Grado en Ingeniería de la Salud
Mención Bioinformática

Desarrollo de Workflows de Análisis de Datos

Data Analytics Workflows Development

Realizado por
Irene Sánchez Jiménez

Tutorizado por
Ismael Navas Delgado
Antonio Benítez Hidalgo

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, junio de 2021

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADA EN INGENIERÍA DE LA SALUD

MENCIÓN BIOINFORMÁTICA

Desarrollo de Workflows de Análisis de Datos

Data Analytics Workflows Development

Realizado por
Irene Sánchez Jiménez

Tutorizado por
Ismael Navas Delgado
Antonio Benítez Hidalgo

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2021

Fecha defensa: julio de 2021

Abstract

For this Final Degree Project, we have developed a way to demonstrate the usefulness of semantic technologies, especially Linked Data, in the world of Life Sciences.

For the development of this tool, we have analysed the repositories that are available in the Linked Open Data Cloud and are related to genetic diseases, genes and proteins in depth. The mechanisms by which these repositories have been created in RDF have been studied in order to replicate them locally.

For those databases where there is no RDF repository or it is very outdated, we have analysed how these technologies can be used with existing public data.

Finally, a web application has been developed so that end users (scientists) can take advantage of these technologies and obtain relevant information on genetic diseases, proteins and genes.

Keywords: Linked Data, Life Sciences, Disease, Gene, Protein

Resumen

En este Trabajo Fin de Grado se ha desarrollado un demostrador de la utilidad de las tecnologías semánticas, especialmente del Linked Data, en el mundo de las Ciencias de la Vida.

Para el desarrollo de este demostrador, se han analizado en profundidad los repositorios que se encuentran disponibles en Linked Open Data Cloud y tienen relación con enfermedades genéticas, genes y proteínas. Se han estudiado los mecanismos por los que se han creado esos repositorios en RDF para poder replicarlos de manera local.

Para aquellas bases de datos para las que no existe repositorio en RDF o este se encuentra muy desactualizado, hemos analizado la forma en la que se pueden usar estas tecnologías con los datos públicos ya existentes.

Se ha desarrollado, finalmente, una aplicación Web para que los usuarios finales (científicos) puedan aprovechar estas tecnologías y obtener información de interés sobre enfermedades genéticas, proteínas y genes.

Palabras clave: Linked Data, Ciencias de la Vida, Enfermedad, Gen, Proteína

Índice

1. Introducción **7**

1.1. Motivación 7

1.2. Objetivos 8

1.3. Estructura del documento 9

2. Contexto **11**

2.1. Aplicaciones con Linked Data en Ciencias de la Vida 11

2.2. Tecnologías utilizadas 12

2.2.1. RDF 12

2.2.2. Linked Data 13

3. Bases de datos **15**

3.1. Definición de las bases 15

3.1.1. GenBank 15

3.1.2. OMIM 15

3.1.3. UniProt 16

3.2. Descripción del proceso de acceso manual a las bases 16

3.2.1. GenBank 16

3.2.2. OMIM 18

3.2.3. Uniprot 19

3.3. Datos disponibles en RDF 21

3.3.1. GenBank 21

3.3.2. OMIM 24

3.3.3. Uniprot 25

4. Requisitos **27**

4.1. Aplicación Web 27

4.1.1. Requisitos funcionales 27

4.1.2. Requisitos no funcionales 30

4.2. Acceso a RDF 30
4.2.1. Requisitos funcionales 30

5. Diseño **33**
5.1. Diagrama de casos de uso 33
5.2. Diagramas de secuencias 34
5.2.1. GenBank 34
5.2.2. OMIM 36
5.2.3. Uniprot 38

6. Implementación **43**
6.1. Aplicación Web 44
6.2. Acceso a RDF 45
6.2.1. Uniprot 46
6.2.2. OMIM 50
6.2.3. GenBank 56

7. Resultados **69**
7.1. Genes 69
7.2. Enfermedades 73
7.3. Proteínas 77

8. Conclusiones y Líneas Futuras **83**
8.1. Conclusiones 83
8.2. Líneas Futuras 85

1

Introducción

1.1. Motivación

En la actualidad, tenemos a nuestra disposición una gran cantidad de bases de datos de distinta índole. Dentro de estas, una parte importante son las bases de datos biológicas. Por ejemplo, podemos encontrar bases de datos sobre genes (EMBL[1] o GenBank[2]), sobre enfermedades (OMIM[3]) y sobre proteínas (Uniprot[4] o PDB[5]).

La información disponible en estas bases de datos se encuentra extremadamente ligada entre sí ya que, por ejemplo:

- Un gen o la combinación de varios genes pueden producir enfermedades genéticas.
- Los genes se traducen en proteínas.
- Hay proteínas cuya alteración deriva en enfermedades.

Por tanto, estas bases de datos se encuentran muy relacionadas, llegando al punto de que las entradas en todas ellas contienen referencias cruzadas con varias bases de datos. El problema es que estas bases de datos están diseñadas para ser utilizadas por el ser humano (médicos, investigadores, farmacéuticos, etc) y no por ordenadores, por lo que resulta complicado la integración y automatización a la hora de recuperar información de las mismas.

Con la aparición de las tecnologías semánticas se han puesto en valor todos estos datos públicos de gran interés para la comunidad científica. En la nube de Linked Open Data Cloud[6] podemos ver que estas tecnologías han tenido una enorme aceptación en el ámbito de Las Ciencias de la Vida (Figura 1). Desgraciadamente, la explotación de esta información no está siendo del todo efectiva ya que hay problemas que surgen por su propia naturaleza: repositorios que se encuentran inaccesibles, datos que no se actualizan con frecuencia, tiempos de respuesta elevados, etc.

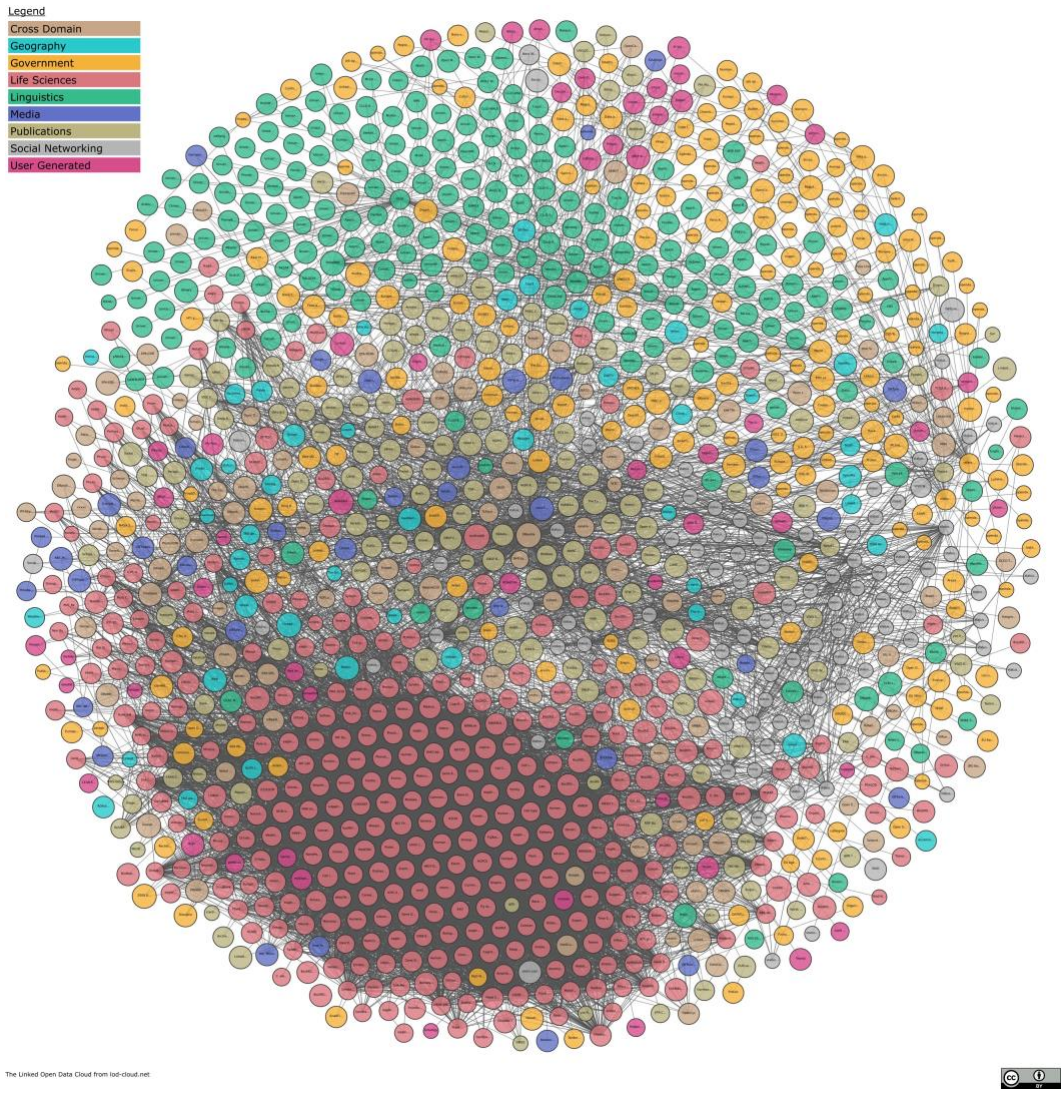


Figura 1: *Figura actual del Linked Open Data Cloud*

La publicación de datos como Linked Data[7] se basa en el despliegue de bases de datos RDF con una API Rest que permita la evaluación remota de consultas SPARQL[8]. Estos mecanismos se proporcionan de forma nativa en sistemas gestores de bases de datos como Virtuoso[9] o StarDog[10].

1.2. Objetivos

El objetivo de este Trabajo Fin de Grado es el desarrollo de un demostrador tecnológico de la utilidad de las tecnologías semánticas, en especial del Linked Data en el contexto de Las Ciencias de la Vida. Para ello, se analizarán los repositorios existentes en Linked Open

Data Cloud que tengan relación con enfermedades genéticas, de forma que se investiguen los mecanismos por los que se han generado esos repositorios en RDF, para replicarlos de forma local para al menos dos bases de datos que presenten problemas de acceso o actualizaciones. En aquellos casos para los que no exista repositorio funcionando publicado como Linked Data se analizará la forma en que usar estas tecnologías con los datos públicos existentes. Usando esa información junto al mejor repositorio RDF en este dominio que es el de Uniprot (información de proteínas)[4], se diseñarán consultas federadas (Federated SPARQL[8]) que aprovechen la existencia de relaciones explícitas entre los datos de estas bases de datos.

Usando las consultas diseñadas, se desarrollará una aplicación Web que permita a usuarios finales (científicos) el aprovechamiento de estas tecnologías para obtener información integrada relativa a enfermedades genéticas. Para ello, se desarrollará una API Rest que permita el acceso a un Sistema Gestor de Bases de Datos RDF.

1.3. Estructura del documento

Después de esta introducción, continuamos con un apartado donde hablaremos sobre otros proyectos similares que se han realizado y sobre las tecnologías que vamos a utilizar durante el desarrollo del actual proyecto (Capítulo 2). Seguiremos con una sección que nos ayudará a conocer las tres bases de datos empleadas (Capítulo 3). En el cuarto capítulo, discutiremos cuáles son los requisitos que han de cumplirse tanto en la interfaz de usuario como en la accesibilidad a la información en formato RDF (Capítulo 4). En los dos siguientes capítulos, los más amplios, expondremos el diseño (Capítulo 5) e implementación (Capítulo 6), respectivamente, de nuestra herramienta Web. Le seguirá un apartado donde expondremos la utilidad de la aplicación (Capítulo 7). Finalizaremos la memoria con un apartado de conclusiones y trabajos futuros en esta línea (Capítulo 8).

2

Contexto

2.1. Aplicaciones con Linked Data en Ciencias de la Vida

Los Datos Enlazados o Linked Data han cobrado especial importancia en los últimos tiempos, sobre todo en el ámbito de las ciencias de la vida, ya que ofrecen una manera efectiva de compartir información. La forma de establecer estos enlaces entre bases de datos es similar al mecanismo de hiperenlaces de las páginas Web. Cada base de datos, incluye en sus datos referencias a otras bases de datos, indicando el tipo de relación que existe entre cualquier dato de su base de datos y algún dato de la base de datos referenciada. El tipo de relación más común es el que identifica dos instancias como iguales (*sameIndividualAs*).

Estos datos no solo se enlazan entre sí, sino que los mismos enlaces se caracterizan mediante ontologías, lo que permite distinguir los tipos de enlaces. La definición de estas ontologías es la parte más compleja y tediosa del proceso para los proveedores de datos, pero permite que haya una alta interoperabilidad con el software de análisis y visualización de datos. Este aumento en la interoperabilidad facilita la recuperación de datos de diversa índole, a la vez que hace que tanto esa recuperación como su visualización sea rápida. Esta recuperación se suele hacer mediante el lenguaje de consultas SPARQL, que se utiliza para consultar bases de datos RDF.

En Ciencias de la Vida, como hemos comentado antes, esta tecnología ha tomado gran relevancia, ya que es un ámbito donde conviene tener la información relacionada entre sí atendiendo a ciertos criterios. Por ejemplo, hablando de medicina, hay patologías que pueden tener un origen común o cuadros clínicos similares, información que puede estar enlazada gracias a la Web Semántica y Linked Data.

Hablando de casos particulares, los Datos Enlazados pueden aplicarse al análisis de microarrays en el estudio de la enfermedad de Alzheimer (EA)[11]. A la hora de estudiar esta enfermedad, se realizan unos estudios de microarrays clínicos típicos, donde los genes expre-

sados diferencialmente se identifican de manera estadística al comparar el control (persona sana) con el caso (persona que padece la enfermedad. Entre estos genes, los investigadores reducen aquellos considerados candidatos responsables de la patología en función de sus propiedades: anotaciones funcionales, vías metabólicas y de señalización, etc. Este proceso para reducir los genes se trata de un proceso exploratorio. Los investigadores clínicos tienen que analizar los genes desde tantos puntos de vista como sea posible. Si hay una nueva propiedad de los genes y se encuentra publicada en la web, los investigadores la incorporan para investigarla. En este proceso exploratorio es donde cobran gran importancia los Datos Enlazados. Todas las propiedades y características asociadas a un gen pueden recuperarse rastreando mediante Linked Data, y posteriormente se pueden analizar de manera sencilla.

2.2. Tecnologías utilizadas

2.2.1. RDF

RDF se trata de un modelo conceptual impulsado por la WC3 que cubre la necesidad de establecer un lenguaje común para describir los recursos de la web. Es decir, nos permite formalizar mediante un lenguaje estándar las descripciones sobre los recursos web. De esta manera, se facilita la reutilización.

Una de las características que diferencian a RDF de otros modelos es que ofrece la posibilidad de combinar distintas fuentes de datos, aunque los esquemas correspondientes sean distintos. Además, es fácilmente extensible.

RDF se basa en la idea de identificar los recursos usando los **Uniform Resource Identifiers** o **URIs**, y describiendo estos recursos en términos de *propiedades simples* y *valores*. Una descripción RDF es un conjunto de proposiciones simples (también llamadas sentencias o declaraciones). Cada proposición se conoce también como **tripleta**, porque se compone de sujeto, predicado y objeto. Esta proposición o sentencia se puede expresar usando la tripleta o usando la notación de *grafos de nodos y arcos*.

Por tanto, RDF ofrece un mecanismo para expresar declaraciones simples sobre recursos utilizando *propiedades* y *valores*. No obstante, también es necesario disponer de otro mecanismo que permita definir los vocabularios que queremos usar en estas declaraciones.

RDF no cuenta con un mecanismo que satisfaga estas características, por lo que hace uso

del ***Lenguaje de Descripción de Vocabularios RDF***, más conocido como ***RDF Schema***.

RDF Schema no proporciona un vocabulario en sí, sino que ofrece mecanismos para especificar que las clases o propiedades que se están definiendo son parte de un vocabulario y cómo se espera que se relacionen. También permite definir a los recursos como instancias de una o más clases y que estas clases puedan estar organizadas en una jerarquía.

Para consultar los grafos RDF que se construyen, necesitamos también de un mecanismo que nos permita hacerlo. Así surge **SPARQL**, un lenguaje estandarizado para la consulta de grafos RDF. SPARQL trabaja en términos de *IRIs*, que son un subcojunto de las referencias URI pero que omiten los espacios.

2.2.2. Linked Data

Tim Berners-Lee publicó en 2001 un artículo[12] donde presentó una de las tecnologías que ha acabado siendo clave en el desarrollo de una sociedad totalmente conectada entre sí: **la Web Semántica**. Se pretende que esta web sea una extensión de la web actual en la que el significado (semántica) de la información y de los servicios esté bien definido.

Como resultado de esta iniciativa nació el **Linked Data** o **Datos Enlazados**, que se refiere a un método de publicación unificada y estructurada de los datos. Estos datos se encuentran interrelacionados, por lo que resultan más útiles para aquellos que quieran consultarlos y profundizar en algún tema específico.

En 2010, Tim Berners-Lee publica otro artículo donde propuso un sistema para medir la calidad de los datos abiertos en función a su nivel de reutilización. A este sistema se le conoce como **"Las cinco estrellas"**.

Establece cinco niveles, ordenados de menor a mayor facilidad de reutilización de los datos publicados. Dicha jerarquía mide cómo de abiertos y usables son los datos que puede ofrecer cualquier organización. En la Figura 2 podemos ver un esquema del sistema propuesto por Berners-Lee. Los niveles son los siguientes:

- *Una estrella*. Ofrecer los datos, es decir, la simple publicación de los datos independientemente de su formato. Aquí tendremos los datos abiertos no estructurados (pdf, imagen escaneada), difíciles de manipular y reutilizar.

- *Dos estrellas*. Publicar los datos de manera estructurada, para lo cual se ha usado un

software propietario. Aquí podríamos encontrar un archivo Excel o con extensión XLS.

- *Tres estrellas*. Ofrecer los datos de manera estructurada pero sin haber utilizado un software propietario, sino un software libre, como OpenOffice o CSV.
- *Cuatro estrellas*. Publicar los datos estructurados con URIs que identifiquen los recursos. Aquí encontraríamos el estándar RDF.
- *Cinco estrellas*. Publicar los datos que cumplen con todos los requisitos anteriores, además de estar enlazados con otros datos similares de otras organizaciones. Nivel máximo de madurez.

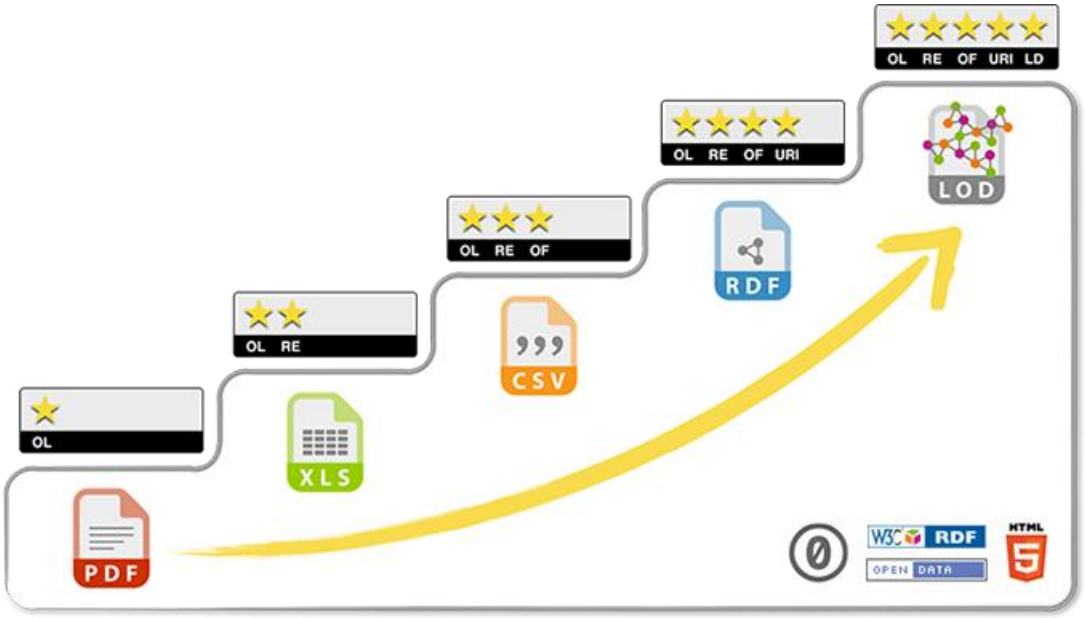


Figura 2: *Método de las Cinco estrellas para medir la capacidad de ser reutilizados de los datos abiertos*

3

Bases de datos

3.1. Definición de las bases

3.1.1. GenBank

GenBank[2] es la base de datos de secuencias genéticas del **NIH** (National Institutes of Health). Fue fundada en 1982 por Walter Goad y reúne todas las secuencias de ADN de acceso público, incluyendo anotaciones.

