizadores (mapas, histogramas, gráficos…). Esa representación vendrá dada por una consulta realizada por el usuario en lenguaje natural, la cual será procesada para extraer de la base de datos la información relevante con la consulta realizada.

LETO actualmente cuenta con diversos problemas derivados de la falta de una arquitectura clara. Durante el presente trabajo se plantea una arquitectura de tres capas para la aplicación, lo que permitirá desacoplar el código existente al crear una nueva capa de presentación desde cero, así como una API para realizar el intercambio de datos. La aplicación implementa una serie de sensores que utilizan modelos de inteligencia artificial para procesar los textos introducidos. Esos sensores también se encontraban acoplados, por lo que se plantea una nueva estructura para que sea más sencilla la gestión e implementación de sensores.

Para poder realizar esto, se ha realizado un análisis del código actual, revisando los requisitos que la aplicación ya implementa y los que no, para así trazar un plan de desarrollo para la puesta en marcha de esas funcionalidades requeridas. LETO se encuentra escrita en Python, por lo que todo el código introducido se ha escrito en ese lenguaje. Se ha creado una API REST implementada con la librería FastAPI en Python. La nueva web de LETO se ha desarrollado utilizando HTML, CSS y JavaScript desde cero.

Esta nueva arquitectura planteada permitirá solventar los problemas actuales, además de abrir un abanico de posibilidades de cara al futuro, al otorgar flexibilidad a la hora de implementar nuevas funcionalidades, como utilizar LETO desde una aplicación móvil, o la creación de nuevos sensores que permitan procesar información de dominios concretos, como puede ser información del sector hotelero o información médica.

Agradecimientos

Este trabajo no habría sido posible realizarlo sin la ayuda de mi tutor, Andrés Montoyo, el cual me ha guiado durante estos 5 meses para que todas las tareas salieran adelante satisfactoriamente. Agradecer también al equipo de profesionales que trabajan en el Grupo de Procesamiento del Lenguaje Natural y Sistemas de Información por toda la ayuda y conocimientos recibidos a lo largo del desarrollo de este trabajo.

Además, este trabajo se entiende como la culminación de mi paso por el Grado en Ingeniería Informática en la Escuela Politécnica Superior de la Universidad de Alicante. Cuatro años de duro trabajo que terminan aquí. Me gustaría agradecer a todos mis compañeros, al personal docente e investigador y al personal de administración y servicios por haber hecho de este tiempo una etapa inolvidable. Han sido unos años de trabajo intenso y de aprendizaje continuo que no habría sido posible sin el apoyo de mi familia y amigos. A todos ellos, infinitas gracias.

Un dicho popular entre muchos estudiantes universitarios dice: “No pases por la universidad sin que la universidad pase por ti”. Objetivo cumplido.

Imposible es solo una gran palabra lanzada por hombres pequeños que encuentran más fácil vivir en el mundo que se les ha dado, que explorar el poder que tienen para cambiarlo.

Muhammad Ali.

Índice general

Índice de figuras

Índice de Códigos

Lista de Acrónimos y Abreviaturas

API	Application Programming Interface.
CSV	Comma-separated value.
GPLSI	Grupo de Procesamiento de Lenguaje Y Sistemas de Información de la Universidad de Alicante.
KDD	Knowledge Discovery in Databases.
LETO	Learning Engine Through Ontologies.
OWL	Web Ontology Language.
REST	Representational State Tranfer.
W3C	World Wide Web Consortium.
WWW	World Wide Web.

1 Introducción

En este capítulo se explicará el punto de partida para realizar el desarrollo propuesto para este Trabajo de Fin de Grado. Para ello, se definirán una serie de objetivos, así como explicar la metodología empleada en el desarrollo e implementación de la propuesta.

1.1 Motivación

Debemos comprender que la sociedad es diversa, por lo que no todo el mundo sabe utilizar un ordenador, entender que es una ontología, o mucho menos programar. Es por ello por lo que se deben de realizar sistemas informáticos intuitivos, sencillos, y pensados para poder ser usados por la mayor cantidad de gente posible, independientemente de su grado de conocimiento técnico.

Actualmente, el uso de aplicaciones para procesar y analizar información de textos y sacar conclusiones de ellos, se encuentra limitado a personas con grandes conocimientos técnicos, como pudiera ser un investigador del área del procesamiento del lenguaje natural.

Es por ello por lo que los nuevos avances científicos en el campo del procesamiento del lenguaje natural y de la inteligencia artificial actualmente no están democratizados, es decir, no están al alcance de toda la población.

Por otro lado, es muy importante que las aplicaciones se realicen de forma estructurada y siguiendo patrones de diseño. Realizar aplicaciones acopladas dificulta la implementación de nuevas funcionalidades, además de impedir una fácil mantenibilidad del código. Seguir las metodologías que marca la ingeniería del software, tales como la obtención y análisis de requisitos o la definición de una arquitectura, hace que el riesgo de fracaso de una aplicación sea menor, así como aumentar la calidad de esta.

Por todo esto, se han de democratizar los usos de los nuevos avances en el campo del procesamiento del lenguaje natural creando aplicaciones intuitivas y manejables por cualquier tipo de usuario, sin depender de si es experto o no en ese campo. Además, las aplicaciones creadas deberán de contar con una estructura que haga el software mantenible y entendible por los diferentes desarrolladores, usando procesos y metodologías que viabilicen todos esos aspectos.

Como base para este Trabajo de Fin de Grado se utilizará una herramienta llamada LETO, desarrollada conjuntamente por el GPLSI y la Escuela de Matemáticas y Computación de la Universidad de la Habana (Cuba). LETO es un marco de aprendizaje basado en ontologías usado para extraer conocimiento de diversas fuentes de información, las cuales formarán un único conocimiento semántico. Esa base de conocimiento semántico se puede enriquecer de diversas formas, ya sea introduciendo datos estructurados (por ejemplo, en un CSV) o datos no estructurados (por ejemplo, un texto o una entrada de Wikipedia).

Se detallan a continuación las funcionalidades con las que cuenta LETO:

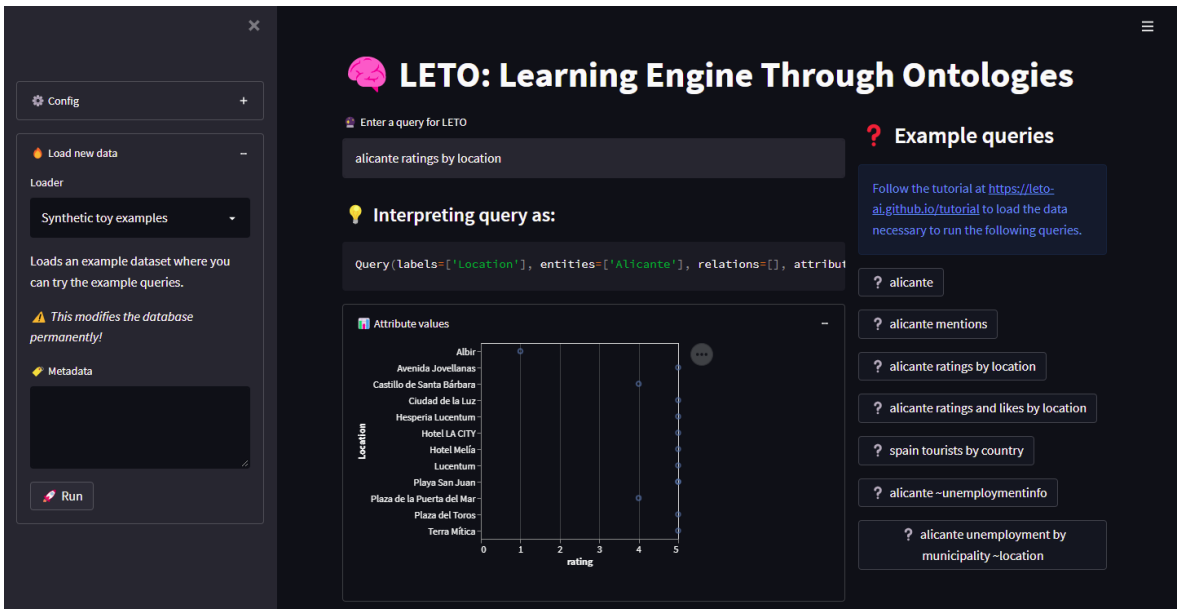

Figura 1.1: Web actual de LETO

- Carga de información de ejemplo: Se pueden introducir una serie de datos de ejemplo que contiene la plataforma. Concretamente son datos sobre la Revolución de Octubre y la Revolución Cubana. Esta funcionalidad permite probar de forma sencilla si todo está funcionando correctamente, sin tener que introducir ningún archivo.
- Carga manual de tuplas: La aplicación permite introducir tuplas de forma manual, siguiendo el formato `entidad[:tipo] - relación - entidad[:tipo]` y pudiendo especificar o no el tipo de la entidad.
- Carga de ficheros CSV: Se pueden subir archivos CSV con datos estructurados. Además, se permite indicar el nombre del tipo de entidad que se creará (propiedad `main_entity`).
- Carga de ficheros de texto: La aplicación permite cargar ficheros de texto con información no estructurada.
- Carga de texto plano: Se permite cargar texto plano, especificándolo en la aplicación. Las frases introducidas deberán estar estructuradas como sujeto-verbo-objeto para que sean reconocidas por la aplicación.
- Carga de datos desde una página de Wikipedia: Se puede especificar una sentencia para buscar una página en Wikipedia, y cargar y procesar así la información que contenga. En este caso, se permite establecer una sensibilidad (entre 0 y 1) que se aplicará a la hora de buscar una página en Wikipedia en base a la sentencia introducida.

En todos los casos menos en el de carga de datos de ejemplo y carga de tuplas se debe de especificar el lenguaje de la información que se está introduciendo, escogiendo entre castellano o inglés. Además, en todos los casos se permiten introducir metadatos que serán contemplados por el sistema a la hora de interpretar la información cargada. Por último, al introducir

información en cualquiera de los formatos, se muestra un mensaje informando al usuario sobre el número de datos generados.

En cuanto a la base de datos, la aplicación permite visualizar el número de datos guardados, así como configurar parámetros de visualización relacionados con los datos, como el número máximo de resultados a mostrar. También, el usuario puede decidir eliminar toda la información que contiene la base de datos.

Por otro lado, y como parte fundamental del sistema, el usuario puede introducir consultas para que LETO las interprete y busque información en función de esa interpretación. Al introducir la consulta, se muestra la interpretación realizada, mostrando las entidades, relaciones o atributos que se están buscando en la base de datos.

Una vez el sistema ha rescatado los resultados, estos son mostrados al usuario mediante una serie de visualizadores, los cuales se explican a continuación:

- Gráfico de entidades: Se muestra un gráfico de entidades en el que encuentra representada toda la información. Además, se muestra información concreta sobre cada entidad representada, pudiendo pinchar en ella para ver esa información.

- Mapa: Se muestra un mapa en el que están señaladas las localizaciones que están representadas en los datos rescatados.

- Entidades y relaciones: Se muestra un gráfico circular en el que se representan los datos obtenidos junto con el porcentaje de aparición con respecto al total.

- Esquema: Se muestran tablas de entidades y relaciones en las que se muestran todos los atributos rescatados junto con el número de veces que aparecen.

- Valores de los atributos: Se muestra una gráfica en la que se representan dos coordenadas.

Estos visualizadores se muestran en función de la información mostrada, es decir, no se muestran siempre los mismos o en el mismo orden. Por ejemplo, en caso de haber información sobre localizaciones, se mostrará el mapa. O en caso de haber información fechada, se mostrará un histograma.

Una vez acabada la explicación de la aplicación a alto nivel, se procederá a explicar el estado del sistema a bajo nivel.