Esta base forma parte de un consorcio llamado *International Nucleotide Sequence Database Collaboration*, integrado por tres bases de datos: *DNA DataBank of Japan* (**DDBJ**), *European Nucleotide Archive* (**ENA**) y **GenBank**. Estas tres bases intercambian información todos los días para ofrecer siempre los mismos datos.

GenBank está diseñada para proporcionar acceso a la comunidad científica a la información más actualizada y completa sobre secuencias de ADN. Por ello, el NCBI no impone ningún tipo de restricción al uso y distribución de sus datos.

3.1.2. OMIM

OMIM (Online Mendelian Inheritance in Man)[3] es una recopilación exhaustiva de genes y fenotipos genéticos humanos que se encuentra disponible de manera gratuita y se actualiza diariamente. Incluye todas las enfermedades de base genética y, si existe evidencia genética, también ofrece información sobre su manifestación fenotípica. Las entradas contienen abundantes enlaces a otros recursos genéticos.

Esta base de datos fue iniciada a principios de los años 60 por el Doctor Victor A. McKusick como un registro de rasgos y trastornos mendelianos, que se tituló Mendelian Inheritance in Man (MIM). La versión en línea, OMIM, fue creada en 1985 por una colaboración entre la Biblioteca Nacional de Medicina y la Biblioteca Médica William H. Welch de Johns Hopkins.

En el año 1987, se puso a disposición pública en Internet, y en 1995, el NCBI desarrolló OMIM para la World Wide Web.

3.1.3. UniProt

UniProt[4] es un repositorio de datos gratuito sobre proteínas. Es la principal base de datos a nivel mundial en cuanto al almacenamiento de información sobre proteínas.

UniProt ha sido desarrollada por el *Instituto Europeo de Bioinformática* (**EBI**), el *Instituto Suizo de Bioinformática* (**SIB**) y el *Recurso de Información sobre Proteínas* (**PIR**). Esta base de datos se mantiene actualizada gracias a los fondos públicos procedentes principalmente del Instituto Nacional de Salud, **NIH** de Reino Unido.

Dentro de UniProt, encontramos *Swiss-Prot* y *TrEMBL*. **Swiss-Prot** es la parte que se encuentra anotada y revisada manualmente. Se trata de una base de datos de secuencias de proteínas de alta calidad, anotada y no redundante, que recopila resultados experimentales, características calculadas y conclusiones científicas. Por otro lado, **TrEMBL** es la sección computacionalmente anotada que complementa a Swiss-Prot para crear UniProtKB. Esta base contiene las traducciones de todas las secuencias codificantes presentes en las bases de datos de secuencias nucleotídicas EMBL/GenBank/DDBJ y también secuencias extraídas de la literatura o enviadas a Swiss-Prot.

3.2. Descripción del proceso de acceso manual a las bases

3.2.1. GenBank

GenBank es accesible de manera pública y gratuita a través de Internet con el siguiente enlace: http://www.ncbi.nlm.nih.gov/genbank/. Si accedemos, encontramos la página de inicio para realizar búsquedas manualmente (Figura 3). Como podemos observar, aparece un buscador en la parte superior y una serie de información acerca de la base de datos.

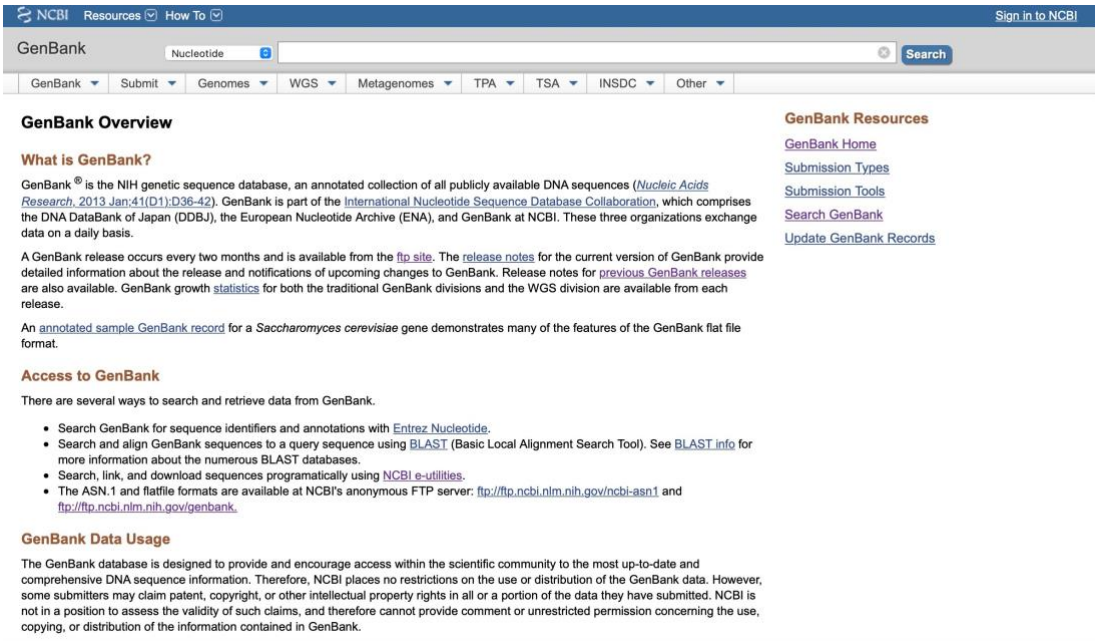


Figura 3: *Página de inicio de GenBank*

Podemos realizar búsquedas usando **palabras clave**, en cuyo caso se escriben entre comillas, por ejemplo ”blindness”. Si se quieren escribir varios conceptos se pueden usar los operadores lógicos AND, OR y NOT. Sin embargo, no es recomendable utilizar este tipo de búsquedas ya que la mayoría de entradas de GenBank carecen de palabras clave.

Es recomendable buscar a través del **nombre del gen**, ya sea completo o abreviado. También se puede buscar mediante el **nombre del autor** o la persona que envió la secuencia. Si se busca por autor, primero debe introducirse el apellido, luego dejar un espacio y escribir el nombre. Se pueden filtrar los resultados de la búsqueda atendiendo a varios criterios, como el tipo de molécula, la especie, etc.

Para acceder directamente a un registro se introduce su **número de acceso**, que se trata de un número único para cada registro, compuesto por letras y número, como por ejemplo MA809397. En el caso de que la secuencia cambie o lo hagan sus anotaciones, lo que cambia es la *versión del registro*. Si la primera versión es la *MA809397.1* y se introducen cambios, la siguiente versión será la *MA809397.2* [13].

3.2.2. OMIM

OMIM es accesible públicamente en Internet a través del enlace: https://omim.org. Al entrar, nos encontramos con la página principal con información general y opciones de búsqueda (Figura 4).

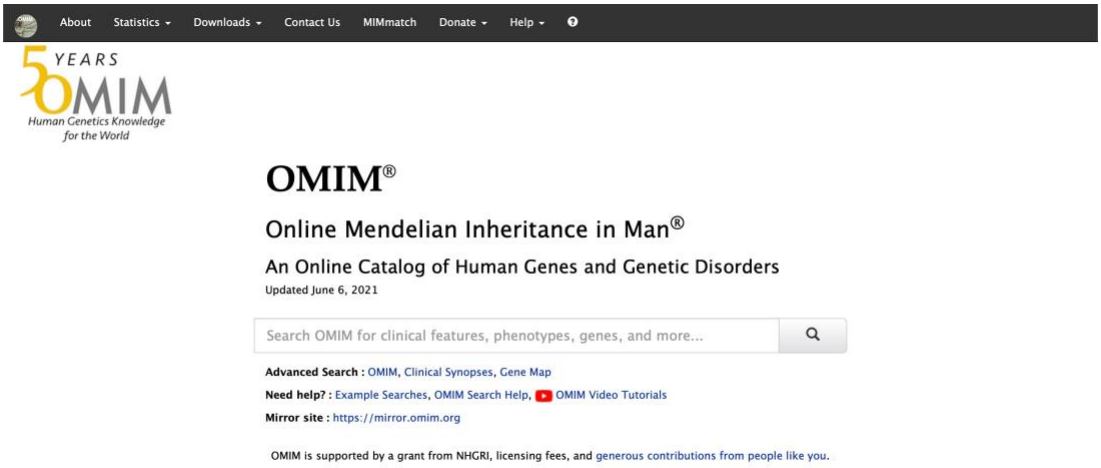


Figura 4: *Página de inicio de OMIM*

Como es normal en páginas dedicadas a la búsqueda y explotación de información, nos encontramos con un buscador, donde podemos introducir las características clínicas, los fenotipos o la enfermedad que queramos investigar. Es importante resaltar que ofrece búsquedas avanzadas, para la página de OMIM en su totalidad, para la sinopsis clínica y para el mapa de genes. La más útil en nuestra búsqueda es la primera, la de **OMIM** (Figura 5). Como podemos ver, se puede filtrar el resultado buscando por título, por variantes alélicas, indicando el número de cromosomas afectado por la enfermedad o que la provoca, etc.

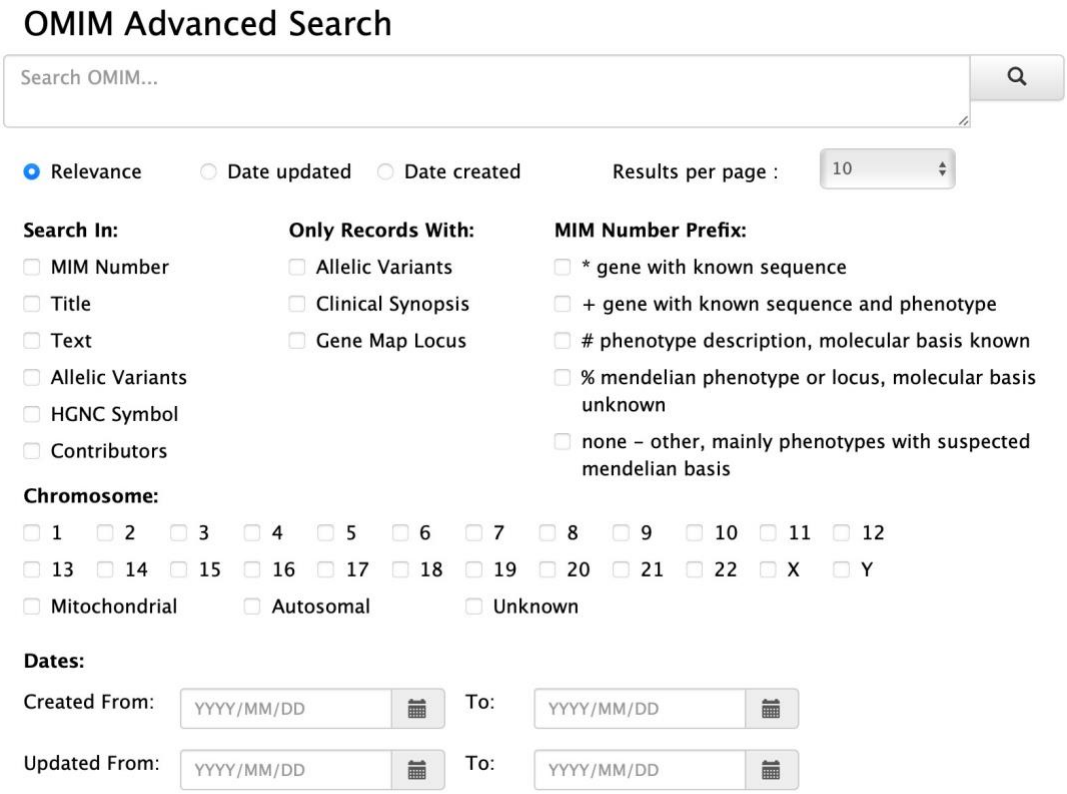


Figura 5: *Búsqueda avanzada en OMIM*

3.2.3. Uniprot

Como ya hemos comentado antes, Uniprot es accesible de manera gratuita a través de Internet usando el siguiente enlace: https://www.uniprot.org. Accediendo a él, nos encontramos la página principal (Figura 6).

Como podemos observar, nos encontramos con un buscador en la parte superior donde podemos introducir la proteína que deseamos buscar.

También se muestra una columna azul en la parte izquierda, para acceder directamente a *UniprotKB*, el eje central donde se recopila la información funcional sobre proteínas, anotada de manera precisa y coherente.

Contamos con otras tres secciones:

- UnirRef

  Para acceder a los clústeres de referencia de Uniprot, que proporcionan conjuntos agrupados de secuencias de Uniprot. Así se ocultan las secuencias que sean redundantes.

- UniParc

  Es una base de datos no redundante que cuenta con la mayor parte de las secuencias de proteínas disponibles públicamente en el mundo.

- Proteomes

  Aquí se proporcionan los proteomas de las especies cuyo genoma esté completamente secuenciado.

En el cuadro inferior de color morado se proporcionan datos de apoyo, como es la posibilidad de acceder a las publicaciones citadas en Uniprot, visitar referencias a otras bases de datos, etc.

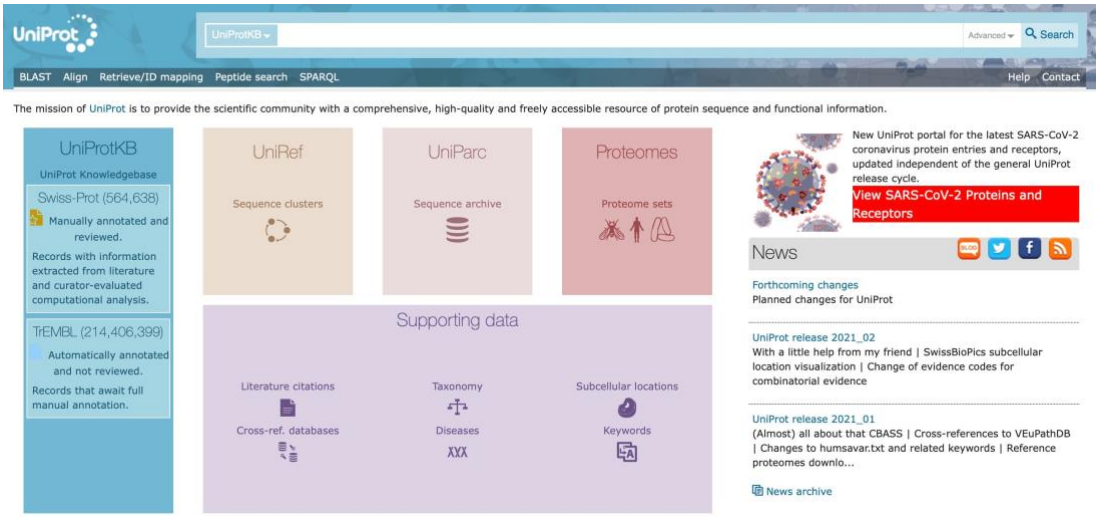


Figura 6: *Página de inicio de Uniprot*

Por otro lado, en el buscador que hemos mencionado al principio nos aparece una pestaña en la parte derecha que dice *"Advanced"*. Si pulsamos sobre esa pestaña nos sale un desplegable en el que podemos seleccionar filtros para la búsqueda (Figura 7).

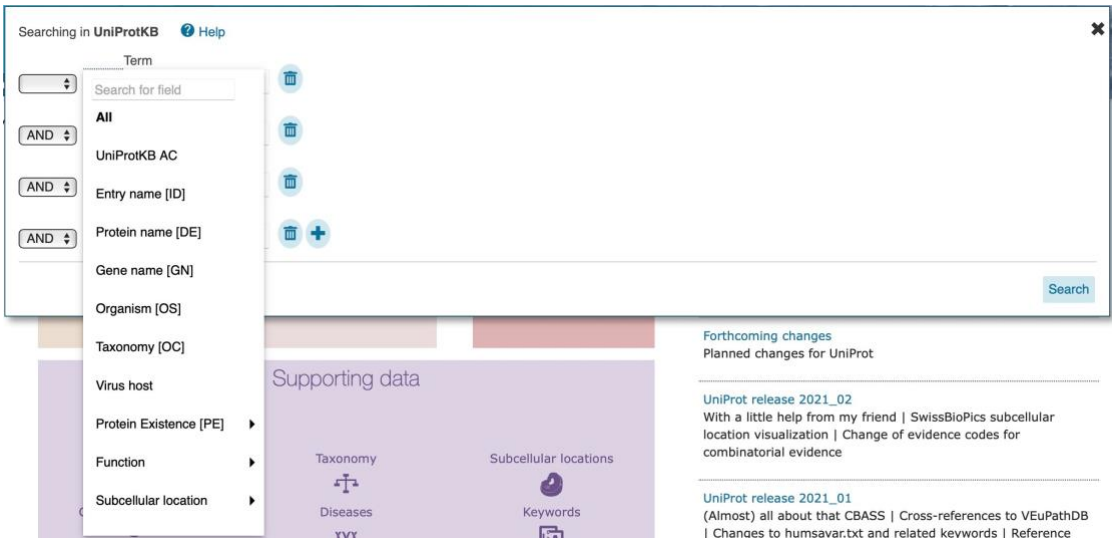


Figura 7: *Búsqueda avanzada en Uniprot*

3.3. Datos disponibles en RDF

La situación ideal para la realización de este proyecto sería la de encontrar los repositorios publicados como bases de datos RDF, ofreciendo un interfaz de consultas a través de SPARQL Endpoints. Desgraciadamente, no se ha dado esa situación idílica.

3.3.1. GenBank

Hemos empezado indagando en *Linked Open Data Cloud* sobre algún repositorio disponible para GenBank. Para nuestra sorpresa, esta base de datos no aparece en la nube. Entonces, hemos pasado a buscar información sobre el NCBI, del cual forma parte GenBank. Para este caso sí se han encontrado resultados. Sin embargo, no han sido los indicados. El único repositorio RDF que se ha encontrado ha sido en *Bio2RDF* [14], que se encuentra bastante desactualizado, ya que este proyecto finalizó y no mantiene su actividad. Bio2RDF permitió ofrecer una gran cantidad de datos públicos en Ciencias de la Vida al inicio de los Linked Data, motivando a algunos propietarios de estos datos públicos a seguir ese camino. Pero el enorme esfuerzo y coste de mantenimiento de estas soluciones hace que mantenerlas de manera externa a las bases de datos originales sea inviable.

Por tanto, optamos por buscar otra solución. Investigando en la Web, hemos encontrado un mecanismo para buscar y descargar información de GenBank y de cualquier base perteneciente

al NCBI: las llamadas **E-utilities**[15]. Son un conjunto de ocho programas que se ejecutan del lado del servidor y ofrecen una interfaz estable para la consulta de bases de datos del Centro Nacional de Información Biotecnológica (NCBI). Lo que hace esta herramienta es usar sintaxis de URL fija, cuyos parámetros se traducen en los valores necesarios para realizar la búsqueda en las bases del NCBI y devolver el resultado.

De las 8 funciones o programas que ofrece, nosotros vamos a hacer uso del ***e-search*** y del ***e-fetch***. Con la primera función, hacemos la búsqueda de los IDs de las entradas de la base de datos, GenBank en nuestro caso, que satisfacen los parámetros introducidos; y con el segundo, accedemos a la información de las entradas.

Por ejemplo, si queremos buscar aquellas entradas que tenga en su título la frase *Human p53 transformation suppressor*, primero hacemos la búsqueda con *e-search* de la siguiente manera:

https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?db=nuccore&term=Human+p53+transformation+suppressor[title]&rettype=gb&retmode=xml

Esta búsqueda nos desvolverá un archivo XML con los IDs de aquellas entradas que satisfagan la condición que hemos indicado (Figura 8).

```
▼<eSearchResult>
   <Count>11</Count>
   <RetMax>11</RetMax>
   <RetStart>0</RetStart>
  ▼<IdList>
     <Id>506452</Id>
     <Id>506450</Id>
     <Id>506448</Id>
     <Id>506446</Id>
     <Id>506444</Id>
     <Id>506442</Id>
     <Id>506440</Id>
     <Id>506438</Id>
     <Id>506434</Id>
     <Id>506436</Id>
     <Id>506432</Id>
   </IdList>
   <TranslationSet/>
  ▼<TranslationStack>
    ▼<TermSet>
       <Term>Human[Title]</Term>
       <Field>Title</Field>
       <Count>3905779</Count>
       <Explode>N</Explode>
     </TermSet>
    ▼<TermSet>
       <Term>p53 transformation suppressor[title]</Term>
       <Field>title</Field>
       <Count>12</Count>
       <Explode>N</Explode>
     </TermSet>
     <OP>AND</OP>
     <OP>GROUP</OP>
   </TranslationStack>
  ▼<QueryTranslation>
     Human[Title] AND p53 transformation suppressor[title]
   </QueryTranslation>
 </eSearchResult>
```

Figura 8: *Resultado de e-search*

Para cada uno de estos resultados, se hará un *e-fetch* de la siguiente manera:

https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nuccore&id=506452&rettype=native&retmode=xml

De esta manera ya obtenemos la información detallada para cada entrada de GenBank. En el caso de que sepamos el ID o el Locus, nos podemos saltar el paso de *e-search*.