Actualmente LETO se encuentra íntegramente desarrollado en Python, usando principalmente la librería Streamlit, la cual permite crear de forma rápida aplicaciones web. Los desarrolladores eligieron esa librería al necesitar un mínimo producto viable (*MVP* en inglés) cuánto antes.

Derivada de esa necesidad de tener un producto software de manera rápida, la aplicación carece de una arquitectura clara. No se encuentran sendas diferencias entre el *front-end* y el *back-end*, estando el proyecto únicamente estructurado en una serie de carpetas, las cuales de detallan en la Figura 1.2.

En la raíz del paquete `leto-mvp` se encuentran 5 directorios, así como 4 ficheros. Para comprender la estructura actual y la utilidad de los diferentes archivos, se explicarán brevemente a continuación las partes importantes de la aplicación:

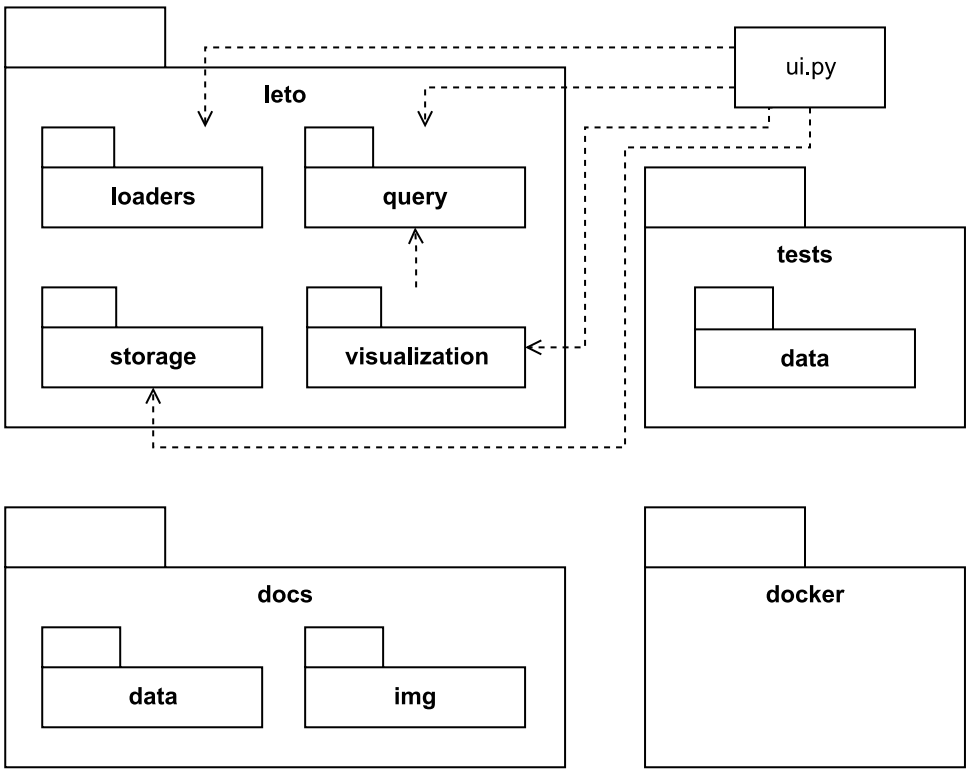

Figura 1.2: Diagrama con los paquetes de LETO

- `leto.py`: Archivo principal de configuración de Streamlit en el que se configuran parámetros generales del sitio web a desplegar por esta librería.
- `makefile`: Archivo utilizado para usar con la instrucción `make` para así construir la imagen de la aplicación, ejecutar una terminal del contenedor, crear una imagen de la documentación…
- `mkdocs`: Archivo de configuración utilizado por el contenedor de la documentación de LETO.
- */data*: Directorio que contiene archivos necesarios para el funcionamiento de la aplicación, como por ejemplo el directorio *models*, el cual incluye los modelos de inteligencia artificial usados por los sensores de la aplicación.
- */docker*: Directorio que contiene todos los archivos necesarios para la gestión de contenedores de la aplicación utilizando Docker. Entre estos archivos se encuentran
  - `dockerfile` y `docs.dockerfile`: archivos utilizados para construir los contendores de forma automática.
  - `docker-compose.yml`: Archivo YAML en el que se encuentran definidos los servicios, volúmenes a utilizar, puertos que se usarán por cada servicio, o comandos a ejecutar cada vez que se inicia un servicio.

  - `requirements.txt` y `docs-requirements.txt`: Archivos de texto en los que se especifican las dependencias para el contenedor de LETO y para el de su documentación, respectivamente.

- */docs*: Directorio que contiene documentación relativa al proyecto, como archivos CSV con información de ejemplo, o imágenes utilizadas en las páginas de la documentación.

- */leto*: Es el directorio más importante. En él se encuentran todos los principales archivos de LETO. Se explica a continuación este directorio en profundidad.
  - */loaders*: En este directorio se encuentran todos los ficheros necesarios para realizar la carga de datos en la plataforma. Las diferentes formas de realizar esa ingesta de datos se encuentran divididas en 5 archivos.
    * `__init__.py`: Se define la clase `Loader`, la cual será utilizada por todos los otros archivos, para realizar la carga de información. Además, este archivo define la función `get_loaders`, utilizada para obtener todas las clases que pueden ser utilizadas para introducir datos.
    * `dummy.py`: En este archivo se define la clase `ManualLoader`, utilizada para introducir tuplas de forma manual, y la clase `ExampleLoader`, utilizada para realizar una carga de datos de ejemplo en LETO.
    * `structured.py`: Se define la clase `CSVLoader`, utilizada para realizar la carga de datos estructurados representados en un fichero CSV.
    * `unstructured.py`: Se definen las clases `SVOFromText` y `SVOFromFile`, para permitir la introducción de información mediante texto plano o mediante un archivo de texto, respectivamente. Además, se define en este archivo la función `get_svo_tripplets` para, dado un texto, detectar una tripleta de sujeto – verbo – predicado, utilizando un modelo de inteligencia artificial.
    * `wikipedia.py`: Se define la clase `WikipediaLoader` para poder cargar el contenido de una entrada de Wikipedia. Para poder realizar esa carga de información, se definen una serie de funciones auxiliares.
  - */query*: En este directorio se encuentran dos ficheros para procesar las consultas realizadas a LETO. Estos dos ficheros son los siguientes.
    * `__init__.py`: Se definen las clases `Query`, `QueryResolver` y `QueryParser`, utilizadas para analizar la consulta realizada, y traducirla a etiquetas, entidades, relaciones, atributos y etiquetas ignoradas. Toda esa información generada creará la consulta a realizar en la base datos para saber que datos se deben rescatar.
    * `rules.py`: En este caso se define la clase `RuleBasedQueryParser`, la cual sirve de ayuda para anteriores clases a la hora de interpretar la consulta realizada.
  - */storage*: En esta carpeta se definen las opciones que tendrá el sistema para almacenar la información. Los ficheros relevantes que contiene son los siguientes.
    * `__init__.py`: Se define la clase `Storage`, la cual servirá como base a la hora de definir nuevas formas de almacenar la información. Además, cuenta con el

método `get_storages`, el cual devuelve los tipos de almacenamiento implementados.

* `neo4j_storage.py`: Aquí se define el almacenamiento principal utilizado por LETO. Se definen las clases `GraphStorage` y `GraphQueryResolver`, utilizadas para interactuar con la base de datos de Neo4j, así como traducir las consultas introducidas al sistema a la sintaxis admitida por la base de datos.

– */visualization*: Esta carpeta sirve para contener todos los visualizadores de LETO, los cuales fueron explicados anteriormente. El único archivo que contiene es el que se detalla a continuación.

* `__init.py__`: En este archivo se encuentran definidos todos los visualizadores que tiene el sistema, cada uno en una clase diferente. Se define la clase `Visualization`, de la cual siempre se crea un objeto al que se le asignan una serie de propiedades, y se le pasa por parámetro a cada uno de los visualizadores. Además, se define la clase `Visualizer`, la cual contiene el método abstracto `visualize`, para que así todos los visualizadores sean implementados usando esa clase. Por último, en el archivo se encuentra definido el método `get_visualizers`, cuya utilidad es conocer cuáles son los visualizadores implementados.

– `model.py`: En él se definen las clases `Entity` y `Relation`, con sus métodos y propiedades, que servirán de base a la hora de procesar información introducida en el sistema.

– `ui.py`: Este archivo es utilizado para configurar la interfaz gráfica de la aplicación, utilizando funciones de Streamlit.

– `utils.py`: Se definen algunas funciones que serán de utilidad en diversas partes de la aplicación.

- */tests*: En este directorio se encuentran dos archivos utilizados para comprobar el funcionamiento del sistema. Concretamente, se realizan pruebas cargando unos ficheros CSV que se encuentran en */tests/data*, así como la carga de texto plano y la carga de un fichero de texto.

Una vez se conoce lo que contiene la aplicación, se explicará como actualmente se realiza el despliegue de esta.

LETO está pensada para ser desplegada de forma sencilla utilizando Docker. Para permitir este despliegue, se encuentran definidos una serie de archivos que se han explicado anteriormente. Los pasos para comenzar a utilizar LETO serían los siguientes:

1. Estar en el directorio *leto-mvp*.
2. Construir la imagen ejecutando `make image`.
3. Lanzar los contenedores ejecutando `make app`.

Tras esos pasos, en Docker se habrán creado tres contenedores: *leto*, *neo4j* y *leto-docs*, los cuales se pueden visualizar en la Figura 1.3. A continuación, se explican todos ellos:

- *leto*: Este contenedor contiene toda la información sobre el sistema, todos los archivos explicados anteriormente. Al ejecutarse, se levanta la aplicación de Streamlit, por lo que accediendo a `localhost:8501`, se puede visualizar el sitio web. Además, se crean cuatro puntos de montaje para acceder así a los archivos necesarios.

- *neo4j*: Es el contenedor que contiene la base de datos. Concretamente, se crea una base de datos usando Neo4j. En este contenedor también se crean varios puntos de montaje, y también se exponen tres puertos para poder realizar las conexiones con la aplicación.

- *leto-docs*: Este último contenedor contiene un servidor con información sobre LETO.

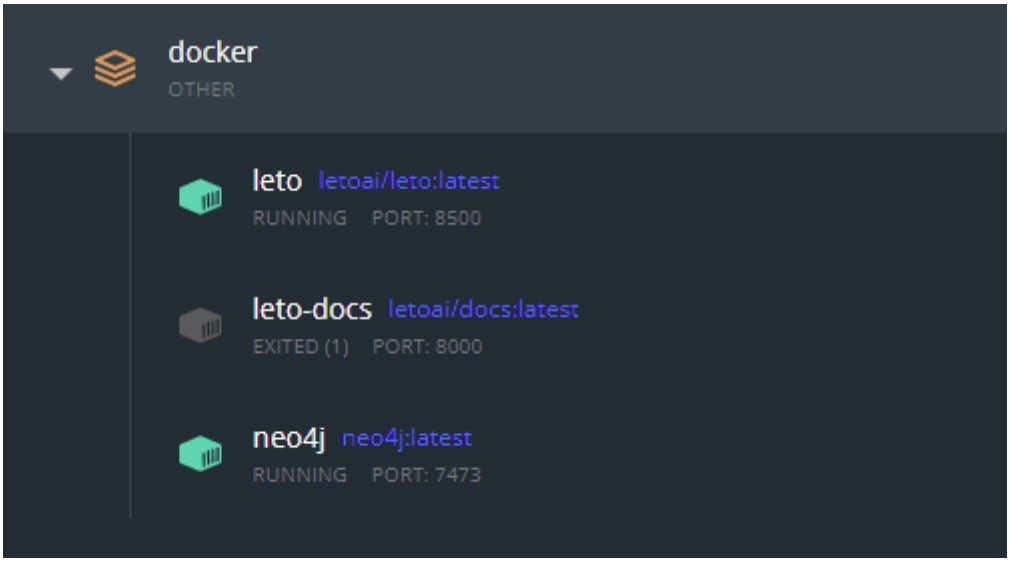

Figura 1.3: Contenedores que se despliegan para utilizar LETO

Finalizada la explicación de toda la plataforma LETO, se explicarán ahora los problemas que presenta.