3.3.2. OMIM

Para el caso de OMIM, igual que hemos hecho para GenBank, hacemos una primera búsqueda en la nube de Linked Open Data.

Obtenemos tres resultados para esta base de datos. Los dos primeros hacen referencia a OMIM en *Bio2RDF*, por lo que los descartamos de momento ya que, como hemos dicho antes, se suelen encontrar desactualizados.

El tercer resultado ofrece información en ***BioPortal*** [16], el mayor repositorio de ontologías médicas del mundo. En la figura 9 podemos observar lo que nos encontramos al acceder al enlace [17] que nos porporciona Linked Open Data para OMIM.

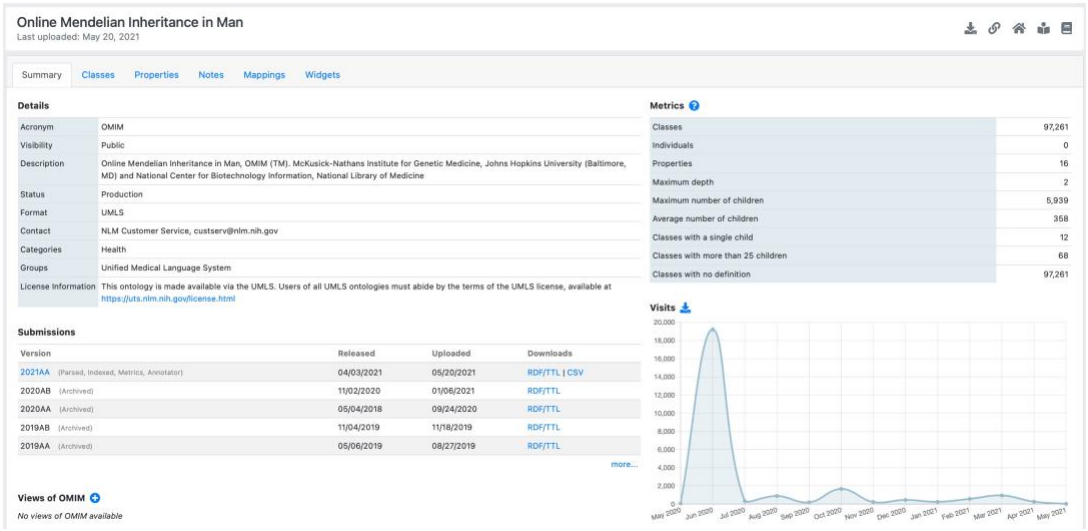


Figura 9: *Información de OMIM en BioPortal*

Nos aparece información sobre la base de datos en cuestión y distintas versiones de la ontología en formato RDF/TTL y, en el caso de la última, también formato CSV.

Por tanto, optamos por descargar la ontología de la última versión, que podemos ver bastante actualizada, y trabajar con ella directamente en *Virtuoso* a través de consultas SPARQL.

3.3.3. Uniprot

En el caso de Uniprot, la búsqueda de información se simplifica notablemente. No ha sido necesario investigar en Linked Open Data Cloud la existencia de algún repositorio RDF porque la propia página web de la base de datos lo facilita.

Uniprot cuenta con una herramienta propia [18] para realizar consultas SPARQL y obtener la información requerida de la base de datos (Figura 10).

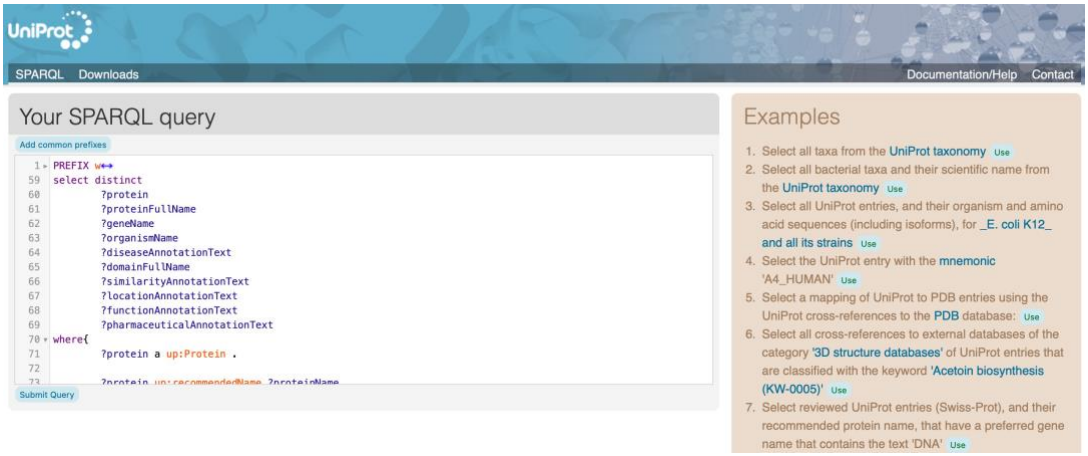


Figura 10: *Endpoint SPARQL de Uniprot*

Por tanto, utilizaremos este Endpoint para consultarlo desde el programa de *Python* que diseñaremos.

4

Requisitos

Al inicio de cualquier proyecto, es esencial determinar qué requisitos va a cumplir nuestra herramienta y dejar claro qué pretendemos ofrecer con ella. Esto es, indicar qué esperamos que haga nuestro sistema y también qué no esperamos que haga. Esta parte del proyecto se conoce como *análisis de requisitos*.

Los requisitos son aquellas condiciones que debe cumplir la herramienta o cualidades que debe poseer para que tanto las necesidades de los usuarios como los intereses de la entidad para la que se realice el proyecto se vean satisfechos.

Estos requisitos se pueden clasificar en base a distintos criterios. La clasificación que vamos a adoptar nosotros es en base a su naturaleza. Esta clasificación comprende:

- **Requisitos funcionales**

  Son aquellos que especifican,de manera concisa, qué es lo que debe hacer el proyecto para que se cumplan las expectativas que se desean.

- **Requisitos no funcionales**

  Estos tratan de describir las restricciones que debe cumplir también el proyecto para que el fin no se vea afectado.

En este proyecto, vamos a distinguir entre los requisitos que debe cumplir nuestra aplicación web y los que debe satisfacer el acceso a los datos RDF.

4.1. Aplicación Web

4.1.1. Requisitos funcionales

- Buscar información de enfermedades genéticas que cumplan los criterios proporcionados.

- **Identificador único.**

  RF-0001

- **Descripción.**

  Un usuario podrá buscar información sobre enfermedades genéticas proporcionando su ID, su nombre, el gen locus, el nombre de algún gen relacionado o los síntomas que tiene.

- **Prioridad.**

  Prioridad Alta.

- Buscar información de genes que cumplan los criterios proporcionados.

  - **Identificador único.**

    RF-0002

  - **Descripción.**

    Un usuario podrá buscar información sobre genes proporcionando su ID, su nombre, su locus, el organismo que lo posee o alguna palabra clave relacionada con él.

  - **Prioridad.**

    Prioridad Alta.

- Buscar información de proteínas que cumplan los criterios proporcionados.

  - **Identificador único.**

    RF-0003

  - **Descripción.**

    Un usuario podrá buscar información sobre proteínas proporcionando su ID, su nombre, algún gen asociado, el organismo en el que se encuentra o alguna anotación de enfermedad, funcional, de similitud o localización.

  - **Prioridad.**

    Prioridad Alta.

- Buscar información de genes a partir de una enfermedad en estudio.

- **Identificador único.**

  RF-0004

- **Descripción.**

  Un usuario podrá buscar información sobre aquellos genes que estén relacionados con la enfermedad genética de estudio.

- **Prioridad.**

  Prioridad Media.

- Buscar información de genes a partir de una proteína en estudio.

  - **Identificador único.**

    RF-0005

  - **Descripción.**

    Un usuario podrá buscar información sobre aquellos genes que estén relacionados con la proteína de estudio.

  - **Prioridad.**

    Prioridad Media.

- Consolidar en una base de datos RDF los datos que no estén accesibles en un SPARQL Endpoint.

  - **Identificador único.**

    RF-0006

  - **Descripción.**

    Todos aquellos datos para los que no se disponga de un SPARQL Endpoint, serán introducidos en una base de datos RDF.

  - **Prioridad.**

    Prioridad Alta.

- Diseñar casos de uso prácticos de consultas federadas.

  - **Identificador único.**

    RF-0007

- **Descripción.**

  Se realizará el diseño de casos de uso para ilustrar el funcionamiento de las consultas federadas.

- **Prioridad.**

  Prioridad Media.

4.1.2. Requisitos no funcionales

- Acceso gratuito y sin previo registro.

  - **Identificador único.**

    RNF-0001

  - **Descripción.**

    Cualquier usuario que desee consultar datos sobre genes, proteínas o enfermedades, podrá hacer uso de la herramienta sin necesidad de registrarse.

- Alta usabilidad para permitir a científicos explorar los datos.

  - **Identificador único.**

    RNF-0002

  - **Descripción.**

    Los procesos de la aplicación deben ser lo más sencillos y rápidos posibles, para que los usuarios encuentren cómodo su uso.

4.2. Acceso a RDF

4.2.1. Requisitos funcionales

- Accesible por un SPARQL Endpoint.

  - **Identificador único.**

    RF-0008

  - **Descripción.**

    Los datos en formato RDF deben estar accesibles mediante un SPARQL Endpoint para poder consultarlos.

- **Prioridad.**

  Prioridad Alta.

- Datos tan actualizados como sea posible.

  - **Identificador único.**

    RF-0009

  - **Descripción.**

    Los datos consultados deben estar actualizados para que la información obtenida sea de máxima confianza.

  - **Prioridad.**

    Prioridad Media.

- Datos tan actualizados como sea posible.

  - **Identificador único.**

    RF-0009

  - **Descripción.**

    Los datos consultados deben estar actualizados para que la información obtenida sea de máxima confianza.

  - **Prioridad.**

    Prioridad Media.

- Accesible desde consultas federadas.

  - **Identificador único.**

    RF-00010

  - **Descripción.**

    Los datos en RDF también deben estar accesibles al realizar consultas federadas.

  - **Prioridad.**

    Prioridad Alta.

5
Diseño

5.1. Diagrama de casos de uso

Después de establecer los requisitos funcionales y no funcionales que pretendemos que cubra nuestra plataforma, es conveniente determinar los casos de uso. Un caso de uso es la descripción de lo que puede hacer un usuario con las funcionalidades que le ofrece la aplicación desarrollada en el proyecto.

En nuestro caso, solo contamos con un tipo de usuario, que será cualquier persona que acceda a la plataforma a través de la web de manera gratuita. Este usuario pasará a denominarse *actor*.

Ahora queda determinar qué acciones podrá realizar el usuario, lo que es en sí el modelado de casos de uso, que se puede ver en la Figura 11.

En primer lugar, nos encontraremos con una página de inicio, donde aparecen tres accesos, uno para cada base de datos.

Si accedemos a Uniprot, podremos realizar la búsqueda proporcionando el ID de la proteína o, por el contrario, añadiendo otros filtros. Cuando se haya realizado la búsqueda y devuelto la información de la proteína o proteínas en cuestión, podremos acceder también a cualquier gen que venga asociado a la proteína o proteínas en estudio. La sección dedicada a OMIM funciona de igual forma. En el caso de acceder a GenBank, solo podremos realizar las búsquedas, tanto por ID como por otro filtro que elijamos.

En cualquiera de las tres secciones aparecerá la opción de volver a la página de inicio.

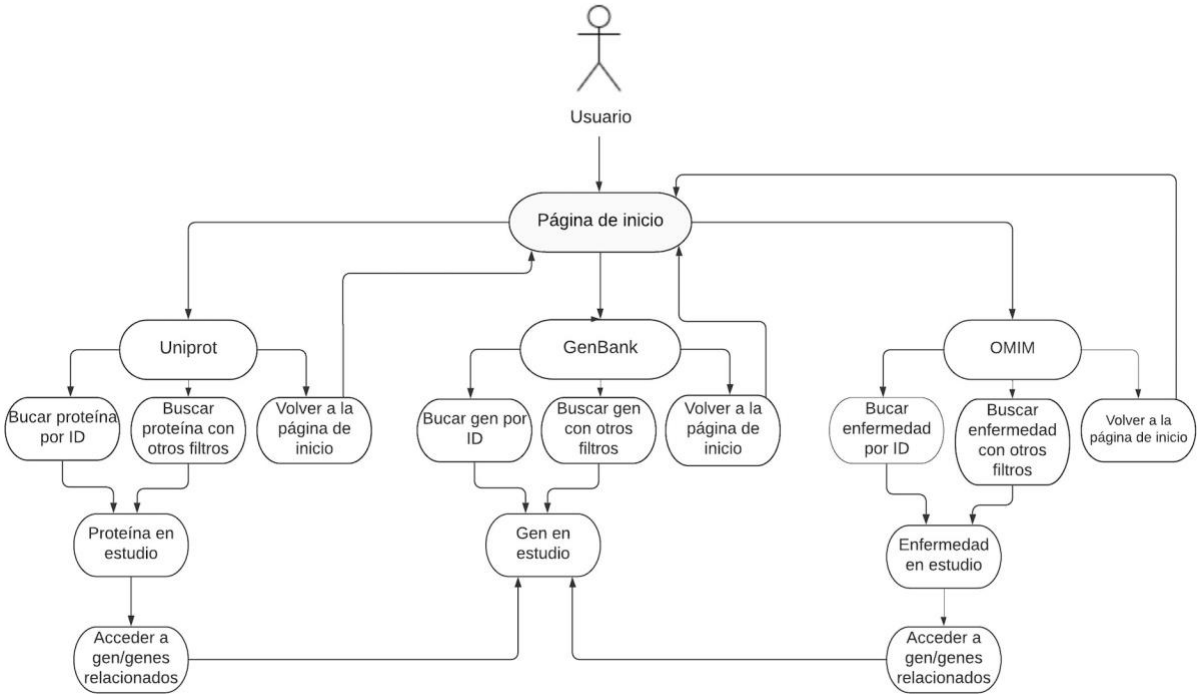


Figura 11: *Diagrama de casos de uso*

5.2. Diagramas de secuencias

Una vez hemos determinado qué es lo que podrá hacer el usuario con las funcionalidades y características que ofrece nuestra plataforma, el siguiente paso es definir cómo será la interacción cuando se hace uso de la misma entre los componentes que la conforman (interfaz, API, repositorio Virtuoso, Gene Downloader y Uniprot SPARQL Endpoint). Para explicar estas interacciones, vamos a hacer uso de los diagramas de secuencias.

Como nuestra aplicación permite el acceso a tres bases de datos distintas y cada una tiene dos tipos de búsquedas (por ID y por filtros), definiremos las relaciones entre los distintos componentes de la plataforma en 6 diagramas de secuencias (Figuras 12, 13, 16, 17, 14 y 15).

5.2.1. GenBank

- **BÚSQUEDA POR ID**.

  En la Figura 12, se muestra la búsqueda de un gen a través de su identificador. En primer

lugar, debemos acceder a la página dedicada a *GenBank* desde la página de inicio. Una vez en la página dedicada a esa base de datos, introducimos el ID del gen que deseamos estudiar. Se realiza una búsqueda previa en el repositorio Virtuoso por si ese gen ya ha sido buscado con anterioridad y se encuentra almacenado; en caso contrario, se hace la búsqueda y descarga directamente en GenBank a través de las *e-utilities*. La información obtenida se pasa a la API y de esta a la interfaz donde el usuario podrá visualizarla. Estos datos se consolidan además en Virtuoso para futuras búsquedas.

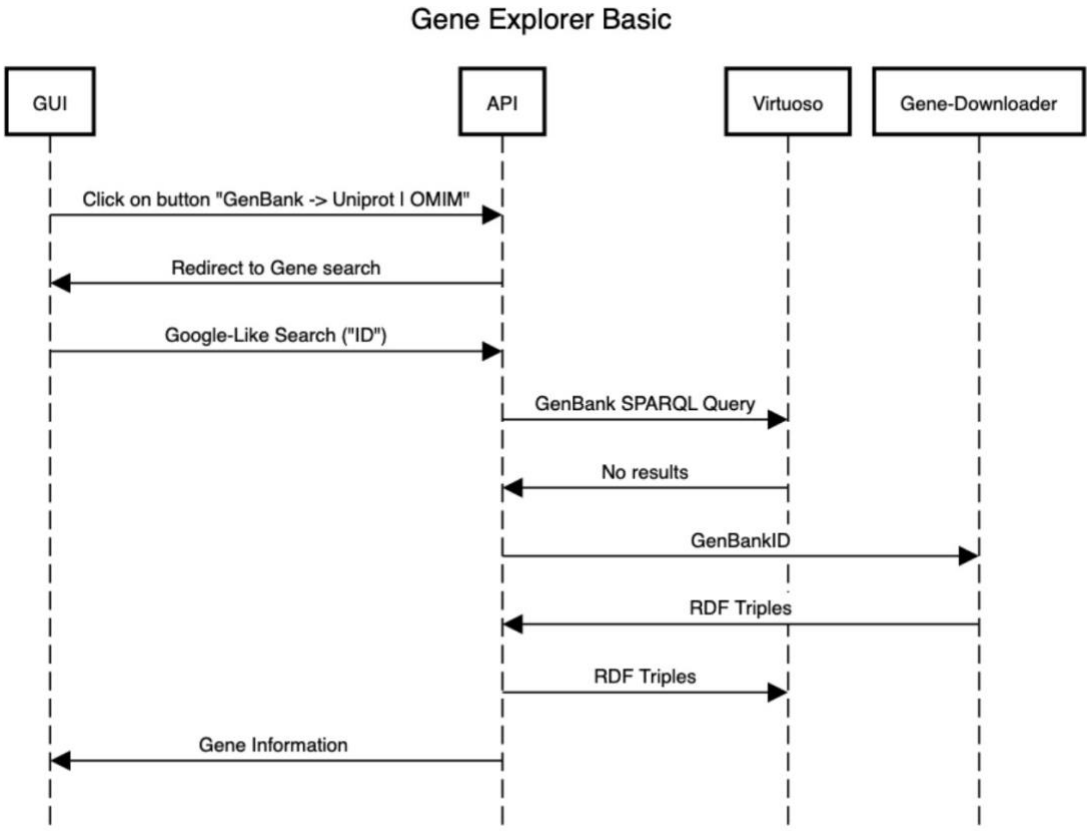


Figura 12: *Búsqueda de un gen a través de su ID.*

- **BÚSQUEDA POR FILTROS**.

  En la Figura 13 se detalla la búsqueda a través de otros filtros, sin hacer uso del ID concreto. En este caso, como en el anterior, hay que acceder a la búsqueda en *GenBank* y, una vez nos encontremos en esa página, seleccionar la búsqueda con filtros. Se desplegará un formulario con los filtros que se pueden completar y, de igual manera se puede volver a cerrar ese formulario. Una vez introducido el término deseado, se realiza la bús-

queda, que seguirá el mismo procedimiento que con el identificador: búsqueda previa en Virtuoso, búsqueda y descarga en GenBank en caso de no estar en el repositorio, envío de datos a la API y, de aquí, a la interfaz. Estos datos se consolidan además en Virtuoso para futuras búsquedas.

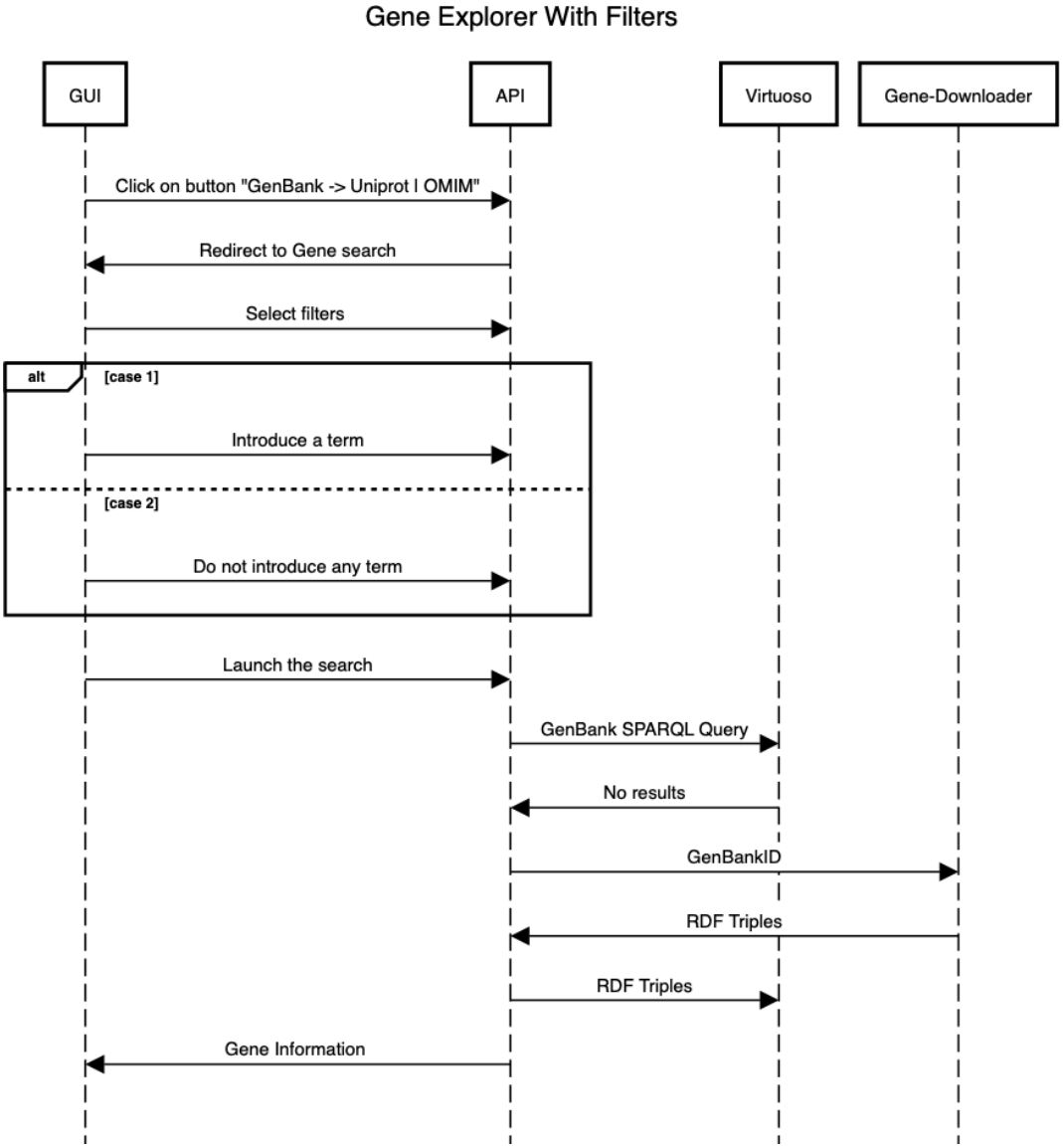


Figura 13: *Búsqueda de un gen a través de filtros.*

5.2.2. OMIM

- **BÚSQUEDA POR ID**.

  En la Figura 14, nos encontramos con la búsqueda por ID de una enfermedad. En primer lugar, debemos acceder a la sección de nuestra plataforma dedicada a *OMIM* desde la

página de inicio. Una vez ahí, introducimos el ID de la enfermedad de estudio. Lanzamos la búsqueda y accedemos al repositorio de OMIM que tenemos montado en Virtuoso con toda la información de esta base de datos. Realizamos la consulta con el ID proporcionado y esta consulta devuelve la información sobre la enfermedad y la lista de genes asociados a la API. Esta información es enviada desde la API a la interfaz. El usuario puede seleccionar cualquier gen para acceder a su información o no hacer nada si no lo desea.

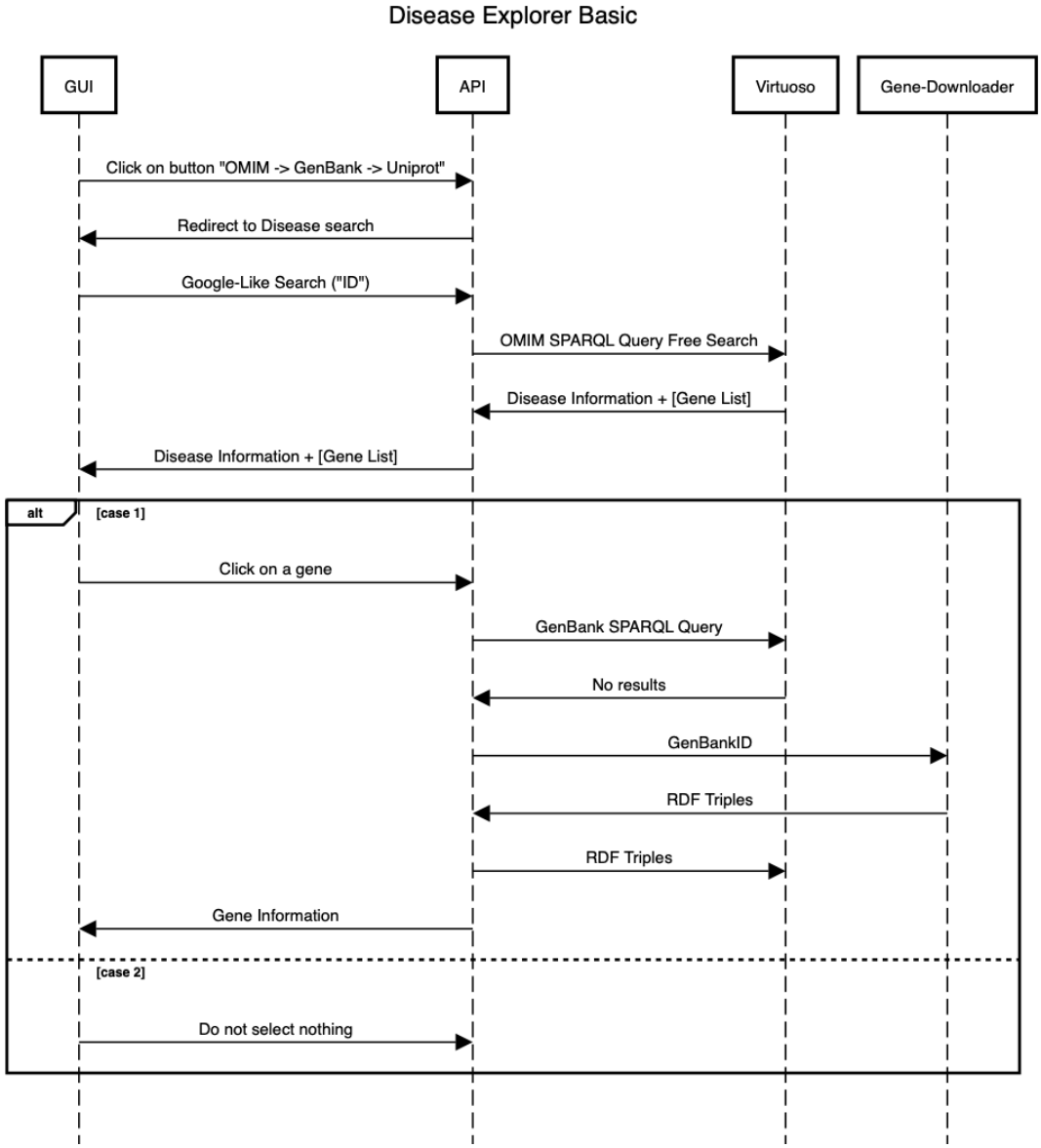


Figura 14: *Búsqueda de una enfermedad a través de su ID.*

- **BÚSQUEDA POR FILTROS**.

  En la imagen que aparece a continuación (Figura 15), se detalla la búsqueda de enfermedades a través de otros filtros, sin hacer uso del ID concreto. En este caso, como en el anterior, hay que acceder a la página habilitada para *OMIM* y, una vez nos encontremos en esa página, pulsar sobre el botón que nos mostrará los filtros disponibles. Se introduce el término o términos deseados y se lanza la búsqueda, que seguirá el mismo procedimiento que con el identificador.

5.2.3. Uniprot

- **BÚSQUEDA POR ID**.

  En la Figura 16, se muestra la búsqueda de una proteína a través de su identificador. Como primer paso, accedemos a la página de nuestra app dedicada a la búsqueda de proteínas. Una vez aquí, introducimos el ID de la proteína en el campo proporcionado para ello y realizamos la búsqueda. Como *Uniprot* cuenta con un SPARQL Endpoint propio accesible y actualizado, se realiza ahí la búsqueda y se devuelve la información en formato RDF a la API. Aquí se procesa y se pasa en el formato adecuado a la interfaz, con la información de la proteína en estudio y los genes asociados a ella. Como se devuelve también los genes asociados a la proteína en cuestión, podemos acceder también a la información sobre esos genes o, por el contrario, no hacer ninguna otra búsqueda. En caso de hacerla, el procedimiento de búsqueda es el mismo que el detallado en el apartado anterior.

- **BÚSQUEDA POR FILTROS**.

  En la siguiente imagen (Figura 17), se explica la búsqueda a través de otros filtros, sin hacer uso del identificador de la proteína. Accedemos a la página para la búsqueda en *Uniprot*, pulsamos el botón que nos permite desplegar los filtros y rellenamos aquellos que deseemos. Lanzamos la búsqueda y el resto del proceso es el mismo que en la búsqueda por ID.

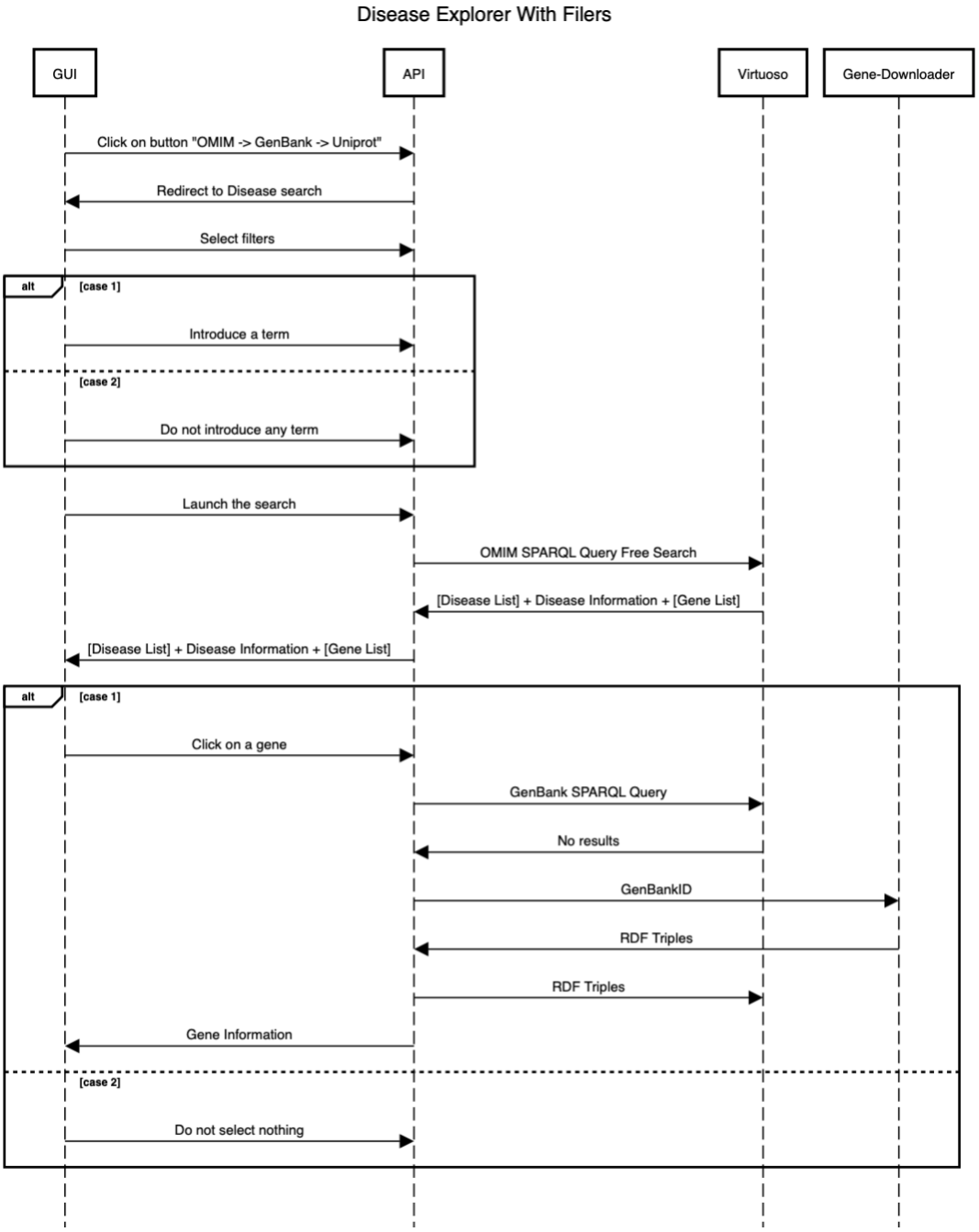


Figura 15: *Búsqueda de una enfermedad a través de filtros.*

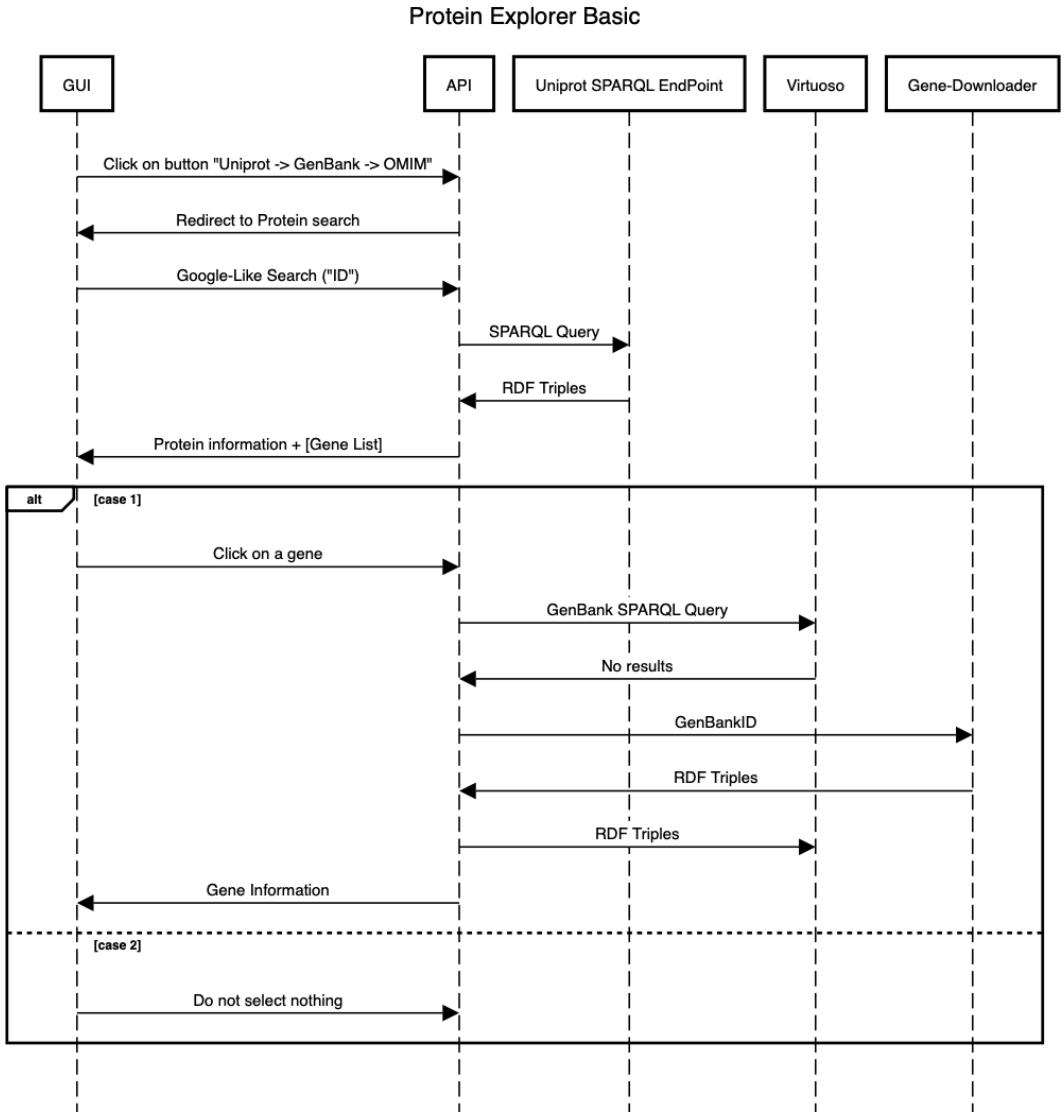


Figura 16: *Búsqueda de una proteína a través de su ID.*

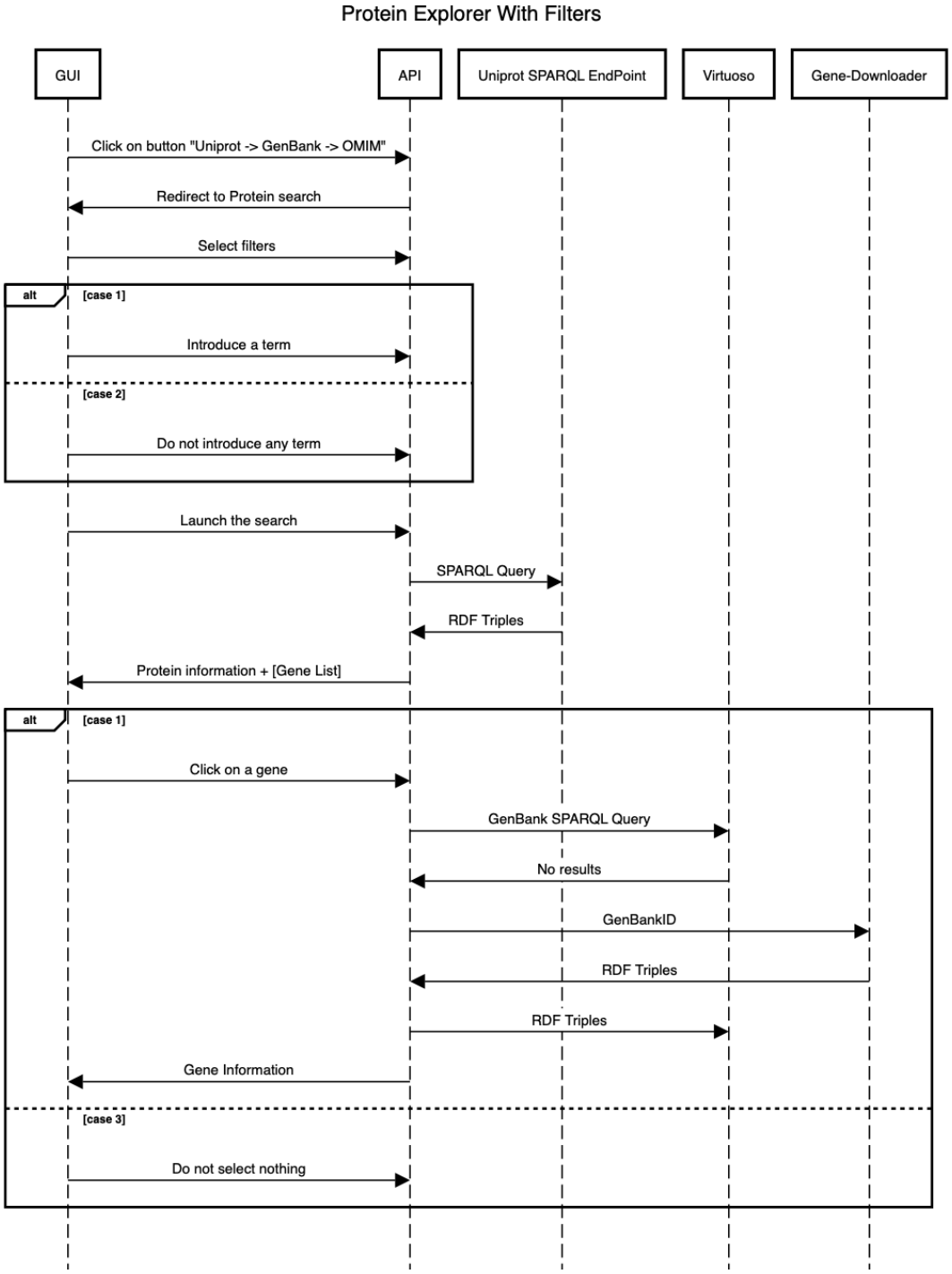


Figura 17: *Búsqueda de una proteína a través de filtros.*

6

Implementación

Para que se entienda mejor la estructura de nuestra aplicación y su funcionamiento, vamos a mostrar primero de manera esquemática la organización de nuestra plataforma, teniendo en cuenta frontend y backend (Figura 18). Detallaremos cada componente que forma parte de nuestra aplicación en las siguientes secciones.