El principal problema es el acoplamiento. La aplicación no presenta una estructura bien definida, por lo que el código cliente y el código servidor se encuentran muy entrelazados. Esto hace complicado comprender el código actual, en el cual hay líneas de código en las que se usa la librería Streamlit (código generado para el cliente), líneas para realizar conexiones y consultas a la base de datos, o líneas que usan librerías de análisis de datos, como Pandas.

Este problema de acoplamiento y falta de arquitecturización del código deriva en dificultar el mantenimiento de la aplicación, así como hacer complicado extender las funcionalidades que se ofrecen. Una de las necesidades de los investigadores que participan en el desarrollo de este proyecto es la de poder implementar de forma rápida nuevos sensores que utilicen modelos de inteligencia artificial para procesar un conjunto de datos de un campo específico, para poder así enfocar la plataforma a un sector específico, como pueda ser el sector hotelero, y analizar así comentarios de los clientes que realicen en páginas web de reseñas especializadas.

1.2 Objetivos

Una vez comprendida la motivación de la realización de este Trabajo de Fin de Grado, el objetivo principal es el desarrollo de una aplicación web para que los usuarios puedan introducir datos, tanto estructurados como no estructurados, para realizar posteriormente un procesamiento de los mismos y traducir la información a una estructura ontológica. Todo ello desde una aplicación con una estructura bien definida y con un *front-end* y un *back-end* diferenciado.

Se deberá de analizar el código existente para descubrir la arquitectura actual de la aplicación, además de conocer las funcionalidades existentes, errores y mejoras. Por otro lado, también se deberá de realizar una captación de requisitos para conocer si las funcionalidades de la aplicación actual son suficientes, hay que implementar alguna nueva, o algunas de las existentes son innecesarias.

Siguiendo con lo explicado en la motivación de este trabajo, la interfaz deberá de estar pensada para ser usada por todo tipo de personas, sin importar su grado de conocimiento en el ámbito de la informática.

Uno de los propósitos será ir añadiendo sensores que utilicen modelos de inteligencia artificial, para que así el usuario puede seleccionar unos u otros en base a la información introducida. Para ello, la aplicación deberá de estar arquitecturizada de tal forma que se permita de forma fácil ir extendiendo las funcionalidades que ofrece, así como los sensores que se pueden aplicar.

Por último, se deberá de conseguir que el despliegue de la aplicación se realice de la forma más sencilla posible.

1.3 Metodología empleada

El desarrollo de este Trabajo de Fin de Grado se ha realizado en un plazo de tiempo de 5 meses. A continuación, se explicará la metodología seguida durante este periodo de tiempo para el desarrollo del presente trabajo.

En el comienzo de la realización del trabajo, se definieron una serie de hitos a ir consiguiendo, así como una planificación semanal de tareas a realizar. Todo este trabajo se debería de realizar tanto de forma presencial en el Grupo de Procesamiento de Lenguaje Y Sistemas de Información de la Universidad de Alicante, como de forma no presencial.

A continuación, se detallan los grandes hitos definidos:

- *Onboarding* al proyecto: Acceso al GitLab del GPLSI y despliegue e instalación de LETO de forma local.
- Análisis del estado actual: Análisis de la estructura de LETO, lectura y comprensión del código actual, revisión de las funcionalidades implementadas y revisión de la documentación existente.
- Captación de requisitos para la nueva versión de LETO: Análisis de los requisitos necesarios para la plataforma, tanto necesidades de funcionalidades a implementar como necesidades de arquitectura del sistema.

- Realización de una propuesta de mejora: Análisis de toda la información y conocimiento generado hasta el momento. Redacción y presentación de una propuesta de mejora.
- Implementación de la nueva versión de LETO: Desarrollo de los cambios a realizar, así como de las nuevas funcionalidades a implementar.

Este último hito es el más extenso, por lo que se procede ahora a explicar cómo se ha desgranado en tareas más pequeñas para así desarrollar la nueva versión de LETO:

- Planificación de la creación de una API para comunicar el *back-end* con el *front-end*.
- Modificación del código actual para permitir la comunicación via API.
- Creación de los métodos de la API.
- Pruebas sobre los métodos creados utilizando Postman.
- Desarrollo de la nueva web, realizando las pertinentes llamadas a la API.
- Pruebas sobre la nueva web para detectar y solventar errores.
- Implementación de nuevos requisitos, cambiando tanto el *front-end* como el *back-end*.

Para cada uno de los hitos se marcaba una fecha límite de realización. Llegada esa fecha, se presentaban los resultados obtenidos y se pensaba en como encauzar el siguiente hito a realizar.

Se han utilizado GitHub y GitLab como herramientas de control de versiones, Google Meet como herramienta de videollamada, y el correo electrónico y aplicaciones de mensajería instantánea como herramientas de comunicación para el día a día.

2 Marco Teórico

En este capítulo se desarrollará el estado del arte de la web semántica, las ontologías y el descubrimiento de conocimiento automático. Durante el desarrollo de este trabajo será utilizado LETO, un sistema software cuya base teórica se basa en esos tres aspectos detallados.

2.1 Web semántica

La web semántica (también llamada Web 3.0), considerada la sucesora de la red informática mundial (en inglés, World Wide Web (WWW)) y lanzada en 1994, es un conjunto de actividades para la creación de tecnologías para publicar datos legibles por máquinas. Impulsada por el World Wide Web Consortium (W3C) y por Tim Berners-Lee (creador de la WWW), la web semántica se impulsó para que así la web no sea únicamente un conjunto de hipertextos, sino que además se pudieran añadir metadatos semánticos y ontológicos.

La web antigua (entendiendo ese concepto como la web que existía previa a la web semántica), se basaba en el lenguaje natural, carecía de una estructura clara de sus contenidos y de una normalización de las descripciones a realizar para los diferentes recursos. Por eso, la mayoría de los contenidos de la web se distribuían de forma desestructurada. Todo ello derivaba en problemas para los motores de búsqueda, al encontrar estas ambigüedades en los resultados que ofrecían a los usuarios.

Gracias a que esos metadatos son legibles por máquinas y es posible interconectarlos entre sí, es posible realizar una optimización en el intercambio de información en la web, identificando de forma unívoca la información y pudiendo establecer relaciones entre diferentes entidades contenidas por la información.

La definición de los conceptos en la web semántica se basa en tres pilares: La semántica, los metadatos y las ontologías.

- Semántica: Es el significado de los términos lingüísticos.
- Metadatos: Datos que describen a un conjunto de datos.
- Ontologías: Conjuntos ordenados de información legibles por humanos o por máquinas.

En la actualidad, son varios los retos a los que tiene que hacer frente la web semántica. El W3C contó hasta 2008 con un grupo de trabajo cuya misión era la de definir de la mejor forma posible como razonar y representar la información incierta que se encuentra en la web. Dos de los retos definidos por este grupo y que los sistemas de razonamiento automático que se están desarrollando en la actualidad van a tener que solventar son los siguientes:

- Volumen de contenido: Los sistemas de razonamiento automático deberán de ser capaces de hacer frente a grandes cantidades de información, al tener la web miles de millones de páginas a procesar.

- Conceptos inciertos: Algunos conceptos pueden resultar imprecisos para los sistemas de razonamiento automático. ¿Qué es claro y que oscuro? ¿Qué es amplio y que es estrecho? Se deben de desarrollar técnicas para poder diluir la confusión que pueden inducir esos términos.

2.2 Ontología

Una ontología define una serie de primitivas de representación con las que modelar un dominio de conocimiento para permitir a las máquinas poder leer la información representada. Esas primitivas de representación suelen ser clases, atributos y relaciones, mediante las cuales se puede representar información y su significado.

Desde la década de los 80, los investigadores en el campo de la inteligencia artificial adoptaron el término ontología, al considerar que su uso permitía realizar modelos computacionales para poder realizar cierto comportamiento automático. Ya en la década de los 90, con el objetivo de poder crear normas que permitieran la interoperabilidad de los sistemas, se definió una pila en la que la capa ontológica era un componente uniforme de los sistemas de conocimiento.

En 1993, Thomas R. Gruber en un artículo en el que explicaba los principios de diseño de ontologías para el intercambio de conocimiento, definía una otología como la "especificación explícita de una conceptualización" que son "los objetos, conceptos y otras entidades que se supone que existen en algún área de interés y la relaciones que se dan entre ellos".

El W3C, en 2004, publicó un artículo en el que describía el uso del Lenguaje de Ontologías Web (en inglés, *Web Ontology Language*, OWL). Ese lenguaje está diseñado para ser usado por aplicaciones que necesiten procesar información, en lugar de mostrar esa información a los usuarios. Aporta un vocabulario con una semántica formal, por lo que las máquinas pueden entender de forma sencilla la información representada.

Una de las ventajas del uso de ontologías con respecto al uso de bases de datos, es el poder especificar una representación del modelo de datos a un nivel de abstracción superior a los diseños, lógicos o físicos, de las bases de datos. Esto permite que los datos representados puedan consultarse o exportarse entre sistemas desarrollados de forma independiente. Las ontologías se pueden dividir de acuerdo con su nivel de generalidad en cuatro tipos diferentes, los cuales se detallan a continuación.

- Ontologías de alto nivel: Describen conceptos generales, los cuales son independientes a un dominio o problema concreto.

- Ontologías de dominio: Describen el vocabulario relacionado con un dominio en específico mediante la especialización de conceptos introducidos en ontologías de alto nivel.

- Ontologías de tareas: Describen el vocabulario referente a una tarea o actividad relacionada con la resolución de problemas. Estas ontologías pueden pertenecer a un mismo dominio o a diferentes dominios.

- Ontologías de aplicación: Describen conceptos pertenecientes a un determinado dominio y a una determinada tarea. Estas ontologías suelen referirse a los roles que las entidades representadas tienen al realizar una tarea.

En el campo de la inteligencia artificial y la web semántica, el uso de las ontologías aporta sendas ventajas, como el hecho que clarificar la estructura de conocimiento al definir los conceptos del dominio al que se dirige, la reducción de la ambigüedad conceptual al aportar una sintaxis común, o por último permitir compartir conocimiento al estar la información estructurada usando una serie de términos definidos y una sintaxis concreta.

2.3 Descubrimiento de conocimiento automático

El descubrimiento de conocimiento automático es el proceso por el cual se extrae información útil de un conjunto de datos. Este proceso es fundamental para que la web semántica pueda funcionar, ya que se necesita un proceso mediante el cual se pueda extraer información de los sitios web de forma eficiente. Quizás se puede pensar que ese proceso se puede resolver utilizando, por ejemplo, la notación de HTML, pero en la realidad, esas anotaciones no están completas o con el nivel de detalle necesario. Por otro lado, las anotaciones o comentarios que se puedan realizar son complicadas de procesar, por lo que se tardaría demasiado tiempo en utilizarlas. Por ello, es necesario contar con sistemas que sean capaces de extraer información de diversas fuentes, tanto de páginas web como de fuentes de datos estructuradas o no estructuradas. Se podría considerar un archivo CSV una fuente de información estructurada, y un fichero de texto plano una fuente de información no estructurada. Uno de los campos en los que más se aplica el descubrimiento automático es en las bases de datos, el llamado en inglés Knowledge Discovery in Databases (KDD). Consiste en un proceso automático para explorar, analizar y modelar grandes repositorios de datos, extrayendo patrones en forma de reglas o funciones a partir de los datos para que el usuario los analice. Ese proceso se podría dividir en cinco etapas: La selección de los datos, el procesamiento y limpieza de esos datos, la transformación y reducción, la minería de datos, y por último la interpretación y evaluación de los resultados obtenidos.