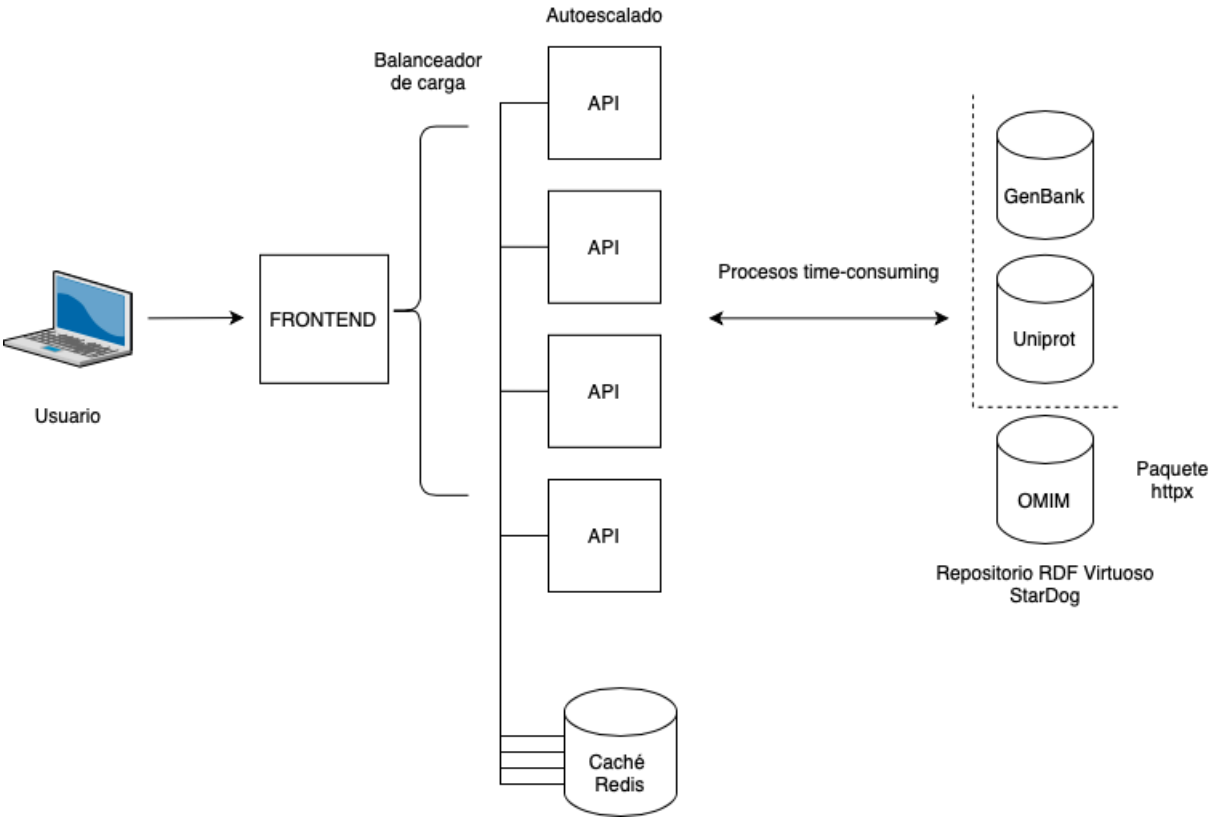


Figura 18: *Esquema global de la aplicación*

6.1. Aplicación Web

Como podemos observar en la figura 18, nuestra plataforma entra en funcionamiento cuando algún usuario acceda a ella a través de la interfaz, es decir, el *frontend*.

Para la construcción de la interfaz gráfica, hemos optado por *JavaScript* utilizando el framework **Next.js** [19]. Se trata de un framework de código abierto construido por Vercel sobre *React.js* [20]. React.js es, por su parte, una librería Javascript centrada principalmente en el desarrollo de interfaces.

Hemos elegido Next.js porque se trata de una herramienta fácil de utilizar a la vez que potente. Simplifica enormemente el trabajo inicial de configurar todo lo necesario para construir una interfaz en JavaScript ya que, instalando una sola dependencia, tienes configurado lo que necesitas para empezar a crear tu aplicación.

Este framework requiere tener una carpeta *pages* en la raíz del proyecto que estemos creando y cada archivo que esté dentro de esa carpeta va a ser una ruta a la que podremos acceder. Esto permite tener un código claro y entendible, al organizarlo por rutas. Cada una de estas rutas no es más que un archivo JavaScript que exportar un componente de React que puede, a su vez, importar más componentes.

Además, es muy fácil desplegar la aplicación una vez esté lista. Todas estas ventajas nos han llevado a decantarnos por Next.js. [21].

Cabe mencionar que el frontend es escalable, es decir, tiene la capacidad de adaptarse al volumen de llamadas que reciba para no perder calidad. Esto se debe a que se trata de un *protocolo sin estado* ***(stateless)***, por lo que cada petición que se hace es independiente de la anterior y de la siguiente, no se guarda la información. Servidores web como *Nginx* nginx y *Apache* [22] pueden ocuparse de esta tarea.

La interfaz se conecta con la *API* para enviar la petición del usuario. El desarrollo de la API se ha realizado con **FastAPI** [23] en *Python*.

FastAPI es un framework en Python diseñado para crear endpoints de API rápidamente, con una ejecución rápida también.

Entre sus características, se encuentra la de restringir sobre los tipos de datos que aceptan las rutas. Es decir, si una ruta tiene el tipo String ([str]), solo aceptará cadenas de caracteres [24].

Al igual que el frontend, la API es escalable al tratarse también de un protocolo sin estado. De tal manera que pueden encontrarse distintas instancias de la API levantadas y que, mediante un *balanceador de cargas*, se derive la petición a aquella que se encuentre menos ocupada. Un *balanceador de cargas* es un componente del hardware que se encarga de distribuir el tráfico de la red y/o de la aplicación en diferentes servidores [25].

Es importante mencionar que tanto el frontend como el backend se encuentran desarrollados por separado y se despliegan también por separado. Es decir, nuestra plataforma presenta una arquitectura de microservicios, compuesta por pequeños servicios que se ejecutan de manera independiente y autónoma [26]. Esto lo hace aún más escalable.

La propiedad del balanceo es conveniente tenerla por si en un futuro decidimos *dockerizar* el proyecto, de tal manera que el frontend se encuentre en un docker y el contenedor de la API en otro y puedan comunicarse. Existe un orquestador de dockers muy famoso llamado ***Kubernetes*** [27] que permite el balanceo, autoescalado, etc. Un orquestador de dockers se encarga de agrupar los contenedores (*dockers*) que componen una aplicación para gestionarla de manera más eficiente.

Como las peticiones a las distintas bases de datos son procesos **time-consuming**, es decir, que requieren mucho tiempo para realizarse, hemos añadido una caché a la que puede acceder cualquier instancia levantada de la API para aquellas peticiones que se realizan con mayor frecuencia. De esta manera, mejoramos el tiempo y, con ello, el rendimiento de nuestra aplicación. Para el almacenamiento en caché, se pueden utilizar herramientas como ***Redis*** [28], **Re**mote **Di**ctionary **S**erver, un almacén de datos clave-valor en memoria rápido y de código abierto.

Aún existiendo esta caché, sigue siendo posible el balanceo de cargas.

6.2. Acceso a RDF

A través de la API, la petición llega a la base de datos que corresponde. Las funciones de acceso y descarga de datos de cada una de ellas se encuentran separadas en distintos archivos Python.

6.2.1. Uniprot

En el caso de que el usuario mande la petición a través de la página dedicada a **Uniprot**, esa consulta debe llegar a esta base de datos. Usamos el paquete **SPARQLWrapper** de Python para el acceso al SPARQL Endpoint de Uniprot.

Desde el frontend, el usuario puede proporcionar el ID o cualquiera de los otros campos, pudiendo completarlos todos o solo uno. Si lo que proporciona es el ID, carece de sentido que siga proporcionando más campos ya que el ID es único y no va a haber error. Es más, puede introducir mal cualquiera de los otros campos y que la búsqueda sea errónea.

La función creada para la consulta del SPARQL Endpoint debe recibir todos los parámetros, aunque el usuario no los haya completado todos. Por tanto, proporcionamos un valor por defecto de un string vacío a todos los parámetros y, si el usuario lo completa, cambia su valor. Así la función los recibe todos e, internamente, se encarga de comprobar cuál no ha sido rellenado.

Esta función construye una consulta SPARQL, donde se busca la URL de la proteína, el nombre de la misma, el nombre del gen, el nombre del organismo, la anotación de localización, la anotación de función, la anotación de similitudes y la anotación de enfermedad.

Cuando se completa un campo, se aplica un *filter* para esa propiedad con el parámetro que se haya introducido; en caso contrario, se hace una búsqueda de todos los posibles nodos que haya en el grafo de Uniprot (Figura 19). Al tener que buscar en todos los posibles nodos cuando no se proporciona un campo, el tiempo de ejecución de la consulta aumenta en gran medida, por lo que hemos tenido que añadir un *LIMIT 20* para que no sea un tiempo excesivo.

El proceso para la construcción de la consulta se puede ver en la figura 20.

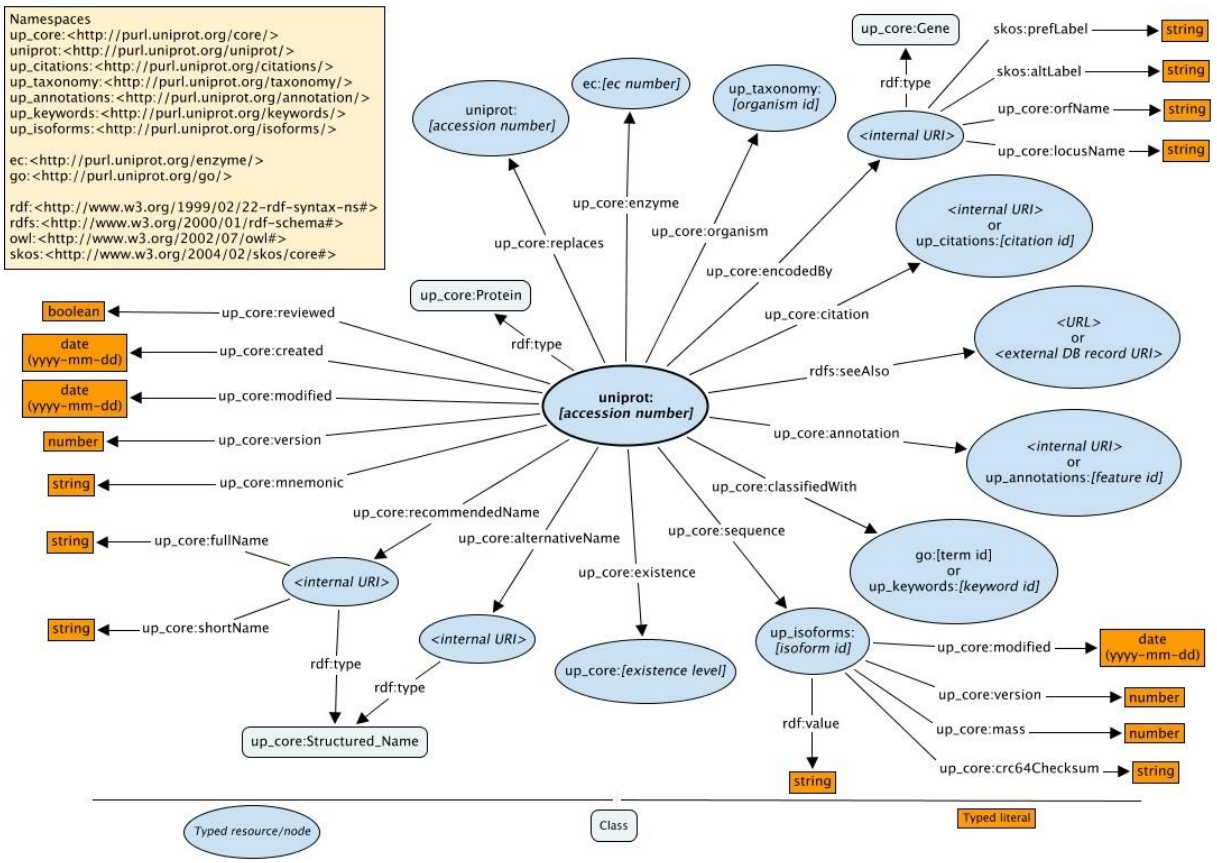


Figura 19: *RDF Schema de Uniprot*

```
query += "select distinct \n"
query += "  ?protein\n"
query += "  ?proteinFullName\n"
query += "  ?geneName\n"
query += "  ?organismName\n"
query += "  ?diseaseAnnotationText\n"
query += "  ?domainFullName\n"
query += "  ?similarityAnnotationText\n"
query += "  ?locationAnnotationText\n"
query += "  ?functionAnnotationText\n"
query += "  ?pharmaceuticalAnnotationText\n"
query += "where{\n"

query += "  ?protein a up:Protein .\n"

if (proteinId != ''):
    query += "  VALUES ?protein {uniprotkb:"+ proteinId + "}\n"

query += "\n"

if (proteinName == ''):
    query += "  OPTIONAL {\n"
query += "  ?protein up:recommendedName ?proteinName .\n"
query += "  ?proteinName up:fullName ?proteinFullName . \n"
if (proteinName !=''):
    query += "  filter( regex(str(?proteinFullName), " + '"' + proteinName + '"' + ",\"i\" )) .\n"
if (proteinName == ''):
    query += "  }\n"
query += "\n"

if (geneName == ''):
    query += "  OPTIONAL {\n"
query += "  ?protein up:encodedBy ?gene .\n"
query += "  ?gene skos:prefLabel ?geneName .\n"
if (geneName != ''):
    query += "  filter( regex(str(?geneName), " + '"' + geneName + '"' + ",\"i\" )) .\n"
if (geneName == ''):
    query += "  }\n"

query += "\n"

if (organismName == ''):
    query += "  OPTIONAL {\n"
query += "  ?protein up:organism ?organism .\n"
query += "  ?organism up:scientificName ?organismName .\n"
if (organismName != ''):
    query += "  filter( regex(str(?organismName), " + '"' + organismName + '"' + ",\"i\" )) .\n"
if (organismName == ''):
    query += "  }\n"

query += "\n"
```

Figura 20: *Proceso de construcción de la consulta SPARQL en Uniprot.*

Si el usuario introduce, por ejemplo, el concepto *Casein kinase* para el nombre de la proteína

y *Homo sapiens* para el organismo, se construye la consulta de la figura 21.

```
select distinct
  ?protein
        ?proteinFullName
        ?geneName
        ?organismName
        ?diseaseAnnotationText
        ?domainFullName
        ?similarityAnnotationText
        ?locationAnnotationText
        ?functionAnnotationText
        ?pharmaceuticalAnnotationText
where{
        ?protein a up:Protein .

        ?protein up:recommendedName ?proteinName .
        ?proteinName up:fullName ?proteinFullName .
        filter( regex(str(?proteinFullName), "Casein kinase","i" )) .

        OPTIONAL {
        ?protein up:encodedBy ?gene .
        ?gene skos:prefLabel ?geneName .
        }

        ?protein up:organism ?organism .
        ?organism up:scientificName ?organismName .
        filter( regex(str(?organismName), "Homo sapiens","i" )) .

        OPTIONAL {
        ?protein up:annotation ?diseaseAnnotation .
        ?diseaseAnnotation a up:Disease_Annotation .
        ?diseaseAnnotation up:disease ?disease .
        ?disease rdfs:comment ?diseaseAnnotationText
        }

        OPTIONAL {
        ?protein up:domain ?domain .
        ?domain up:recommendedName ?domainName .
        ?domainName up:fullName ?domainFullName .
        }

        OPTIONAL {
        ?protein up:annotation ?similarityAnnotation .
        ?similarityAnnotation a up:Similarity_Annotation .
        ?similarityAnnotation rdfs:comment ?similarityAnnotationText .
        }
```

Figura 21: *Ejemplo de consulta SPARQL en Uniprot.*

Creamos una objeto del tipo SPARQLWrapper al que le indicamos la dirección del servicio que debe recibir las consultas y responderlas (https://sparql.uniprot.org/sparql), le pasamos la consulta que hemos montado con los parámetros introducidos, indicamos el formato en el que queremos que nos devuelva los resultados (*JSON*) y lanzamos la consulta.

Hacemos una conversión del resultado al formato elegido, ya que con lo que hemos indicado antes hacemos que los datos se descarguen en ese formato pero en un documento de texto, por lo que hay que pasarlo al formato JSON como tal.

Finalmente, se trata ese resultado ya en JSON a través de una función para que cada proteína sea un diccionario con tantas claves como elementos se consultan en el SELECT de la petición. Esta función también va añadiendo cada uno de los diccionarios que se obtienen a una lista.

La lista se devuelve como resultado a través de la API al frontend, donde se visualiza en forma de tabla, con una fila por cada proteína y una columna para cada uno de las claves del diccionario.

6.2.2. OMIM

Si, por el contrario, la petición es enviada desde la sección dedicada a **OMIM**, entonces deberá llegar la consulta a esta base de datos.

En este caso, no disponemos de un Endpoint ya construido al que acceder mediante consultas como en Uniprot, pero sí hemos podido obtener la ontología completa bastante actualizada. Por tanto, en primer lugar, creamos el repositorio para OMIM en Virtuoso y subimos manualmente el archivo.

Para eso, accedemos a *Virtuoso Conductor*, que lo hemos debido de instalar previamente, e introducimos las credenciales que estableciésemos en el momento de la instalación en la página de la figura 22.

Figura 22: *Página de inicio de Virtuoso Conductor.*

Al entrar ya con el usuario y la contraseña, la página es la misma que la de inicio pero en esta ya podemos acceder a las entradas de la parte superior (figura 23).

Figura 23: *Página principal Virtuoso Conductor.*

Para subir el archivo **ttl** con la ontología, accedemos a la pestaña *Linked Data* y, dentro de esta a *Quad Store Upload*. Aquí nos aparece la opción de subir un archivo y de darle el nombre que queramos a nuestro grafo. Ese nombre será el nombre de nuestra base de datos de OMIM en Virtuoso (Figura 24).

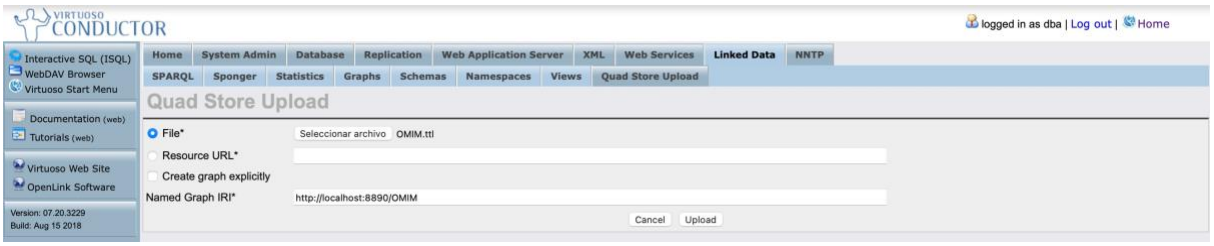


Figura 24: *Subida de la ontología de OMIM a Virtuoso.*

Para el código de la API hemos hecho uso del paquete de Python **httpx**, un cliente HTTP que proporciona APIs tanto síncronas como asíncronas y que nos permitirá acceder a Virtuoso.

El usuario puede buscar una enfermedad a través de su ID o mediante cualquier otro filtro, como ocurre con las proteínas. Puede completar uno o todos los filtros en cada búsqueda.

La función que recibe los parámetros a través de la API los requiere todos. Por tanto, como hacemos con Uniprot y veremos con GenBank, proporcionamos en el frontend un valor por defecto de un string vacío a todos los parámetros para que, en el caso de que no se complete ese campo, sí llegue el parámetro a la función.

Definimos los parámetros de conexión con el repositorio de OMIM en Virtuoso, que son el endpoint (`http://localhost:8890/sparql-auth/`), la base de datos (`http://localhost:8890/OMIM`), el usuario y la contraseña. Establecemos conexión con estos parámetros.

Construimos la consulta SPARQL pidiendo el ID de la enfermedad, el nombre, el nombre del gen, el locus del gen y los síntomas. La construcción de la consulta SPARQL se puede ver en la figura 25.

```
consulta += "select distinct \n"
consulta += "  ?id\n"
consulta += "  ?name\n"
consulta += "  ?geneSymbol\n"
consulta += "  ?geneLocus\n"
consulta += "  ?symptom\n"
consulta += "where{{\n"

consulta += " ?indv umls:hasSTY ?sty .\n"
consulta += " ?sty skos:prefLabel ?labelSty.\n"
consulta += " filter(regex(lcase(?labelSty)," + '"' + DISEASE + '"' + ")).\n"

consulta += " ?indv skos:notation ?id.\n"
if (id != ''):
    consulta += " filter(regex(lcase(?id), " + '"' + id + '"' + ")).\n"
consulta += "\n"

consulta += " ?indv skos:prefLabel ?name.\n"
if (name != ''):
    consulta += " filter(regex(lcase(?name), " + '"' + name + '"' + ",\"i\" )).\n"
consulta += "\n"

consulta += " ?indv omim:has_manifestation ?hasMan.\n"
consulta += " ?hasMan skos:prefLabel ?symptom.\n"
if (symptom != ''):
    consulta += " filter(regex(lcase(?symptom), " + '"' + symptom + '"' + ",\"i\" )).\n"
consulta += "\n"

consulta += " ?indv omim:GENELOCUS ?geneLocus.\n"
if (symptom != ''):
    consulta += " filter(regex(lcase(?geneLocus), " + '"' + geneLocus + '"' + ",\"i\" )).\n"
consulta += "\n"

consulta += " ?indv omim:GENESYMBOL ?geneSymbol.\n"
if (symptom != ''):
    consulta += " filter(regex(lcase(?geneSymbol), " + '"' + geneSymbol + '"' + ",\"i\" )).\n"
consulta += "\n"

consulta += "}}\n"
```

Figura 25: *Proceso de construcción de la consulta SPARQL en OMIM.*

Para aquellos parámetros que sí hayan sido proporcionados aplicaremos un *filter* y, en caso de ser un string vacío, haremos una búsqueda de todas las posibles combinaciones de nodos que haya en el grafo (Figura 26). Como podemos observar en la figura, nos encontramos ante una ontología de gran volumen, con numerosos nodos, lo cual dificulta enormemente el trabajo con ella y la construcción de cualquier consulta.

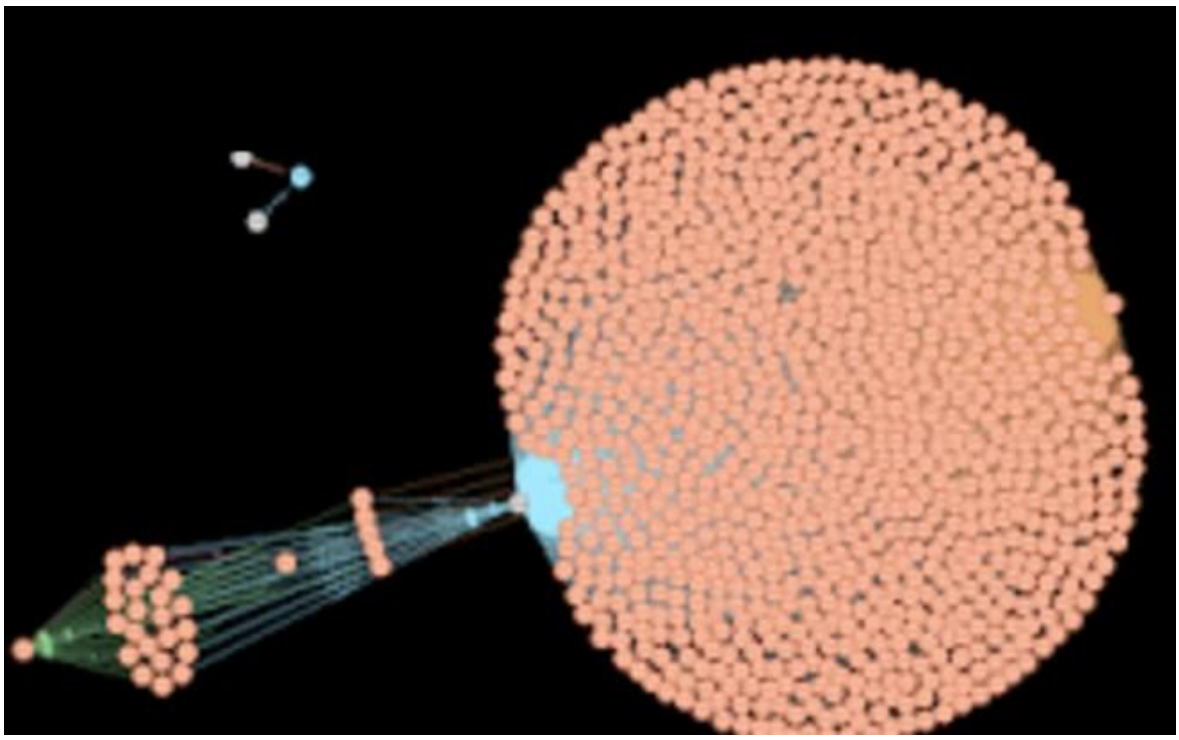


Figura 26: *RDF Schema de OMIM*

En la figura 27 podemos ver el código de la consulta en OMIM para la enfermedad de *Alzheimer*. Aparece un filter que establece que la búsqueda solo va a hacerse para aquellas entradas que estén clasificadas como *disease or syndrome* y ese filtro es fijo siempre.

```
select distinct
  ?id
  ?name
  ?geneSymbol
  ?geneLocus
  ?symptom
where{
 ?indv umls:hasSTY ?sty .
 ?sty skos:prefLabel ?labelSty.
 filter(regex(lcase(?labelSty),"disease or syndrome")).
 ?indv skos:notation ?id.

 ?indv skos:prefLabel ?name.
 filter(regex(lcase(?name), "Alzheimer","i" )).

 ?indv omim:has_manifestation ?hasMan.
 ?hasMan skos:prefLabel ?symptom.

 ?indv omim:GENELOCUS ?geneLocus.

 ?indv omim:GENESYMBOL ?geneSymbol.

}
```

Figura 27: *Ejemplo de consulta SPARQL en OMIM*.

Esta misma consulta podemos probarla en Virtuoso, en la pestaña *SPARQL* dentro de *Linked Data* (Figura 28). Se pueden observar múltiples resultados para un mismo ID de la enfermedad. Esto se debe a que una enfermedad tiene más de un síntoma y gen asociados.

Query

```
select distinct
  ?id
  ?name
  ?geneSymbol
  ?geneLocus
  ?symptom
where{
?indv umls:hasSTY ?sty .
?sty skos:prefLabel ?labelSty.
filter(regex(lcase(?labelSty),"disease or syndrome")).
?indv skos:notation ?id.

?indv skos:prefLabel ?name.
filter(regex(lcase(?name), "Alzheimer","i" )).
```

Execute Save Load Clear

id	name	geneSymbol	geneLocus	symptom
"104300"^^<http://www.w3.org/2001/XMLSchema#string>	"ALZHEIMER DISEASE"@en	"HFE"^^<http://www.w3.org/2001/XMLSchema#string>	"6p21.3"^^<http://www.w3.org/2001/XMLSchema#string>	"Presenile and senile dementia"@en
"104300"^^<http://www.w3.org/2001/XMLSchema#string>	"ALZHEIMER DISEASE"@en	"HFE1"^^<http://www.w3.org/2001/XMLSchema#string>	"6p21.3"^^<http://www.w3.org/2001/XMLSchema#string>	"Presenile and senile dementia"@en
"104300"^^<http://www.w3.org/2001/XMLSchema#string>	"ALZHEIMER DISEASE"@en	"HLA-H"^^<http://www.w3.org/2001/XMLSchema#string>	"6p21.3"^^<http://www.w3.org/2001/XMLSchema#string>	"Presenile and senile dementia"@en
"104300"^^<http://www.w3.org/2001/XMLSchema#string>	"ALZHEIMER DISEASE"@en	"MVCD7"^^<http://www.w3.org/2001/XMLSchema#string>	"6p21.3"^^<http://www.w3.org/2001/XMLSchema#string>	"Presenile and senile dementia"@en
"104300"^^<http://www.w3.org/2001/XMLSchema#string>	"ALZHEIMER DISEASE"@en	"TFQTL2"^^<http://www.w3.org/2001/XMLSchema#string>	"6p21.3"^^<http://www.w3.org/2001/XMLSchema#string>	"Presenile and senile dementia"@en
"104300"^^<http://www.w3.org/2001/XMLSchema#string>	"ALZHEIMER DISEASE"@en	"HFE"^^<http://www.w3.org/2001/XMLSchema#string>	"6p21.3"^^<http://www.w3.org/2001/XMLSchema#string>	"Genetic heterogeneity"@en
"104300"^^<http://www.w3.org/2001/XMLSchema#string>	"ALZHEIMER DISEASE"@en	"HFE1"^^<http://www.w3.org/2001/XMLSchema#string>	"6p21.3"^^<http://www.w3.org/2001/XMLSchema#string>	"Genetic heterogeneity"@en
"104300"^^<http://www.w3.org/2001/XMLSchema#string>	"ALZHEIMER DISEASE"@en	"HLA-H"^^<http://www.w3.org/2001/XMLSchema#string>	"6p21.3"^^<http://www.w3.org/2001/XMLSchema#string>	"Genetic heterogeneity"@en
"104300"^^<http://www.w3.org/2001/XMLSchema#string>	"ALZHEIMER DISEASE"@en	"MVCD7"^^<http://www.w3.org/2001/XMLSchema#string>	"6p21.3"^^<http://www.w3.org/2001/XMLSchema#string>	"Genetic heterogeneity"@en
"104300"^^<http://www.w3.org/2001/XMLSchema#string>	"ALZHEIMER DISEASE"@en	"TFQTL2"^^<http://www.w3.org/2001/XMLSchema#string>	"6p21.3"^^<http://www.w3.org/2001/XMLSchema#string>	"Genetic heterogeneity"@en

Figura 28: *Ejemplo de consulta SPARQL en OMIM en Virtuoso.*

Una vez tenemos el resultado, en formato JSON, debemos tratarlo con otra función para que cada enfermedad sea un diccionario que tenga como claves todos los elementos requeridos en el SELECT. Todos estos diccionarios se van introduciendo en una lista en la misma función. Como hemos visto, una enfermedad con ID único aparece más de una vez, y para evitar tanta repetición hemos tenido que utilizar otra función que recorra los diccionarios, compruebe su ID, los agrupe según el mismo y unifique los síntomas. Los genes no pueden unirse en una sola celda porque al representarlos en el frontend se hace asignándole un link que te lleva a la búsqueda de ese gen en la sección de GenBank, y esto no puede hacerse si están todos en la misma celda. Todo esto dificulta bastante el proceso.

Esta lista se envía como resultado al frontend donde se visualiza en forma de tabla, con una fila por cada enfermedad y una columna por cada clave del diccionario.

6.2.3. GenBank

Por último, si la petición enviada por el usuario a través del frontend a la API es desde la página dedicada a **GenBank**, esa petición deberá llegar a esta base.

En este caso, como ya comentamos en el capítulo 3, accedemos a la información a través de las **E-utilities**, con las funciones *e-search* y *e-fetch*.

Para GenBank, el usuario puede proporcionar el ID, el locus del gen, el nombre del gen, el organismo o una descripción, pero solo puede completar un campo en cada búsqueda. Por

tanto, se permite enviar cualquier campo pero que en cada llamada solo uno tenga información. Por ello, desde el frontend se le da un valor por defecto a todos los parámetros de un string vacío y aquel que se complete cambia su valor. Esos parámetros se reciben en una función que debe comprobar qué parámetro no es un string vacío y, en función de eso, realizar una u otra acción.

Si se proporciona el *ID* o el *locus*, la función lo busca previamente en el repositorio de GenBank en Virtuoso. Para ello, construimos un objeto de la clase *Virtuoso* que hemos creado en la API, que se trata de una clase con métodos que nos permite la conexión con el repositorio y otros métodos como consultas e inserciones. A ese objeto le pasamos los credenciales para el acceso, es decir, endpoint de consulta (http://localhost:8890/sparql-auth/), usuario, contraseña y base de datos (http://localhost:8890/GENBANK). El proceso para construir la consulta que se le pasa a este objeto *Virtuoso* y comprueba la existencia de ese id o locus es el que aparece en la figura 29.

```
consulta += "select\n"
consulta += "  ?id\n"
consulta += "  ?locus\n"
consulta += "  ?description\n"
consulta += "  ?organism\n"
consulta += "  ?taxonomy\n"
consulta += "  ?name\n"
consulta += "  *\n"
consulta += "where{{\n"

consulta += " ?gen rdf:type gen:Gene.\n"
consulta += " ?gen gen:ID ?id.\n"
if (id != ''):
    consulta += " filter(regex(lcase(?id), " + '"' + id + '"' + ")).\n"
consulta += "\n"

consulta += " ?gen gen:Locus ?locus .\n"
if (locus != ''):
    consulta += " filter(regex(lcase(?locus), " + '"' + locus + '"' + ")).\n"
consulta += "\n"

consulta += " ?gen gen:Description ?description.\n"
consulta += " ?gen gen:Organism ?organism.\n"
consulta += " ?gen gen:Taxonomy ?taxonomy.\n"
consulta += " ?gen gen:Name ?name.\n"
consulta += "\n"

consulta += "}}\n"
```

Figura 29: *Proceso de construcción de la consulta SPARQL en GenBank.*

Imaginemos, por ejemplo, que el usuario busca el gen con ID *50644*. La consulta que se construye es la de la figura 30.

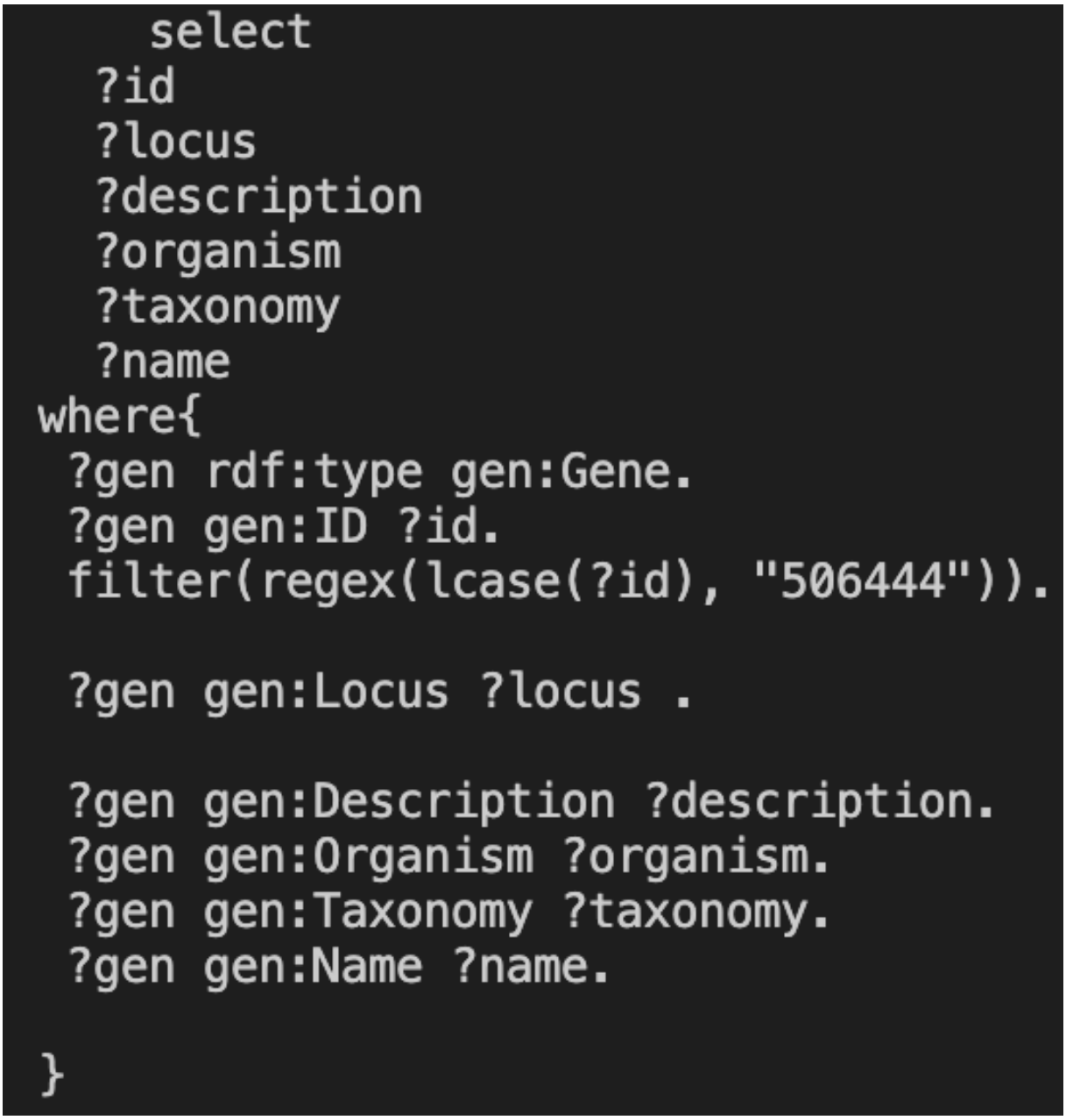


Figura 30: *Ejemplo de consulta SPARQL en GenBank.*

Si ese ID o locus ya se encuentran en el repositorio, la consulta traerá la información en formato JSON, mediante una función debe convertir ese JSON a una serie de diccionarios. Esos diccionarios se deben recorrer para quedarnos únicamente con el valor de cada campo, y una vez que ya tenemos eso, los vamos metiendo en una lista.

Entonces, comprobamos si esa lista producto de la consulta esta vacía. De ser así, es que

no está el ID o locus buscado en el repositorio y hay que optar por el otro tipo de búsqueda.

En ese caso se realiza un *e-fetch* con el valor proporcionado. Descargamos un archivo XML con la información del gen en cuestión, cuyo aspecto se puede ver en la figura 31. Podemos ver que tiene bastantes anidaciones, es decir, niveles dentro de otros niveles. Esto hace que el proceso para recorrerlo y sacar la información de interés sea muy complejo. Además, no todos los archivos se descargan con el mismo orden de anidaciones, por lo que hay que tener bastantes alternativas en cuenta.

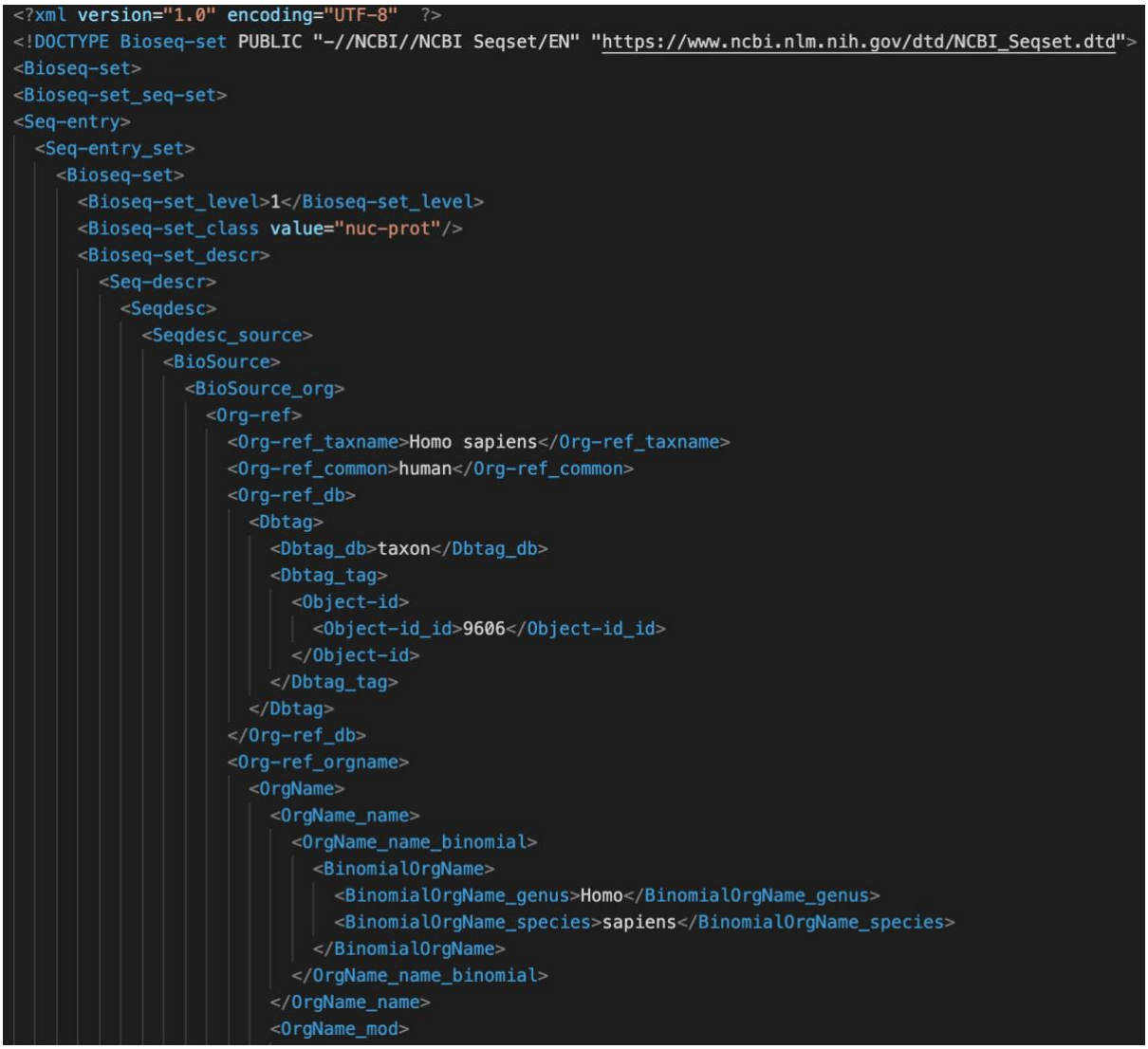


Figura 31: *Archivo XML que se descarga cuando se hace una búsqueda por ID o locus.*

Mediante una función, abrimos ese archivo XML y lo pasamos a un diccionario utilizando un módulo de Python llamado **cElementTree**. Una vez tenemos ese archivo en forma de diccionario, tenemos que ir recorriéndolo y almacenando los campos que queremos en un nuevo diccionario que será el gen o genes que se corresponden con el ID o locus proporcionado. Para saber qué campos hay que seleccionar de ese diccionario inmenso, primero hemos tenido que hacer un estudio de varios archivos XML de descarga, para conocer el patrón que tienen. Este proceso es algo lento y que hace el trabajo con GenBank muy pesado.