3 Tecnologías empleadas

En este capítulo se explicarán una serie de tecnologías que se han empleado. Se explicarán tanto herramientas de trabajo como tecnologías utilizadas para el desarrollo.

3.1 Python

Python es un lenguaje de programación de alto nivel, interpretado y orientado a objetos. Diseñado por Guido van Rossum, fue publicado en 1991, y 31 años después, su última versión estable es la 3.10.4. Este lenguaje es administrado por la Python Software Foundation y posee una licencia de código abierto llamada Python Software Foundation License, lo que permite distribuir el lenguaje gratuitamente. Es un lenguaje multiparadigma, permitiendo la programación orientada a objetos, la programación imperativa, y la programación funcional. Utiliza un tipado dinámico y un conteo de referencias para la gestión de memoria. Soporta módulos y paquetes, lo que permite modularizar el código y fomentar la reusabilidad. Todas estas características han hecho que este lenguaje sea muy popular, siendo según GitHub el más popular en el mundo.

3.2 API

Una interfaz de programación de aplicaciones (conocida como API por sus siglas en inglés) se puede definir como un conjunto de definiciones y protocolos que permiten la comunicación entre diferentes sistemas, como pueda ser una aplicación y un servidor. Mediante el uso de APIs se puede conseguir una abstracción entre diferentes capas en los que se encuentre estructurado el código de una aplicación. En la actualidad, muchas aplicaciones populares como Google Maps o Telegram ofrecen APIs para comunicarse son sus servicios, y poder así integrarse en sitios de terceros. Se pueden definir cuatro tipos de API: SOAP, XML-RPC, JSON-RPC y REST.

- API SOAP: Nombre dado de las siglas en inglés de Simple Object Access Protocol, es un protocolo de intercambio de información y datos en XML. SOAP define su estructura de mensajes y métodos en WSDL, un lenguaje basado en XML.
- API XML-RPC: Es el protocolo más antiguo y simple. Utiliza XML como formato de datos y llamadas HTTP como sistema de comunicación.
- API JSON-RPC: Es el mismo protocolo que el anterior, solo que, en lugar de utilizar XML como formato de datos, los formatea en JSON.
- API REST: La transferencia de estado representacional (REST, de la siglas en inglés), se utiliza junto a HTTP para obtener datos en cualquier formato. Algunas de las características de REST es que es un protocolo cliente/servidor sin estado, cuenta con un

conjunto de operaciones (también llamados verbos) bien definidas, tiene una sintaxis universal al identificar los recursos por una URI, y se permite el uso de hipermedios (como HTML o XML).

3.3 FastAPI

FastAPI es un framework web utilizado para el desarrollo de APIs RESTfull en Python. Permite la creación de APIs de manera rápida y fácil, con un alto rendimiento en tiempos de procesamiento de peticiones. Además, genera de forma automática documentación de los métodos implementados. Se encuentra basasda en Pydantic, librería que permite la validación de datos y la gestión de configuraciones utilizando anotaciones. Algunas de las características por las que destaca este framework son:

- Rapidez y facilidad de desarrollo.
- Rapidez en la ejecución, con métricas de rendimiento similares a las de lenguajes de programación como NodeJS o Go.
- Fácil de usar, teniendo una sintaxis muy intuitiva.
- Flexibilidad a la hora de determinar los parámetros de los métodos, lo que reduce la duplicación de código y posibles errores.

3.4 Streamlit

Streamlit es un framework gratuito de código abierto escrito en Python. Es utilizado frecuentemente para la creación de sitios web de manera rápida en los campos de la ciencia de datos y la inteligencia artificial, al incorporar métodos muy usados, como pueda ser la representación de gráficas en base a información almacenada en un CSV. Es compatible con la mayoría de librerías de Python, como scikit-learn, Keras, PyTorch o Pandas. Las páginas web con Streamlit se desarrollan del mismo modo que se escribe código en Python. Se utilizan una serie de directivas proporcionadas por el framework, y Streamlit se encarga de ir actualizando la web conforme se van produciendo cambios en el código. Cuenta con un extenso catálogo de widgets y de layouts que permiten personalizar la aplicación lo máximo posible.

3.5 Neo4j

Neo4j es un software desarrollado en Java utilizado para la creación de bases de datos orientadas a grafos. Cuenta con una licencia comercial y con otra de software libre (GNU Affero General Public License). Neo4j utiliza grafos para representar y almacenar datos, así como para relacionar estos datos. De todos los tipos de grafos existentes, utiliza grafos de propiedad, los cuales tienen peso y etiquetas, y en los que es posible asignar propiedades a nodos y a relaciones. Tres de las principales ventajas que ofrece son las siguientes:

- Rendimiento: Las bases de datos orientadas a grafos tienen mejor rendimiento que las relacionales o no relacionales. Esto es útil cuando se manejan grandes volúmenes de datos, al no descender el rendimiento con esas cantidades de datos.

- Agilidad: La agilidad que otorga Neo4j es uno de sus puntos fuertes. Los desarrolladores aseguran que puede soportar más de 34.000 millones de datos almacenados sin que esto suponga un problema.
- Flexibilidad y escalabilidad: Las posibilidades que ofrece el sistema en estos aspectos son enormes gracias a estar trabajando con una estructura de base de datos de grafos.

3.6 Docker

Docker es un software libre de código abierto utilizado para automatizar el despliegue de aplicaciones dentro de contenedores, proporcionando una capa de abstracción adicional, así como automatizar la virtualización de aplicaciones en diferentes sistemas operativos. En los contenedores de Docker se pueden crear, probar e implementar aplicaciones de forma rápida y sencilla. Cada contenedor incluye todo lo necesario para que el software se pueda ejecutar (sistema operativo, bibliotecas, código de la aplicación…). Algunos beneficios de utilizar contenedores de Docker son los siguientes:

- Rápido despliegue: Con Docker se puede desplegar una aplicación en pocos minutos, al estar toda la configuración necesaria planificada.
- Aislamiento: Cada contenedor tendrá sus datos aislados del resto de contenedores, así como poder asignarles sus propios recursos de computación.
- Seguridad: Gracias al aislamiento comentado en el punto anterior, se consigue cierto nivel de seguridad. Un contenedor de Docker no puede conocer los procesos ejecutado en otro contenedor.
- Compatibilidad: Los problemas de compatibilidad entre diferentes ordenadores desaparecen al tener configurado un contenedor, el cual se despliega exactamente igual en todos los dispositivos, al tener de forma asilada todas las dependencias necesarias.

3.7 Postman

Postman es una plataforma para diseñar, construir y probar APIs. Originalmente era una extensión del navegador Google Chrome, pero en la actualidad dispone de aplicaciones para diversos sistemas operativos, así como una versión web. Las principales funcionalidades que ofrece Postman son las siguientes:

- Permite la creación de peticiones a APIs propias o de terceros.
- Permita elaborar pruebas para comprobar el correcto funcionamiento de una API.
- Ofrece la posibilidad de trabajar colaborativamente gracias a los entornos de trabajo.
- Se pueden organizar las peticiones en carpetas, funcionalidades y módulos.
- Permite generar documentación de una API.

3.8 Visual Studio Code

Visual Studio Code (también llamado VSCode) es un editor de código fuente creado por Microsoft. Dispone de versiones para Windows, Linux, macOS y un editor web. Lanzado en 2015, es gratuito y de código abierto. Está creado usando el framework Electron. Permite depurar el código, instalar extensiones gracias a su amplio catálogo, autocompletar el código (una función llamada IntelliSense), integrarse con repositorios de Git… Además, es completamente personalizable, pudiendo cambiar así los colores del programa, el estilo del texto o los atajos de teclado. Sus funciones soportan la mayoría de lenguajes de programación existentes, siendo la función de resaltado de sintaxis la que más lenguajes soporta.

4 Desarrollo

En este capítulo se explicará todo el trabajo desarrollado, así como lo conseguido, en base a los hitos de trabajo definidos anteriormente. Se dividirá el capítulo en las siguientes secciones:

- Propuesta de nueva aplicación: En esta sección se explicará todo el proceso seguido a la hora de realizar una nueva propuesta de aplicación. Se expondrá el análisis de funcionalidades que LETO ya tiene implementadas, así como el análisis de requisitos realizado para que sean implementados en la nueva versión. Por otro lado, se analizará la arquitectura propuesta para reestructurar el código actual.

- Desarrollo de la nueva aplicación: Una vez se conoce la propuesta realizada, se explicará detalladamente cómo se ha llevado a cabo la implementación de todo lo explicado en ella. Por un lado, de expondrán los cambios realizados sobre el sistema actual, así como el nuevo trabajo realizado. Se detallarán las herramientas y tecnologías empleadas para el desarrollo.

4.1 Propuesta de nueva aplicación

En este apartado se desarrollará la propuesta de la nueva aplicación a crear contando como base el estado actual de LETO. Se comenzará hablando de los requisitos de la aplicación.

Una parte fundamental en la Ingeniería de Software es la captación de requisitos. Los requisitos son las condiciones que el producto software a desarrollar debe de cumplir o satisfacer. Esos requisitos vendrán marcados por los clientes o usuarios finales de la aplicación. La captación de requisitos es el proceso mediante el cual se descubren los requisitos que necesitará el producto a desarrollar.

Esta captación se puede realizar de diferentes formas. Se puede, por ejemplo, realizar una lluvia de ideas en la que estén presentes diferentes tipos de participantes para así captar ideas, seleccionarlas y transformarlas a requisitos. Otra forma de realizar la captación de requisitos puede ser realizando prototipos y mostrándoselos a los usuarios, para así conocer si satisfacen sus necesidades y poder comenzar a desarrollar en base a esos prototipos realizados.

En este caso, para realizar la labor de la captación de requisitos se realizaron diversas reuniones en las que estuvieron presentes varios investigadores del GPLSI. En base a esas reuniones, se identificaron principalmente cuatro requisitos que actualmente LETO implementa, los cuales se detallan a continuación:

- Ingesta de datos: Se pueden introducir datos a la plataforma procedentes de diferentes fuentes, tales como archivos CSV, TXT, páginas de Wikipedia o texto plano.

- Gestión de los datos: Se permite visualizar el número de elementos almacenados en la base de datos, así como eliminarlos todos.

- Consultas: Los usuarios pueden realizar consultas, las cuales son procesadas para devolver una serie de datos visualizados de diferentes formas (datos representados en un mapa, histogramas, grafos…). Se muestran a los usuarios consultas de ejemplo.
- Despliegue: La aplicación está estructurada en una serie de contenedores para facilitar el despliegue.

Por otro lado, una vez conocidos los requisitos ya implementados, se analizaron los requisitos necesarios para la aplicación y que aún no se encuentran desarrollados. Se detallan a continuación esos requisitos a implementar:

- Arquitectura: Se debe de realizar un *front-end* desacoplado del *back-end*. Para ello, se debe de implementar una API para realizar el intercambio de datos.
- Documentación API: Se debe de generar una documentación de la API implementada para que futuros desarrolladores conozcan como consumir los métodos de esa API.
- Sensores: Se debe de arquitecturizar el código para permitir añadir de forma sencilla sensores que procesen la información introducida.

Todos estos requisitos serán analizados para conocer como técnicamente se pueden aplicar.

En cuanto a la arquitectura propuesta, se plantea una arquitectura de tres capas. La programación por capas es una arquitectura cliente-servidor cuyo objetivo principal es realizar una separación lógica de los componentes software que constituyen un sistema. Al tener varias capas perfectamente divididas, el desarrollo de cada una de ellas se puede llevar a cabo de forma independiente, y los cambios que se produzcan en una capa no producirán un gran impacto en toda la aplicación. El diseño de programación por capas más utilizado es el diseño en tres capas, siendo esas capas la presentación, la de aplicación (también llamada capa de negocio) y la de datos.