Ese diccionario ya creado con los campos que queremos se va añadiendo en una lista, que

será la que se pase como resultado al frontend a través de la API. Además de eso, también se le pasa a otra función, que va recorriendo la lista de diccionarios y formando tripletas para subirlas al repositorio de GenBank en Virtuoso. Para eso, tenemos que crear un objeto *Virtuoso* y pasarle los credenciales para establecer la conexión con la base de datos (endpoint de consulta, base de datos, usuario y contraseña). A este objeto le vamos pasando las tripletas a través de un *INSERT*. En la figura 32 tenemos ese código de *INSERT*, donde el elemento *store* es el objeto de la clase Virtuoso que hemos creado y *database* es la base de datos en la que queremos que se inserte la nueva información, en nuestro caso, http://localhost:8890/GENBANK.

```
"INSERT DATA { GRAPH <" + store.database + "> {" + triples + "} }"
```

Figura 32: *Código para el INSERT de la tripleta en la base de datos.*

En el caso de que se haya proporcionado *una descripción* o un *organismo*, primero se utiliza el *e-search* para obtener los ID de los genes que satisfagan esa condición, de la siguiente forma:

https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?db=nuccore&term= '+ **description** +'[title]&rettype=gb&retmode=xml

ó

https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?db=nuccore& term='+ **organism** +'[organism]&rettype=gb&retmode=xml

Descargamos un archivo XML con el resultado de esta búsqueda: IDs de los genes relacionados con la descripción u organismo proporcionado. El archivo que descargamos tiene el formato que aparece en la figura 33.

Figura 33: *Archivo XML que descargamos tras una búsqueda por e-search.*

Mediante una función, pasamos ese archivo a diccionario, lo recorremos y obtenemos una lista con los IDs.

Una vez tenemos los IDs, hacemos una búsqueda en Virtuoso por cada uno de los que haya en la lista de la manera que lo hacemos cuando solo se proporciona un ID o un locus. Si el ID se encuentra en el repositorio, traemos la información en JSON, mediante otra función la pasamos a diccionarios y estos diccionarios los recorremos para solo quedarnos con los valores de los campos, desechando todos lo demás. Los diccionarios nuevos se van introduciendo en una lista.

Si, por el contrario, no se ha encontrado, ese ID sigue en la lista que creamos al recorrer el archivo de descarga de los IDs. Comprobamos si quedan IDs en la lista, lo cual quiere decir que hay entradas no almacenadas en Virtuoso y, si es así, por cada ID de esa lista, hacemos un *e-fetch* y lo descargamos en formato XML. Este archivo XML tiene el mismo aspecto que el de la figura 31.

Por tanto, lo abrimos y lo pasamos a diccionario con el módulo **cElementTree**. Recorremos ese diccionario seleccionando los campos que nos interesen, pasando por distintos niveles de profundidad en el mismo, y vamos metiendo esos campos en un nuevo diccionario. Ese nuevo diccionario se añade a la lista de diccionarios que creamos cuando hacemos la búsqueda en Virtuoso. Si no se encontró ninguno en la búsqueda, esa lista estará vacía. Además, debemos

pasar los nuevos diccionarios al repositorio de GenBank en Virtuoso para próximas búsquedas. Para ello, usamos una función para convertir cada uno de esos diccionarios a tripletas, creamos el objeto *Virtuoso* con los credenciales y le pasamos las tripletas una por una.

Para acabar el proceso, la lista se envía como resultado a través de la API al frontend, donde el usuario podrá visualizarlo en forma de tabla, siendo cada fila de esa tabla un diccionario de la lista y cada columna una clave del diccionario (ID, Locus, Description, Taxonomy, Organism, Gene Name).

Por último, si el usuario proporciona un *nombre*, se realiza una primera búsqueda con *e-search* para encontrar los IDs correspondientes, como se ve a continuación:

https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?db=gene&term=' + **name** + '[Gene Name]&rettype=gb&retmode=xml

Utilizamos otro tipo de *e-search* porque, después de varias pruebas de descargas y estudio de las posibilidades de las *E-utilities*, nos dimos cuenta que la única forma de encontrar información fiable al pasar nombres de genes era utilizar como**db** *gene* y no *nuccore* como en los casos anteriores. *Gene* y *Nuccore* son dos bases de datos .[en]trez"de genes que proporciona el NCBI.

Descargamos el archivo XML que contiene los IDs de los genes almacenados con ese nombre, cuyo aspecto es igual que el de la figura 33.

Utilizamos la función que usamos también para la búsqueda por descripción y organismo para abrir ese archivo y pasarlo a un diccionario. Recorremos este diccionario y pasamos cada uno de los IDs a una lista.

Como ocurre con *descripción* y *organismo*, hacemos una búsqueda previa de cada uno de esos IDs de la lista en Virtuoso. Para ello, creamos el objeto *Virtuoso* que nos permitirá acceder al repositorio y les pasamos los credenciales de acceso (endpoint, base de datos concreta, usuario y contraseña). Construimos la consulta como en la figura 30 y se la pasamos al objeto *Virtuoso* para que la ejecute.

En caso de haber información ya almacenada, la traemos en formato JSON, la pasamos de JSON a una lista de diccionarios con una función y vamos recorriendo uno a uno esos diccionarios para quedarnos solo con los valores de los campos. Cada diccionario se mete en una nueva lista. Ese ID que ya ha sido buscado en el repositorio y encontrado se borra de la

lista de IDs que creamos tras el *e-search*.

Si después de la búsqueda y de habernos traído la información de aquellos que ya estaban en el repositorio quedan IDs en la lista, para cada ID de esa lista lanzamos un *e-fetch* y descargamos el XML con la información. El archivo XML de descarga tiene el formato de la figura 34. Si lo comparamos con la figura 31, podemos ver que el formato es distinto, tanto en nombre de los campos como en anidamiento. Es por ello por lo que tenemos que utilizar una función para pasar de XML a diccionario en el caso de ID, locus, organismo y descripción y otra distinta para nombre.

Figura 34: *Archivo XML que descargamos tras una búsqueda por ID a partir de una búsqueda por nombre.*

Abrimos este XML y lo pasamos a diccionario con el módulo **cElementTree** de Python. Recorremos el diccionario y nos quedamos con aquellos campos que nos interesan. También hemos tenido que comparar varios archivos XML con este formato para saber el patrón y

dónde se encuentran los datos de interés. Al igual que con el XML de la figura 31, en este caso no todos son exactamente iguales, por lo que también hemos tenido que tener en cuenta las diferencias a la hora de recorrerlo. Introducimos esos campos en un nuevo diccionario y ese diccionario a la lista ya creada tras la búsqueda en el repositorio. No olvidemos introducir la nueva información en el repositorio en Virtuoso para futuras consultas. Para ello, por cada diccionario de la lista, creamos una tripleta. Al igual que para la consulta en el repositorio, para la inserción también tenemos que crear un objeto de la clase *Virtuoso* que nos permita el acceso al repositorio. Le pasamos los credenciales para la conexión y le vamos insertando una por una las tripletas creadas.

Finalmente, la lista de diccionarios la pasamos al frontend a través de la API, donde se representará en forma de tabla con una fila por diccionario y una columna por campo de información.

Como hemos podido comprobar, en GenBank la búsqueda tiene muchas peculiaridades, muchos procesos alternativos y numerosos pasos hasta lograr el objetivo final. Esto ha hecho muy tedioso el trabajo.

7

Resultados

En esta sección mostraremos el resultado final de este Trabajo Fin de Grado, mostrando la aplicación final, la manera en la que el usuario puede interactuar con ella y cómo se muestran los resultados tras las búsquedas. La página principal de nuestra plataforma se puede observar en la figura 35.

Nos encontramos con una página sencilla, que tiene una imagen central en la cabecera del Linked Open Data Cloud y un título ”SEARCHER”. Debajo aparecen tres enlaces, uno para cada base de datos. Veamos qué búsquedas se pueden hacer en cada base de datos.

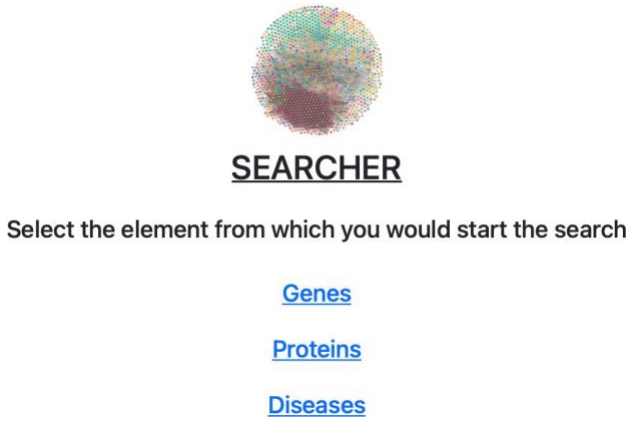


Figura 35: *Página de inicio de la plataforma.*

7.1. Genes

Si elegimos la búsqueda a partir de los genes, nos encontraremos con la página de la figura 36.

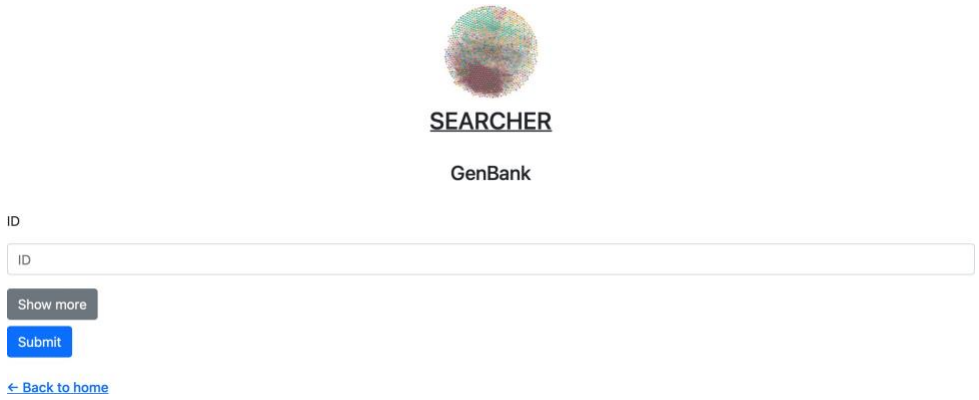


Figura 36: *Página principal de la sección de búsqueda de GenBank.*

Vuelven a aparecer la imagen central del Linked Open Data y el título ”SEARCHER”, que también van a aparecer en las secciones de búsqueda de enfermedades y proteínas. Si pulsamos sobre la imagen o el título, volveremos a la página inicial de la aplicación.

También vemos un subtítulo que informa de la base que vamos a consultar, *GenBank* en este caso.

Debajo nos encontramos con el principal campo de búsqueda, *ID*, un botón para mostrar más filtros, otro botón para mandar los parámetros una vez introducidos y un enlace de vuelta a la página inicial (← *Back to home*).

Hagamos una primera búsqueda proporcionando sólo el ID. Vamos a buscar qué gen se corresponde con el ID *506446* (Figura 37).

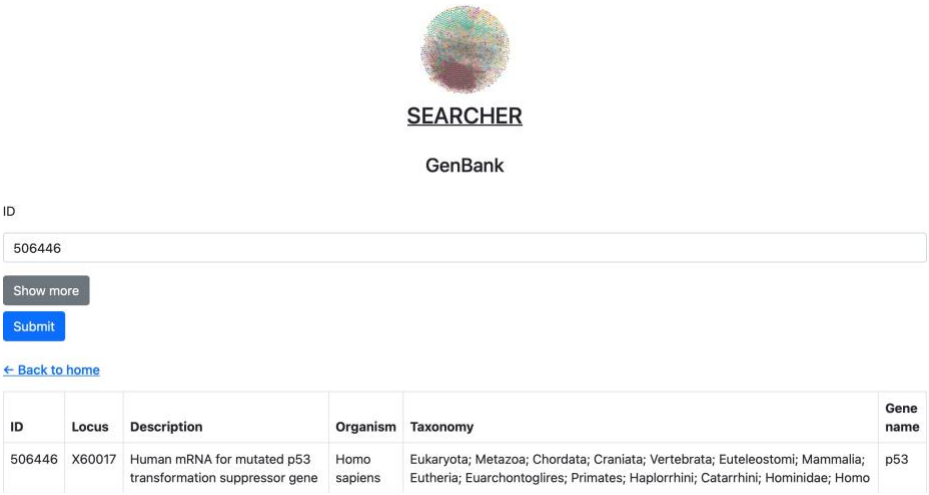


Figura 37: *Búsqueda del gen con ID 506446.*

Como ya hemos comentado anteriormente varias veces, el ID es único, por lo que solo va a poder corresponder a una entrada en GenBank. En la figura 37 podemos comprobar que el ID *506446* pertenece al ARN mensajero del gen *p53* mutado, que es el gen supresor de la transformación de la proteína con el mismo nombre, *p53*.

Si pulsamos sobre el botón *Show more*, podemos ver el resto de filtros que se nos ofrece en la búsqueda (Figura 38). Aparece un campo para el locus del gen, otro para el nombre, otro para el organismo sobre el que queremos buscar genes y, por último, uno para añadir la descripción de un gen. Como se indica en negrita, solo puede rellenarse un campo para que la búsqueda sea correcta (**Fill in one field only**). También se indica que en el caso de buscar un *organism* o *description*, hay que añadir un símbolo + entre palabras.

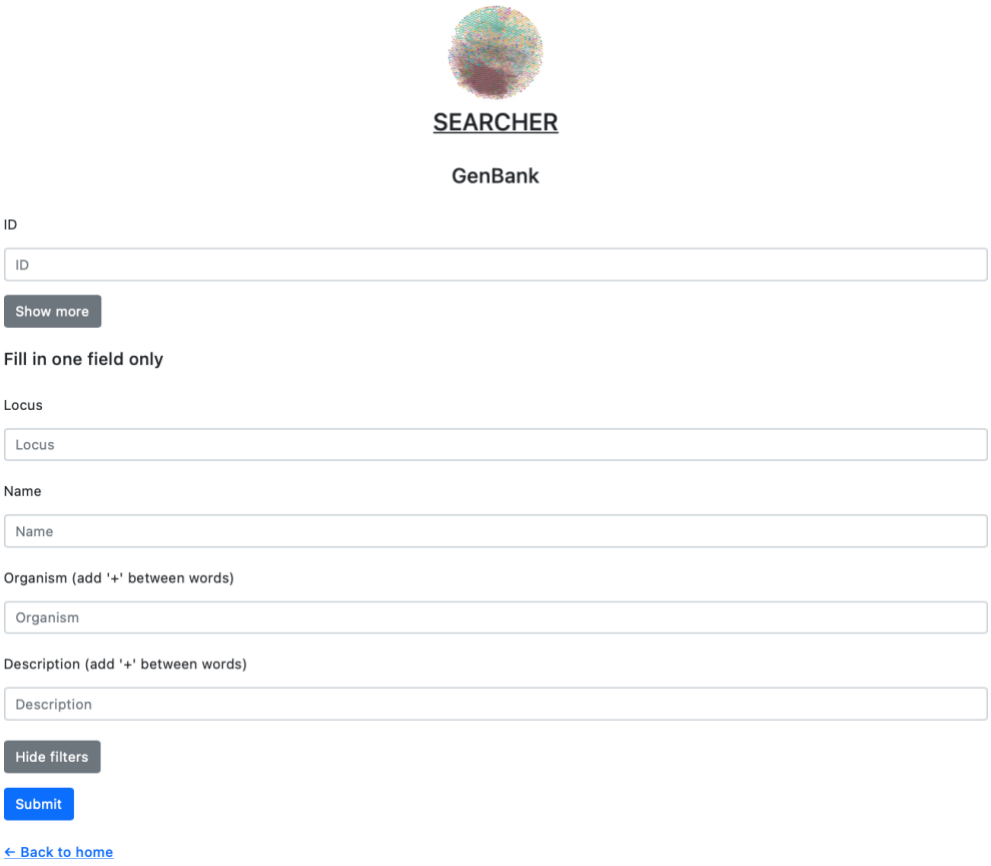


Figura 38: *Filtros de búsqueda en GenBank.*

Realizamos una búsqueda para el gen con el nombre *p53*, que es el que hemos obtenido en la anterior búsqueda por ID.

Como podemos observar en la figura 39, obtenemos varias filas de genes que se corresponden con el nombre *p53*. Si nos fijamos en los organismos al que pertenecen, todos son distintos, porque un gen puede estar en varias especies. De ahí que nos aparezca más de un resultado.

ID	Locus	Description	Organism	Taxonomy	Gene name
7157	No locus given	tumor protein p53	Homo sapiens	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Mammalia; Eutheria; Euarchontoglires; Primates; Haplorrhini; Catarrhini; Hominidae; Homo	TP53
22059	NC_000077	transformation related protein 53	Mus musculus	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Mammalia; Eutheria; Euarchontoglires; Glires; Rodentia; Myomorpha; Muroidea; Muridae; Murinae; Mus; Mus	Trp53
24842	NC_051345	tumor protein p53	Rattus norvegicus	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Mammalia; Eutheria; Euarchontoglires; Glires; Rodentia; Myomorpha; Muroidea; Muridae; Murinae; Rattus	Tp53
30590	No locus given	tumor protein p53	Danio rerio	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Actinopterygii; Neopterygii; Teleostei; Ostariophysi; Cypriniformes; Danionidae; Danioninae; Danio	tp53
2768677	NT_033777	No description given	Drosophila melanogaster	Eukaryota; Metazoa; Ecdysozoa; Arthropoda; Hexapoda; Insecta; Pterygota; Neoptera; Holometabola; Diptera; Brachycera; Muscomorpha; Ephydroidea; Drosophilidae; Drosophila; Sophophora	p53
403869	No locus given	tumor protein p53	Canis lupus familiaris	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Mammalia; Eutheria; Laurasiatheria; Carnivora; Caniformia; Canidae; Canis	TP53
397276	NC_010454	tumor protein p53	Sus scrofa	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Mammalia; Eutheria; Laurasiatheria; Artiodactyla; Suina; Suidae; Sus	TP53
397926	NC_054375	tumor protein p53 L homeolog	Xenopus laevis	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Amphibia; Batrachia; Anura; Pipoidea; Pipidae; Xenopodinae; Xenopus; Xenopus	tp53.L
100049321	NC_019876	tumor protein p53	Oryzias latipes	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Actinopterygii; Neopterygii; Teleostei; Neoteleostei; Acanthomorphata; Ovalentaria; Atherinomorphae; Beloniformes; Adrianichthyidae; Oryziinae; Oryzias	tp53
100062044	NC_009154	tumor protein p53	Equus caballus	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Mammalia; Eutheria; Laurasiatheria; Perissodactyla; Equidae; Equus	TP53

Figura 39: *Resultado de la búsqueda del gen ”p53”.*

7.2. Enfermedades

Si en lugar de elegir la búsqueda por genes, queremos buscar enfermedades, la página que se nos abrirá es exactamente igual que la de *GenBank*, con la única diferencia que el título que aparece es *OMIM* (Figura 40).