Las tres capas propuestas para esta arquitectura son las siguientes:

- Capa de presentación: Será el *front-end* de la aplicación, la parte visual de la misma. El usuario interactuará con esta capa para introducir datos, visualizarlos… Esta capa se desarrollará sin ninguna base, utilizando HTML, CSS y JavaScript. Además, esta capa realizará llamadas a la API.
- Capa de aplicación: Esta capa será el *back-end* de la aplicación, lugar en el que se procesará la información.
- Capa de datos: Se considerará que esta capa es la base de datos. Esta base de datos está montada usando Neo4j, y se encuentra en un contendedor.

En base a esas tres capas, se estructurará el código existente para así permitir también incorporar código nuevo.

En cuanto a la API a desarrollar, se explicará a continuación la propuesta de API en base a las necesidades requeridas. Una de las motivaciones de la creación de esta API es el desacoplamiento del código actual de la plataforma. Mediante esta propuesta, la capa de presentación estará perfectamente separada del resto de capas. En un futuro, la API puede

aportar beneficios a LETO en el caso de que quiera desarrollarse, por ejemplo, una aplicación móvil para visualizar los datos que contiene la plataforma, o se quiera desplegar la capa de presentación en un lugar diferente al resto de capas.

Además, el hecho de exponer la información a través de una API, hará que en caso de necesitar añadir, por ejemplo, un nuevo método de entrada de información o modificar los parámetros que reciba algún endpoint, sea una tarea fácil.

Para las cargas de información en la plataforma, será necesario un endpoint para cada uno de los tipos de entrada admitidos. Para la gestión de la información de la base de datos, se pretende exponer un endpoint para consultar el número de datos que están almacenados, así como uno para limpiar la información guardada. Por último, se creará un endpoint para comprobar que la API está funcionando correctamente, uno para realizar las consultas de información y otro para obtener la lista de sensores implementados.

En total, se deberán de crear 11 endpoints, cada uno con sus correspondientes parámetros.

Otro de los requisitos solicitados es la interoperabilidad de la antigua y la nueva plataforma. Para ello, se deberán de realizar modificaciones e inclusiones de código sin que esto haga que el código existente deje de funcionar.

4.2 Desarrollo de la nueva aplicación

Una vez realizada la captación de requisitos para la aplicación, se procede a explicar cómo se han llevado a la realidad todos esos requisitos. Para ello, se explicarán paso a paso las acciones realizadas, siguiendo los hitos marcados y explicados en la metodología de este trabajo.

Vistos los requisitos que se requieren de la API, se desarrollaron los endpoints necesarios para cumplirlos todos. Estos endpoints creados son:

- */*: Método GET que se utilizará para comprobar que la API está funcionando correctamente.
- */load/csv*: Método POST para introducir la información en un CSV. En el cuerpo de la petición se introducirá el fichero, se especificarán los números de sensores, el lenguaje de la información, y cuál es la entidad principal.
- */load/wikipedia*: Método POST para especificar una página de Wikipedia y cargar información. En el cuerpo de la petición se especificará la consulta a realizar a Wikipedia y el lenguaje de la información a incorporar.
- */load/examples*: Método POST para incorporar información de ejemplo.
- */load/tuples*: Método para incorporar tuplas de forma manual. La información de la tupla se pasará como parámetro de la petición.
- */load/txt*: Método POST para incorporar información desde un fichero de texto. En el cuerpo de la petición se incorporará el archivo de texto, así como especificar el lenguaje del texto.
- */load/text*: Método POST para incorporar información en base a un texto introducido. Ese texto se especificará en el cuerpo de la petición, así como el lenguaje en el que está escrito.

- */db/size*: Método GET para conocer el número de datos almacenados en la base de datos.
- */db/clear*: Método GET para limpiar la base de datos.
- */query*: Método POST para especificar una consulta a LETO. Esa consulta se especificará por parámetro en la petición.
- */sensors*: Método GET para obtener una lista con los sensores que se encuentran implementados en la aplicación.

Toda la API se ha implementado utilizando FastAPI. En el Código 4.1 se muestra uno de los métodos implementados, concretamente el método para cargar ficheros CSV.

Código 4.1: Método para la carga de ficheros CSV a través de la API

```
# LOAD CSV
@app.post("/load/csv")
async def load_CSV(file: UploadFile, sensors: Optional[str] = "0", language: Optional[str] = 'es', ↩
    ↪ main_entity: Optional[str] = ''):
    if sensors == "":
        raise HTTPException(
            status_code=400,
            detail="Incorrect sensors",
            headers={"X-Error": "The sensors ID must be between 0 and " + str(Sensor.sensorsList().↩
                ↪ __len__()-1)},
        )
    else:
        with open("file.csv", "wb") as buffer:
            shutil.copyfileobj(file.file, buffer)

        factsAdded = load(CSVLoader(paths=["file.csv"], sensors = sensors, language=getattr(Language, ↩
            ↪ language), main_entity = main_entity))

        return {"filename": file.filename, "numFactsAdded" : factsAdded, "size" : storage.size}
```

Se puede observar como el método se encuentra anotado con `@app.post("/load/csv")`. La anotación indica la URL a la que se deberá de enviar la petición, así como el verbo de esa petición, que en este caso será GET.

La API se expondrá mediante el puerto 8500. Para poder exponerla en ese puerto, se debe de ejecutar el comando `uvicorn api:app --host 0.0.0.0 --port 8500`. En ese comando de Uvicorn, se especifica el nombre del fichero donde está la API ("api" en este caso), el lugar dónde se desplegará (`localhost` en este caso, especificado como `0.0.0.0`), y por último el puerto en el cual se atenderán las peticiones.

Para la ejecución fácil de la API, se ha creado un pequeño script `bash`, el cual se encuentra ubicado en la misma carpeta de la API.

Una vez acabada la implementación de la API, se creó un proyecto en Postman para poder probar todos los métodos implementados de una sencilla forma. La estructura de carpetas creada es la siguiente.

- Carpeta *DATA*: Contiene todas las peticiones relacionadas con la ingesta de datos de la plataforma.

- Carpeta *DATABASE*: Contiene los métodos para consultar el número de elementos almacenados en la base de datos y para eliminar todos los datos.

En la raíz del proyecto se encuentran los métodos para comprobar que la API está activa, realizar una consulta a LETO, y obtener una lista de los sensores implementados. En la Figura 4.1 se muestra la estructura de carpetas explicada.

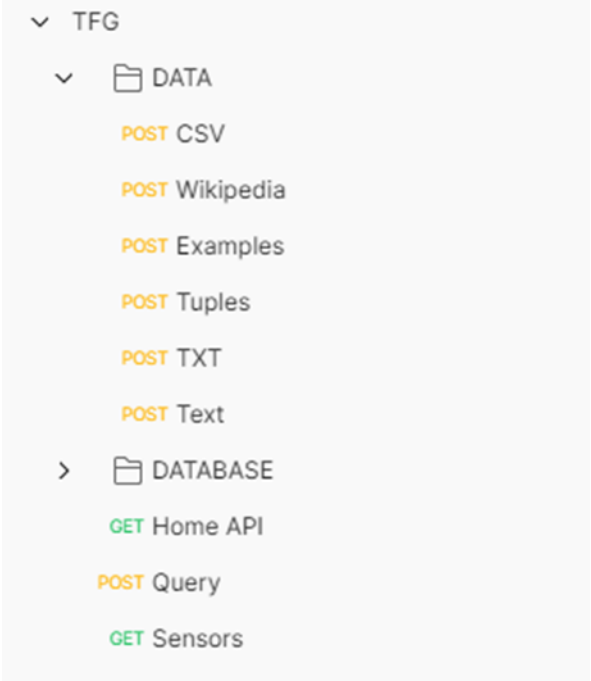

Figura 4.1: Estructura de carpetas en Postman

En la Figura 4.2 se muestra la petición creada para enviar un archivo CSV a la API.

Se especifica en la petición el verbo, la URL, así como todos los elementos del cuerpo necesarios. Una vez toda esa información está completada, se puede probar a enviar la petición, recibiendo una respuesta por parte de la API, la cual se muestra en la Figura 4.3.

Una vez finalizada la API, se debieron de realizar cambios en el código existente, con el objetivo de mantener la anterior versión (la montada sobre Streamlit) y la nueva interoperables.

El primer cambio realizado ha sido en la parte de los archivos que realizan la carga de la información en LETO (los archivos que se encuentran en la carpeta `loaders`). La modificación realizada ha sido sobre el archivo que carga información de forma estructurada, es decir, archivos CSV.

Esa clase contenía, además del código necesario para procesar el fichero y la creación básica de las entidades, el código en el que se aplicaba el único sensor que tenía implementado. Al ser uno de los requerimientos para la nueva aplicación el hecho de poder implementar de forma fácil nuevos sensores, estos no podrían estar acoplados en la misma clase del resto del código de ingesta de datos a través de un CSV.

Es por ello por lo que se ha eliminado el código de esa clase, y se ha creado una nueva,

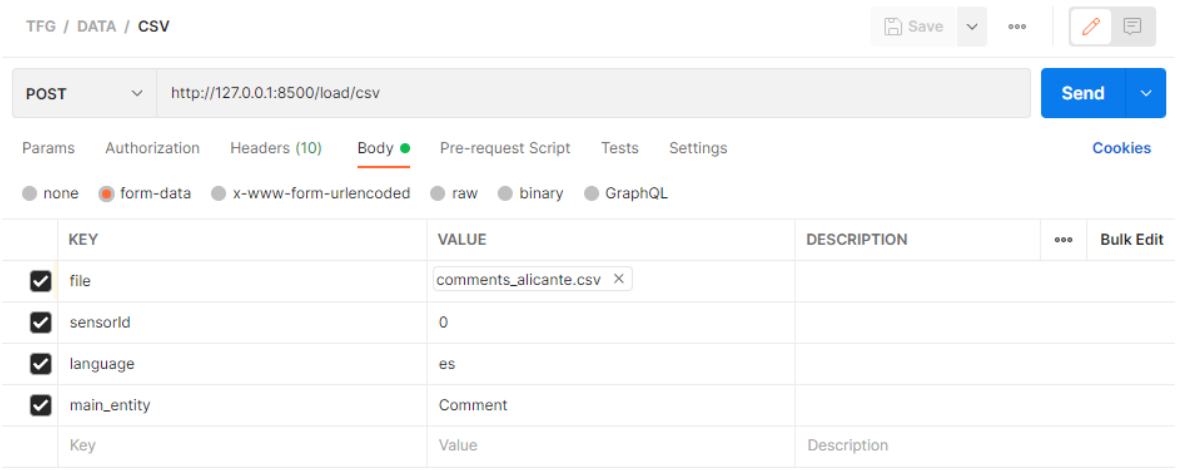

Figura 4.2: Captura de la petición POST desde Postman para incorporar un CSV a LETO

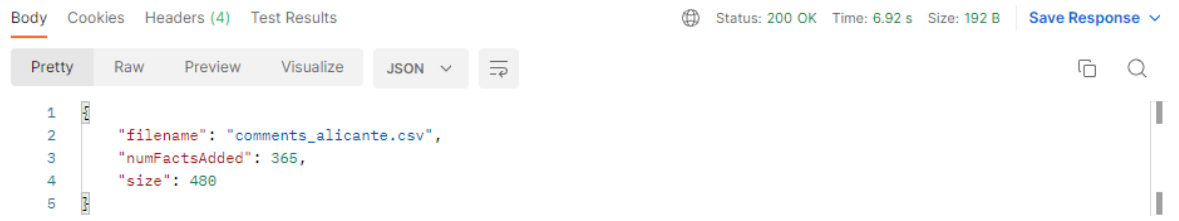

Figura 4.3: Captura de la respuesta a la petición POST desde Postman para incorporar un CSV a LETO

llamada `Sensor`. Esta clase contiene una función para devolver una lista de los sensores implementados, otra para aplicar un determinado sensor (este método será el que llame la clase que procesa los CSV), y, por último, una función por cada sensor implementado.

Se muestra en el Código 4.2 del único sensor que implementa la aplicación, el cual utiliza dos modelos de inteligencia artificial (uno para textos en castellano y otro para textos en inglés) para la detección de entidades.

Código 4.2: Código del sensor *sensorNER*

```
def sensorNER(self, tupl, column_types, main):
        text_columns = [c for c, t in column_types.items() if t == "text"]
        nlp = get_model(self.language) if text_columns else None

        for c in text_columns:
            text = tupl[c]
            entities = [
                Entity(e.text.strip(), e.label_) for e in nlp(text).ents
            ]

            for e in entities:
                if e.type == "LOC":
                    e.type = "Location"
                elif e.type == "PER":
                    e.type = "Person"
                elif e.type == "ORG":
```

```
                    e.type = "Organization"
                else:
                    continue

                yield Relation(
                    entity_from=main, entity_to=e, label="mention", field=c
                )
```

Una vez implementada esa nueva clase, se modificó el código de la clase de lectura de archivos CSV. En el Código 4.3 se muestran las líneas de código que llaman a la función de aplicar el sensor.

Código 4.3: Líneas de código para llamar a los sensores

```
for it in Sensor.applySensor(self, tupl, column_types, main, self.sensors):
                    yield it
```

Otra parte del código donde se han introducido nuevas funciones es en la parte de visualización de datos. LETO implementa una clase base para cada visualizador, así como una clase por cada tipo de visualizador.

Para permitir mostrar esos visualizadores en el nuevo *front-end*, se ha optado por devolver en la respuesta a la petición de la API, el código HTML de cada uno de ellos, para así insertar directamente ese código en la web. Para poder realizar eso, se ha editado esa clase base que luego usan todos los visualizadores, para añadir una nueva función llamada `visualizeAPI`, función a la que llamará el código de la API y que devolverá un HTML representando un visualizador concreto.

En el Código 4.4 se muestra la clase base llamada `Visualization`, la cual contiene el método `visualizeAPI`.

Código 4.4: Clase Visualization

```
class Visualization:
    def __init__(self, title: str, score: float, run: Callable) -> None:
        self.score = score
        self.title = title
        self.run = run

    def visualize(self):
        with st.expander(self.title, self.score > 0):
            self.run()

    def visualizeAPI(self):
        with st.expander(self.title, self.score > 0):
            return self.run()

    def valid(self) -> bool:
        return True

    class Empty:
        score = 0
        title = None

        def valid(self) -> bool:
            return False
```

En cada visualizador, mucha parte del código será compartida tanto para visualizar la

información mediante la web antigua como desde la web nueva. Es por ello por lo que todo ese código compartido se ha extraído a funciones llamadas `visualizationCommon`.

En el Código 4.5 se muestra un ejemplo de función `visualizeAPI`, en este caso la del sensor que genera un mapa con localizaciones.

Código 4.5: Método visualizeAPI de la clase MapVisualizer

```
def visualizeAPI(self, query: Query, response: List[Relation])−> Visualization:
        countries = []
        locations = []

        for tuple in response:
            for e in [tuple.entity_from, tuple.entity_to]:
                if not (
                    query.mentions(entity=e.name)
                    or query.mentions(relation=tuple.label)
                ):
                    continue

                if e.name in query.entities and e.name in self.visualizables:
                    countries.append(e)

                if "lat" in e.attrs and "lon" in e.attrs:
                    locations.append(e)

        if not countries + locations:
            return Visualization.Empty()

        regions = set(d.name for d in countries)

        def visualization():
            map = self.visualizationCommon(countries, locations, regions)
            map.to_html(filename="/home/coder/leto/data/map.html", notebook_display=True, ↩
                ↪ iframe_width='50%', iframe_height=100)
            return {"type": "map", "content": open("/home/coder/leto/data/map.html").read()}

        return Visualization(
            title="  Map",
            score=len(countries + locations) / len(response),
            run=visualization,
        )
```

En la función `visualization` que contiene se puede observar cómo se devuelve un JSON especificando el tipo de visualizador y su código HTML.

Por último, este archivo que contiene todos los visualizadores contiene una función para devolver una lista de todos ellos. Se muestra en el Código 4.6.

Código 4.6: Método get_visualizers

```
def get_visualizers():
    return [
        GraphVisualizer(),
        MapVisualizer(),
        CountEntitiesVisualizer(),
        SchemaVisualizer(),
        AttributeVisualizer(),
        TimeseriesVisualizer(),
    ]
```

Acabados de ver los cambios sobre los visualizadores, se comenta un pequeño cambio realizado en el archivo que contiene la parte visual de la antigua web (archivo `ui.py`, montado principalmente con funciones de Streamlit). Una de las funciones que contiene, llamada `_build_cls`, es utilizada por los archivos que realizan la carga de datos en la aplicación para detectar los argumentos pasados. Se ha tenido que crear un nuevo caso, la introducción de un tipo de datos de ficheros que utiliza FastAPI.

Una vez realizados todos los cambios sobre el código antiguo de LETO, así como haber creado nuevas partes para implementar la API, se puede dar por finalizada esta parte del trabajo.

Como se ha comentado anteriormente, LETO se encuentra montado sobre una estructura de contenedores de Docker. Sobre los archivos que definen esos contenedores se han realizado una serie de pequeños cambios. Esos cambios han sido:

- En el archivo en el que se especifican todas las dependencias a instalar, se ha incluido la de FastAPI.
- En el archivo del `docker-compose`, se ha añadido un puerto al contenedor que despliega LETO para poder utilizar la API a través de él.
- En el `dockerfile` del contenedor de documentación se ha eliminado la directiva que hacía que todas las dependencias se descargaran desde un servidor de la Universidad de las Ciencias Informáticas de La Habana (Cuba), el cual no respondía.

Una vez realizados esos cambios, los contenedores se pueden desplegar fácilmente utilizando el `makefile` que contiene el paquete.

5 Conclusiones

En el presente Trabajo de Fin de Grado, a partir de LETO se ha conseguido tener un sistema con un *front-end* claramente diferenciado del resto de la aplicación, pudiéndose desplegar sin ningún tipo de problema en entornos diferentes. Además, mediante la implementación de la nueva API, se consiguen unas mayores ventajas para la plataforma, al conseguir una mayor flexibilidad que antes no se tenía debido a que el código estaba fuertemente acoplado al usar las funciones de Streamlit.

Esas posibilidades de ventaja en cuanto al funcionamiento y versatilidad que ahora ofrece la plataforma son el principal logro conseguido en este trabajo, resolviendo el gran acoplamiento que tenía la plataforma LETO. Al resolver el problema del acoplamiento, el nuevo sistema facilitará el uso e implementación en esta arquitectura de nuevos sensores para aplicar LETO a nuevos dominios, haciendo independientes el desarrollo de los sensores de su utilización en la plataforma.

Por lo tanto, todos los objetivos enumerados en este Trabajo de Fin de Grado, han sido satisfechos con éxito.

5.1 Trabajo futuro

Una vez vista la conclusión realizada, así como el trabajo realizado, se pueden plantear una serie de aspectos sobre LETO en los que se podría trabajar en el futuro.

Llegados a este punto, LETO cuenta con una API mediante la cual es posible exponer todos los métodos que los desarrolladores consideren. Esos métodos actualmente son consumidos por la página web creada. Pero podrían no ser consumidos solo por esa web. En un futuro, se podría plantear el desarrollo de una aplicación móvil para realizar consultas a LETO, y visualizar así todo el conocimiento que tenga almacenada la plataforma de forma rápida desde cualquier dispositivo móvil.

Por otro lado, se explicó al inicio del trabajo que uno de los objetivos era que LETO pudiera implementar de forma rápida nuevos sensores que utilizaran modelos de inteligencia artificial para poder así procesar los datos introducidos. Esos sensores se podrían especializar en un dominio concreto, para poder ofrecer esta plataforma a sectores específicos de la economía, como pudiera ser el sector del turismo. Siguiendo con el ejemplo del turismo, se podrían extraer y analizar datos provenientes de páginas de reseñas o de comentarios de clientes de hoteles o restaurantes, para poder observar y relacionar todos los comentarios y sacar conclusiones. Incluso, se podrían implementar nuevos visualizadores para, en base a los datos almacenados, predecir comportamientos futuros de los usuarios.

Bibliografía

Estevez-Velarde, S., Gutiérrez, Y., Almeida-Cruz, Y., y Montoyo, A. (2021). General-purpose hierarchical optimisation of machine learning pipelines with grammatical evolution. *Information Sciences*, *543*, 58–71.

Estevez-Velarde, S., Gutiérrez, Y., Montoyo, A., Piad-Morffis, A., Muñoz, R., y Almeida-Cruz, Y. (2018). Gathering object interactions as semantic knowledge. En *Proceedings on the international conference on artificial intelligence (icai)* (pp. 363–369).

Estevez-Velarde, S., Montoyo, A., Almeida-Cruz, Y., Gutierrez, Y., Piad-Morffis, A., y Muñoz, R. (2019, Septiembre). Demo application for leto: Learning engine through ontologies. En *Proceedings of the international conference on recent advances in natural language processing (ranlp 2019)* (pp. 276–284).

Piad-Morffis, A., Gutierrez, Y., Estevez-Velarde, S., y Muñoz, R. (2019, Junio). A general-purpose annotation model for knowledge discovery: Case study in spanish clinical text. En *Proceedings of the 2nd clinical natural language processing workshop* (pp. 79–88).

6 Anexo I: Estado de la nueva web de LETO

En este anexo se mostrará el resultado de la creación de la nueva web para interactuar con LETO. Esta web se ha creado utilizando HTML, CSS y JS. Se han utilizado elementos de Bootstrap, y desde el JavaScript se consumen los métodos del API creada.

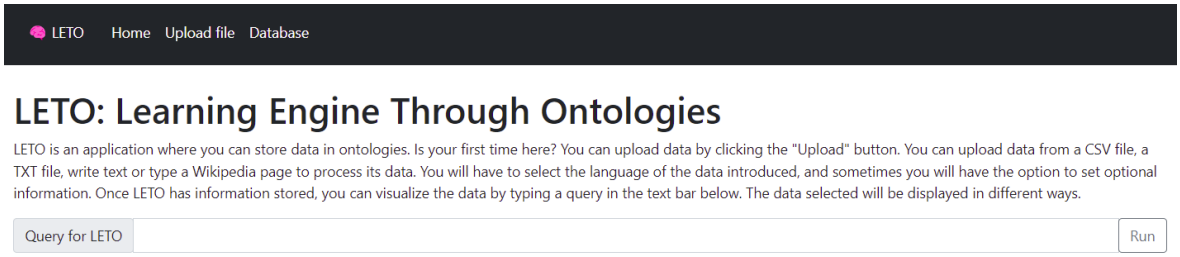

Figura 6.1: Página de inicio de LETO

En la Figura 6.1 se muestra la vista inicial de la aplicación. En ella se puede visualizar una barra superior, la cual contiene tres botones: Uno para ir al inicio de la página, otro para subir archivos y el último para mostrar el estado de la base de datos. En la parte inferior se muestra el título de la web, así como un breve texto descriptivo de LETO. Por último, se encuentra una barra para introducir la consulta a realizar.

En caso de querer introducir información, se deberá de pinchar sobre el botón “Upload file”. Aparecerá un menú lateral en el que se puede seleccionar desde un desplegable el método de entrada de la información. En la Figura 6.2 se muestra el caso de querer introducir los datos mediante un archivo CSV. El usuario podrá seleccionar el fichero, escribir su entidad principal, seleccionar el lenguaje (entre castellano e inglés), y ver y seleccionar una lista de sensores a aplicar sobre la información.

Por otro lado, en caso de querer consultar el estado de la base de datos, se deberá de pinchar sobre el botón “Database”. Volverá a aparecer otro menú lateral, pero en este caso contendrá información sobre el número de elementos almacenados, así como un botón para limpiar la base de datos. En la Figura 6.3 se muestra ese menú.

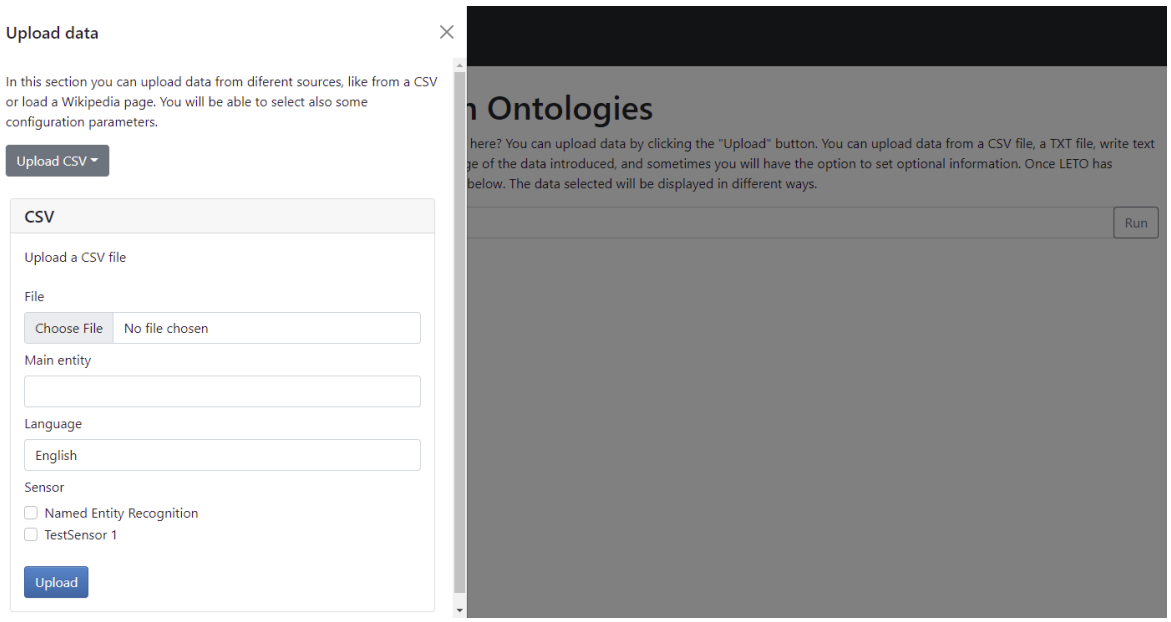

Figura 6.2: Menú para introducir información mediante un fichero CSV

Por último, en caso de querer realizar una consulta, se deberá de escribir en lenguaje natural sobre la barra de la página inicial "Query for LETO". Tras escribirla y pinchar sobre "Run" se ejecutará la respuesta y aparecerán una serie de visualizadores. En la Figura 6.4 se muestra un ejemplo de visualizador, que en este caso es un gráfico.

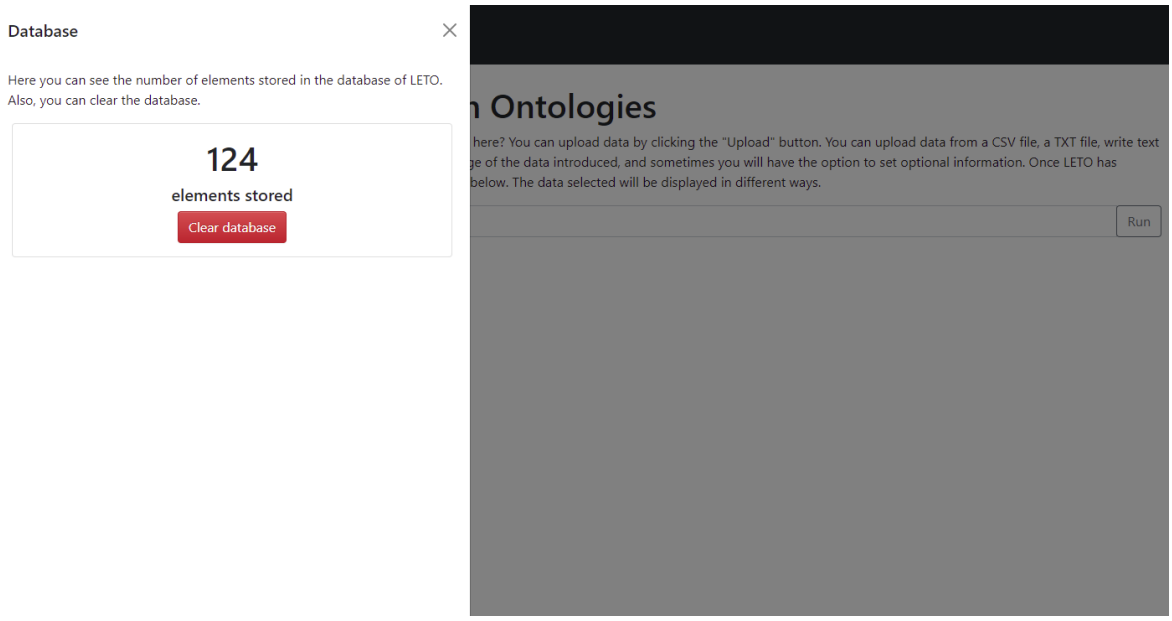

Figura 6.3: Menú para consultar la información de la base de datos

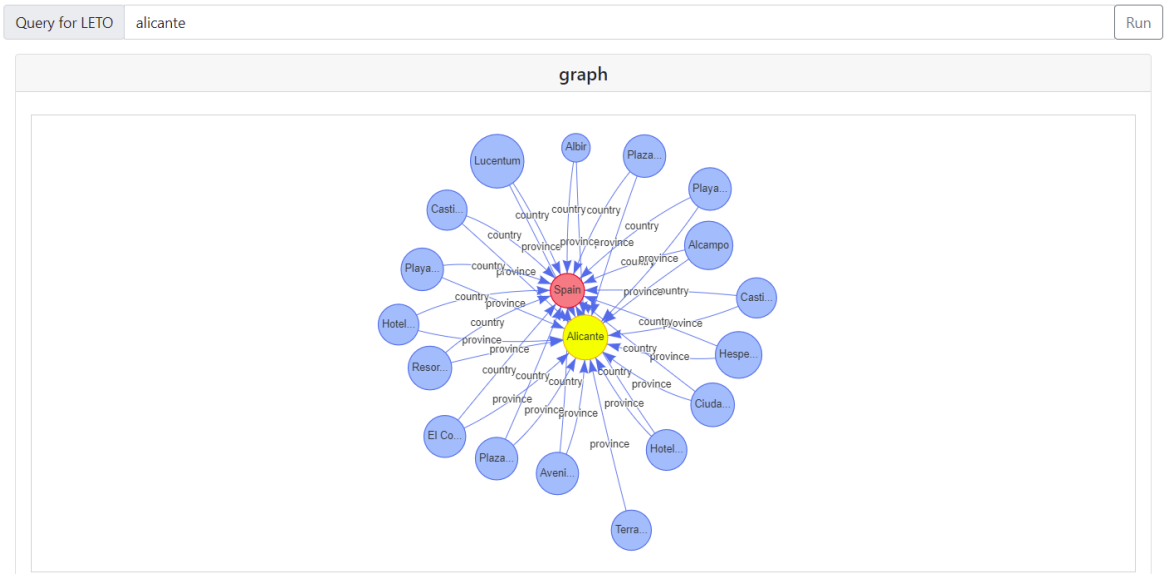

Figura 6.4: Gráfico generado a partir de una consulta realizada a LETO

7 Anexo II: Ejemplo de uso

Una vez visto en el Anexo I el diseño de la nueva web, se procede a realizar en este anexo un ejemplo de uso. Para el ejemplo, se utilizarán tres archivos CSV:

- `locations_alicante.csv`: Este archivo contiene una lista de localizaciones de la provincia de Alicante. Todas esas localizaciones vienen identificadas por un nombre, su latitud y longitud, así como información sobre el país y provincia dónde se encuentran (en este caso siempre serán localizaciones de Alicante, España).
- `comments_alicante.csv`: Se describen comentarios realizados sobre ciertas localizaciones, todas ellas de Alicante. Además, se proporciona información sobre la fecha del comentario, así como su origen y el país desde el que se ha realizado.
- `tourism_spain.csv`: Este último fichero contiene información acerca de los turistas recibidos en España cada mes.

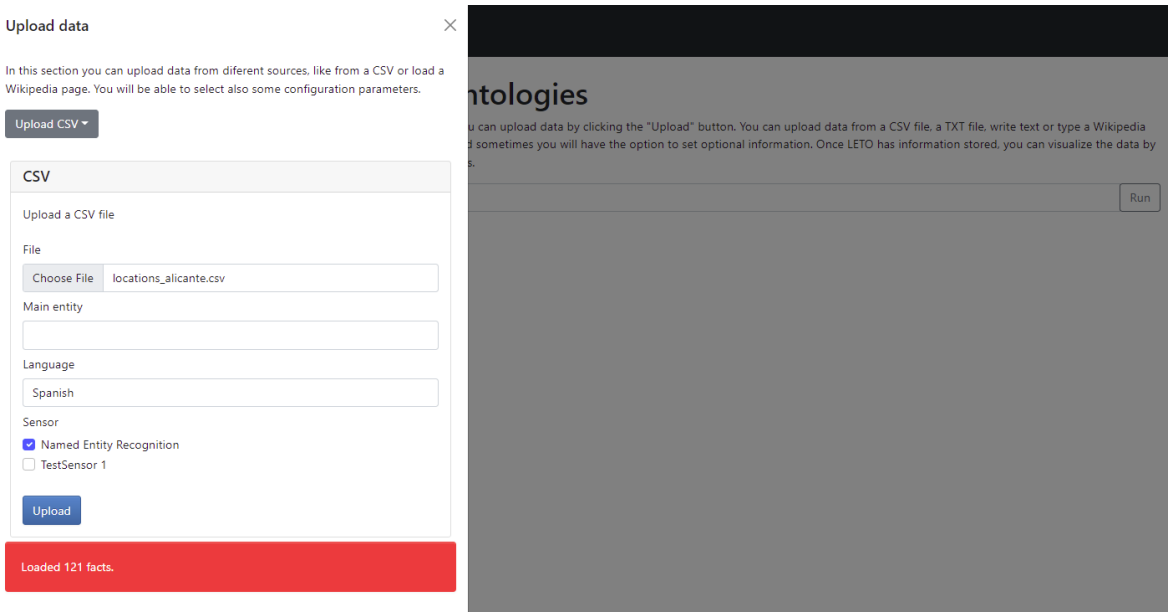

Figura 7.1: Carga del fichero CSV

Una vez se conocen los archivos a utilizar, se describe el proceso de carga.

1. Se carga el fichero `locations_alicante.csv`, especificando que el lenguaje de la información es el castellano y que se utilizará el sensor Named Entity Recognition.

2. Se carga el fichero `comments_alicante.csv` con la misma configuración anterior, pero en este caso se especificará que la entidad principal es el comentario ("comment").

3. Se carga `tourism_spain.csv` especificando que la entidad principal se llamará "TourismInfo".

Al realizar una carga de información, aparece un recuadro rojo mostrando el número de elementos reconocidos. Se muestra en la Figura 7.1 el estado de la web al cargar el primer fichero.

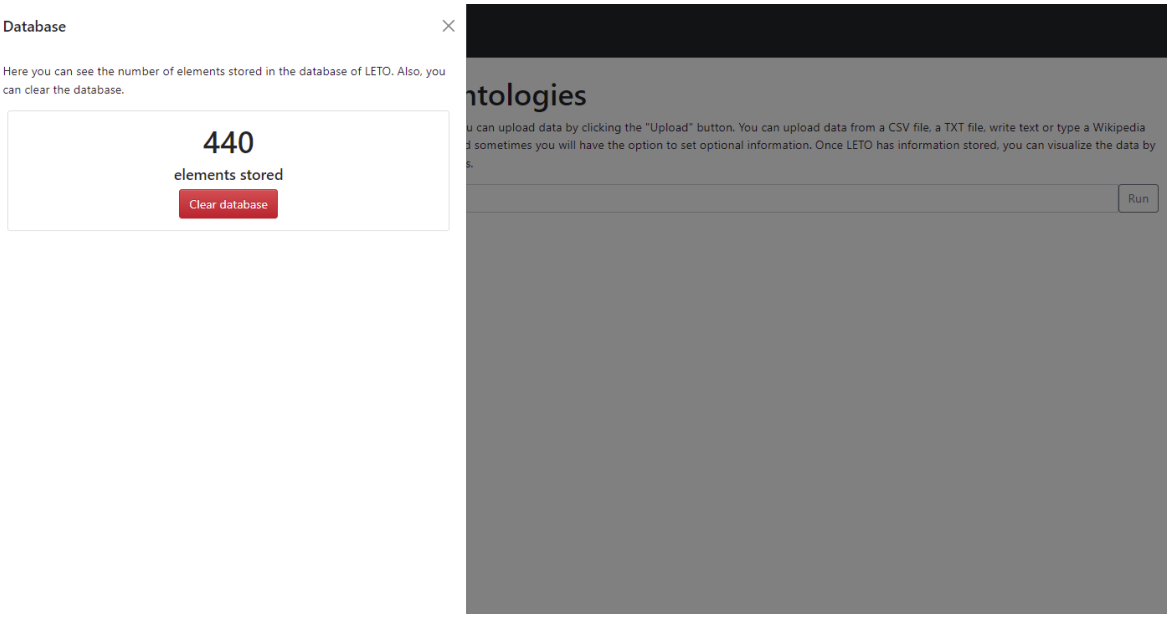

Figura 7.2: Menú en el que se muestran el número de elementos almacenados tras las carga

Al tener cargada ya toda la información, se puede consultar el estado de la base de datos para comprobar que toda la información se ha cargado. En la Figura 7.2 se puede visualizar ese apartado de la web.

Ahora que la aplicación cuenta con información, se pueden realizar consultas, todas ellas escritas en lenguaje natural. Se mostrarán varios ejemplos, desde consultas más sencillas a más complicadas.

La primera consulta que se realiza es "spain". El primer visualizador que se muestra es el de un gráfico, como se puede apreciar en la Figura 7.3. En él se puede ver como la entidad con más relaciones es "Spain", la cual se relaciona con muchos eventos (entidades azules) y lugares (entidades amarillas). Al pinchar sobre una entidad se puede visualizar más información sobre la misma. En este caso, al pinchar sobre "Alhambra" se visualizan los datos de longitud y latitud.

Se realiza otra consulta, "alicante mentions". Ahora se pueden observar otros gráficos. Se muestra en la Figura 7.4 un mapa en el que se encuentran representadas todas las localizaciones que se cuentan en la base de datos y que son mencionadas en los comentarios. Otro de los visualizadores, el de la Figura 7.5, muestra un gráfico circular en el que se representan las localizaciones mencionadas y su porcentaje de aparición con respecto al total.

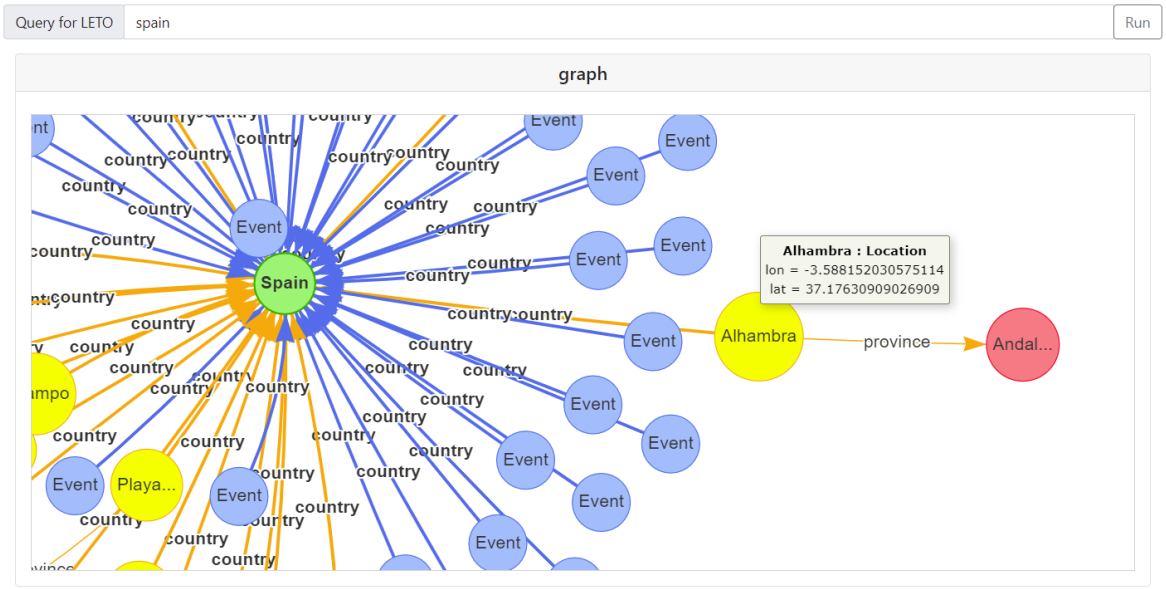

Figura 7.3: Gráfico mostrado tras realizar la consulta ”spain”

Por último, se realiza la consulta “spain tourists by country”. Al recibir los datos, se muestra el gráfico de la Figura 7.6 en que se representan los turistas que han visitado España con respecto a la fecha.

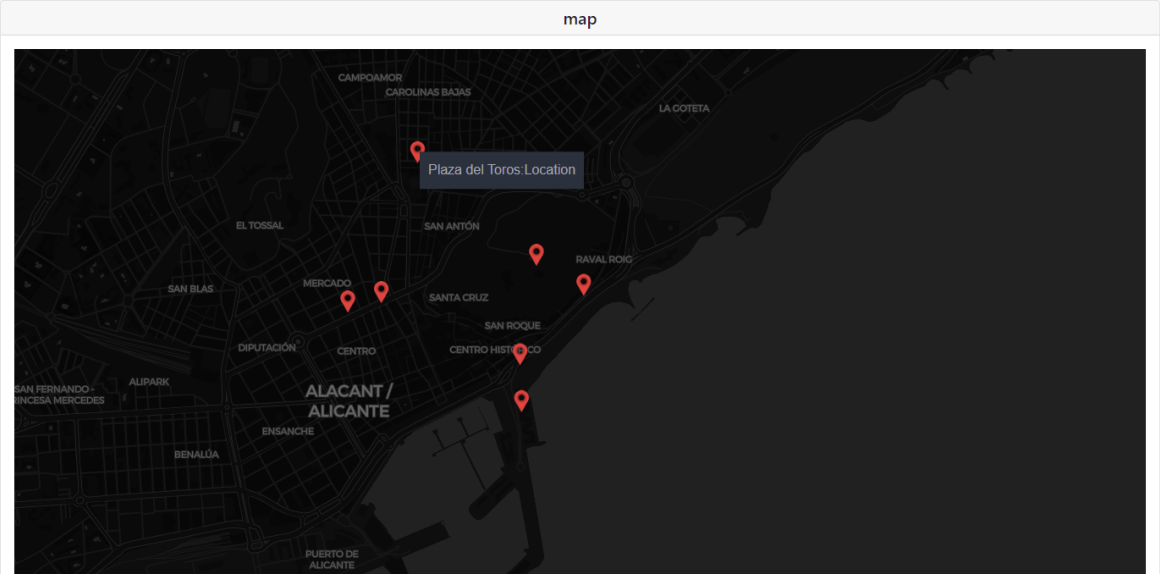

Figura 7.4: Mapa en el que se representan las localizaciones de Alicante que tiene LETO

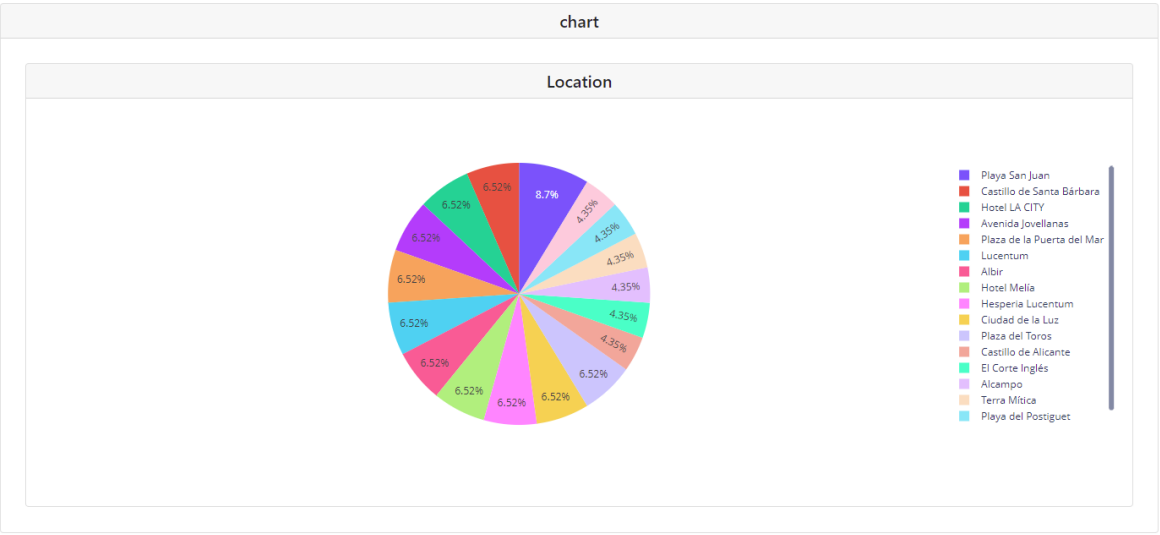

Figura 7.5: Gráfico circular en el que se representan las localizaciones de Alicante que tiene LETO

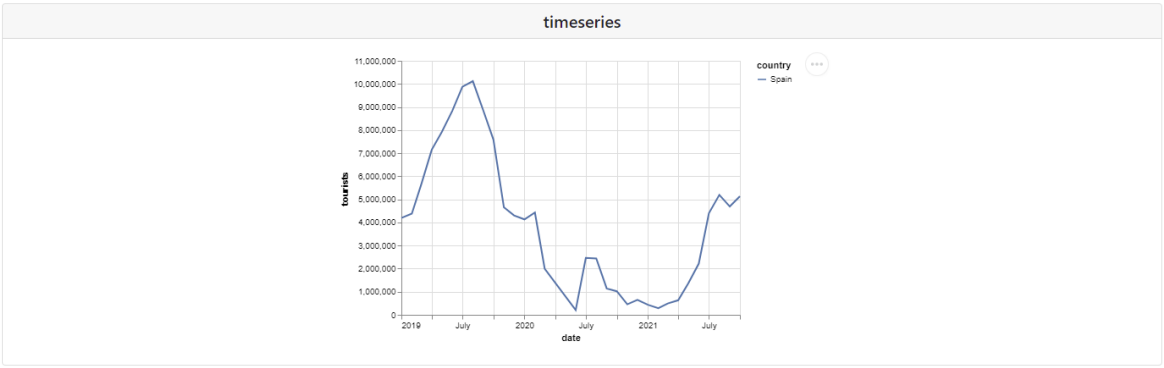

Figura 7.6: Gráfico con los turistas que ha recibido España cada mes