Vamos a buscar la enfermedad que se corresponde con el ID *275200*. En la figura 41, podemos observar que el ID se corresponde con la enfermedad de *NON GOITROUS CONGENITAL HYPOTHYROIDISM*, es decir, hipotiroidismo no congénito en el que tampoco se ha dado un agrandamiento de la glándula tiroides, de ahí lo de *NON GOITROUS*.

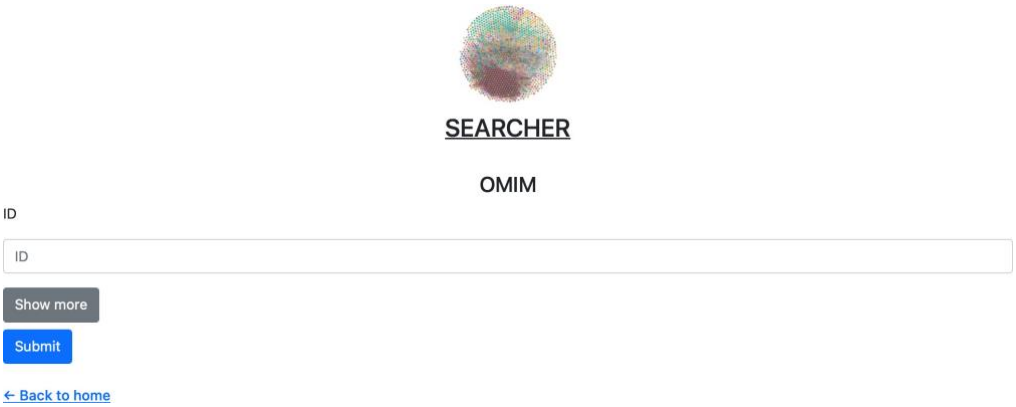


Figura 40: *Página principal de la sección de búsqueda de OMIM.*

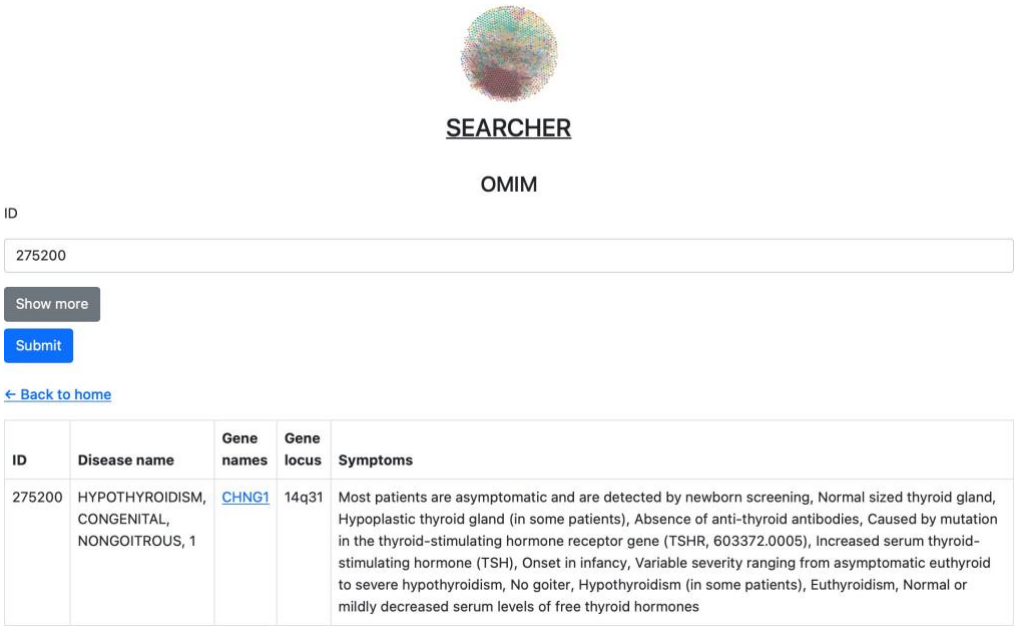
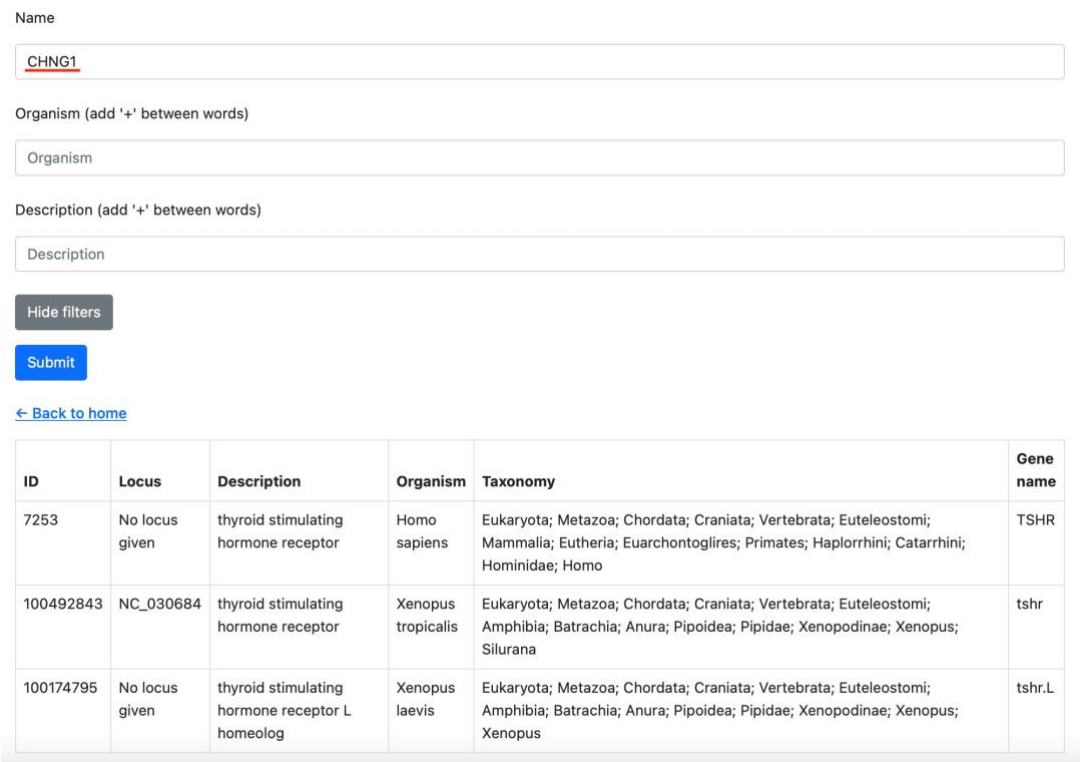


Figura 41: *Búsqueda de la enfermedad con ID 275200.*

También podemos observar en la imagen 41 que el gen asociado a la enfermedad viene en forma de enlace. Esto es así porque si pulsamos sobre él, nos lleva directamente al resultado de la búsqueda de ese gen en GenBank (Figura 42). Según el resultado, se trata del gen receptor de la hormona estimulante del tiroides, que se encuentra en *Homo Sapiens, Xenopus tropicalis* y *Xenopus laevis*.



ID	Locus	Description	Organism	Taxonomy	Gene name
7253	No locus given	thyroid stimulating hormone receptor	Homo sapiens	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Mammalia; Eutheria; Euarchontoglires; Primates; Haplorrhini; Catarrhini; Hominidae; Homo	TSHR
100492843	NC_030684	thyroid stimulating hormone receptor	Xenopus tropicalis	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Amphibia; Batrachia; Anura; Pipoidea; Pipidae; Xenopodinae; Xenopus; Silurana	tshr
100174795	No locus given	thyroid stimulating hormone receptor L homeolog	Xenopus laevis	Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi; Amphibia; Batrachia; Anura; Pipoidea; Pipidae; Xenopodinae; Xenopus; Xenopus	tshr.L

Figura 42: *Resultado de la búsqueda del gen CHNG1 desde una enfermedad en OMIM.*

Veamos ahora qué otros filtros nos ofrece la búsqueda en *OMIM*. Según la figura 43, podemos buscar también por nombre de la enfermedad, nombre y locus de algún gen que pueda estar relacionado con una enfermedad y síntomas.

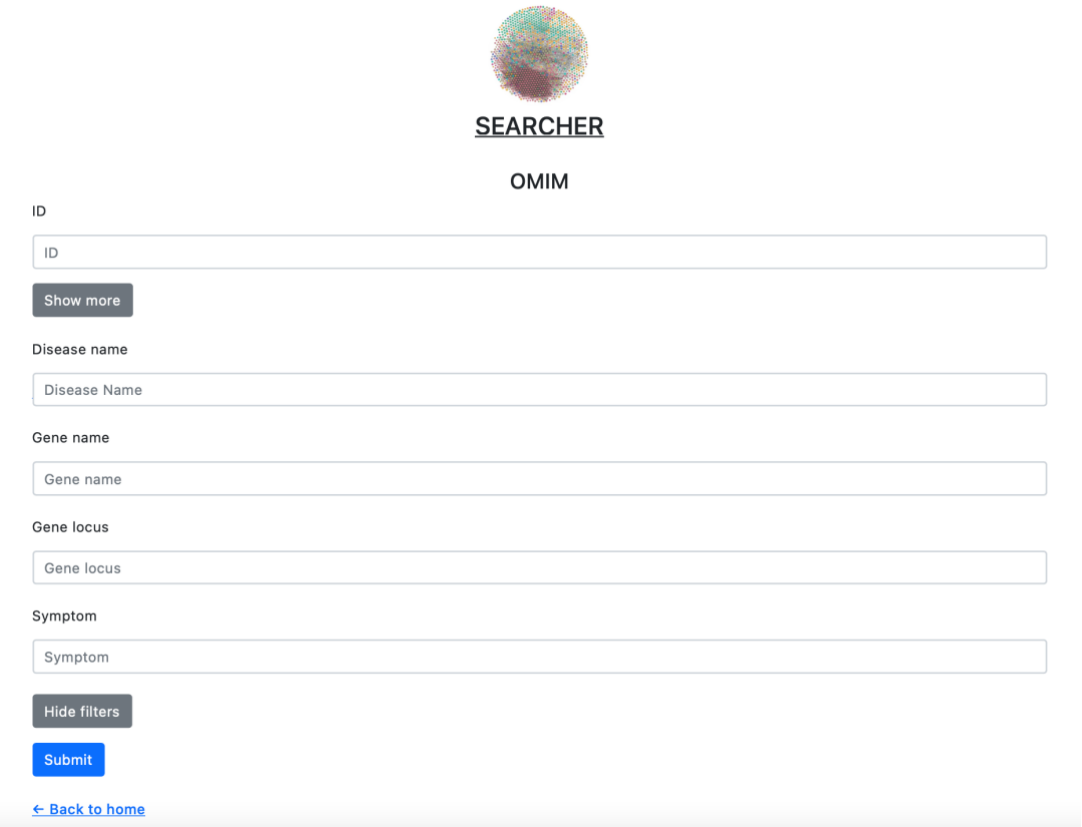


Figura 43: *Filtros de búsqueda en OMIM.*

Vamos a realizar una búsqueda por nombre para la enfermedad de *Alzheimer*. Como podemos ver en la figura 44, obtenemos 5 resultados, uno por cada tipo de *Alzheimer* que hay almacenado en la ontología. Cada uno tiene unos síntomas distintos y un locus y gen asociados diferentes. Al igual que en la búsqueda anterior, también podemos acceder a los genes asociados y obtener la información directamente.

ID	Disease name	Gene names	Gene locus	Symptoms
104300	ALZHEIMER DISEASE	HFE	6p21.3	Presenile and senile dementia, Genetic heterogeneity, Caused by mutation in the amyloid beta (A4) precursor protein gene (APP, 104760.0002), Parkinsonism, Neurofibrillary tangles composed of disordered microtubules, Susceptibility conferred by mutation in the alpha-2-macroglobulin gene (A2M, 103950.0005), Long tract signs
607822	ALZHEIMER DISEASE 3	AD2	19q13.2	Seizures, Dysarthria, Extensor plantar responses, Personality changes, Severe phenotype, Gait disturbances, Extrapyramidal signs, Eosinophilic 'cotton wool' plaques without dense congophilic core in various brain regions, Cortical atrophy, Loss of attention, Loss of executive functions, Behavioral changes, Social withdrawal, Onset in late twenties to thirties, A subset of patients have a 'visual variant', Rapidly progressive, Myoclonus, Dystonia, Apraxia, Dysphagia, Memory loss, Neurofibrillary tangles, Spastic quadriparesis, Alzheimer disease, early-onset, Dementia, progressive, Loss of language ability, Hyperreflexia in lower limbs, Constructional apraxia (in a subset of patients), Visuospatial agnosia (in a subset of patients), Optic ataxia (in a subset of patients), Cortical and subcortical regions involved, Amyloid plaques, Later onset has been reported, Caused by mutation in the presenilin-1 gene (PSEN1, 104311.0001)
608907	ALZHEIMER DISEASE 9, SUSCEPTIBILITY TO	ABCA7	19p13.3	Depression, Cortical atrophy, Mean age at onset 73 years (range 54 to 90), Disinhibition, Alzheimer disease, Memory loss, progressive, Language difficulties, Extrapyramidal signs (in some patients), Dyspraxia (in some patients), Hippocampal atrophy, Neuropathology shows neurofibrillary tangles, Senile plaques, Frontal signs, Susceptibility conferred by mutation in the ATP-binding cassette, subfamily A, member 7 gene (ABCA7, {605414.0001)}
606889	ALZHEIMER DISEASE 4	AD4	1q31-q42	Early onset, between 35-60 years, Accounts for <2% of patients with Alzheimer's disease, Neurofibrillary tangles and neuritic senile plaques rare, See entry 104300 for general information on Alzheimer disease, Presenile dementia, Severe amyloid angiopathy, Caused by mutation in the presenilin 2 gene (PSEN2, 600759.0001), Sleep-wake cycle disturbance
609636	ALZHEIMER DISEASE 10	AD10	7q36	Speech impairment, Dementia, Neuroimaging shows cortical atrophy, Memory impairment, Praxis, Mean age at onset 66.8 years (range 47-77), Personality changes

Figura 44: *Resultado de la búsqueda de la enfermedad "Alzheimer".*

7.3. Proteínas

En último lugar, si elegimos la búsqueda de proteínas, obtenemos la página de la figura 45. Es una página idéntica a la de búsqueda de genes y enfermedades, solo cambia el título, ya que en este caso es *Uniprot*.

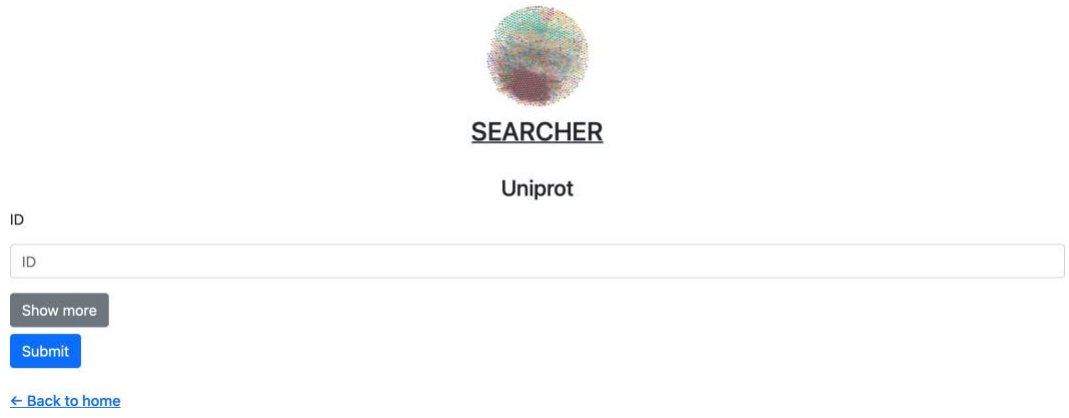


Figura 45: *Página principal de la sección de búsqueda de Uniprot.*

Realicemos la búsqueda de la proteína con ID *P09208*. En la figura 46 tenemos el resultado

de la búsqueda, y podemos ver que se trata de un receptor similar a la insulina que se encuentra en el organismo *Drosophila melanogaster*.

También observamos que el gen asociado se encuentra, como en el caso de la búsqueda por enfermedades, con enlace. Por tanto, podemos realizar la búsqueda de ese gen directamente al pinchar sobre él. En la figura 47, se visualizan los resultados de esta búsqueda, con varias filas que se corresponden con el gen en distintos organismos.

Protein	Gene name	Organism Name	Location Annotation	Function Annotation	Similarity Annotation	Disease Annotation	Protein Name
http://purl.uniprot.org/uniprot/P09208	InR	Drosophila melanogaster	A membrane is a lipid bilayer which surrounds enclosed spaces and compartments. This selectively permeable structure is essential for effective separation of a cell or organelle from its surroundings. Membranes are composed of various types of molecules such as phospholipids, integral membrane proteins, peripheral proteins, glycoproteins, glycolipids, etc. The relative amounts of these components as well as the types of lipids are non-randomly distributed from membrane to membrane as well as between the two leaflets of a membrane.	Has a ligand-stimulated tyrosine-protein kinase activity. Required for cell survival. Regulates body size and organ size by altering cell number and cell size in a cell-autonomous manner. Involved in the development of the embryonic nervous system, and is necessary for axon guidance and targeting in the visual system. Also plays a role in life-span determination.	Belongs to the protein kinase superfamily. Tyr protein kinase family. Insulin receptor subfamily.		Insulin-like receptor

Figura 46: *Búsqueda de la proteína con ID P09208.*

ID	Locus	Description	Organism	Taxonomy	Gene name
42549	NT_033777	Insulin-like receptor	Drosophila melanogaster	Eukaryota; Metazoa; Ecdysozoa; Arthropoda; Hexapoda; Insecta; Pterygota; Neoptera; Holometabola; Diptera; Brachycera; Muscomorpha; Ephydroidea; Drosophilidae; Drosophila; Sophophora	InR
1280455	No locus given	No description given	Anopheles gambiae str. PEST	Eukaryota; Metazoa; Ecdysozoa; Arthropoda; Hexapoda; Insecta; Pterygota; Neoptera; Holometabola; Diptera; Nematocera; Culicoidea; Culicidae; Anophelinae; Anopheles	INR
692560	NC_051362	insulin receptor	Bombyx mori	Eukaryota; Metazoa; Ecdysozoa; Arthropoda; Hexapoda; Insecta; Pterygota; Neoptera; Endopterygota; Lepidoptera; Glossata; Ditrysia; Bombycoidea; Bombycidae; Bombycinae; Bombyx	InR
5574210	NC_035109	insulin-like receptor	Aedes aegypti	Eukaryota; Metazoa; Ecdysozoa; Arthropoda; Hexapoda; Insecta; Pterygota; Neoptera; Endopterygota; Diptera; Nematocera; Culicoidea; Culicidae; Culicinae; Aedini; Aedes; Stegomyia	LOC5574210
411297	NC_037646	insulin-like peptide receptor	Apis mellifera	Eukaryota; Metazoa; Ecdysozoa; Arthropoda; Hexapoda; Insecta; Pterygota; Neoptera; Endopterygota; Hymenoptera; Apocrita; Aculeata; Apoidea; Apidae; Apis	LOC411297
118508044	NC_050202	sex peptide receptor-like	Anopheles stephensi	Eukaryota; Metazoa; Ecdysozoa; Arthropoda; Hexapoda; Insecta; Pterygota; Neoptera; Endopterygota; Diptera; Nematocera; Culicoidea; Culicidae; Anophelinae; Anopheles	LOC118508044
111509794	NW_019291010	insulin-like receptor	Leptinotarsa decemlineata	Eukaryota; Metazoa; Ecdysozoa; Arthropoda; Hexapoda; Insecta; Pterygota; Neoptera; Endopterygota; Coleoptera; Polyphaga; Cucujiformia; Chrysomeloidea; Chrysomelidae; Chrysomelinae; Doryphorini; Leptinotarsa	LOC111509794
661524	NC_007416	insulin-like receptor	Tribolium castaneum	Eukaryota; Metazoa; Ecdysozoa; Arthropoda; Hexapoda; Insecta; Pterygota; Neoptera; Endopterygota; Coleoptera; Polyphaga; Cucujiformia; Tenebrionidae; Tenebrionidae incertae sedis; Tribolium	LOC661524

Figura 47: *Resultado de la búsqueda del gen InR desde una proteína en Uniprot.*

Como en los dos casos anteriores, también tenemos la posibilidad de buscar mediante otros filtros. Según la figura 48, podemos buscar por el nombre de la proteína, el nombre del gen asociado, el locus del gen, el organismo y alguna anotación de enfermedad, localización, función, similitudes y enfermedad.

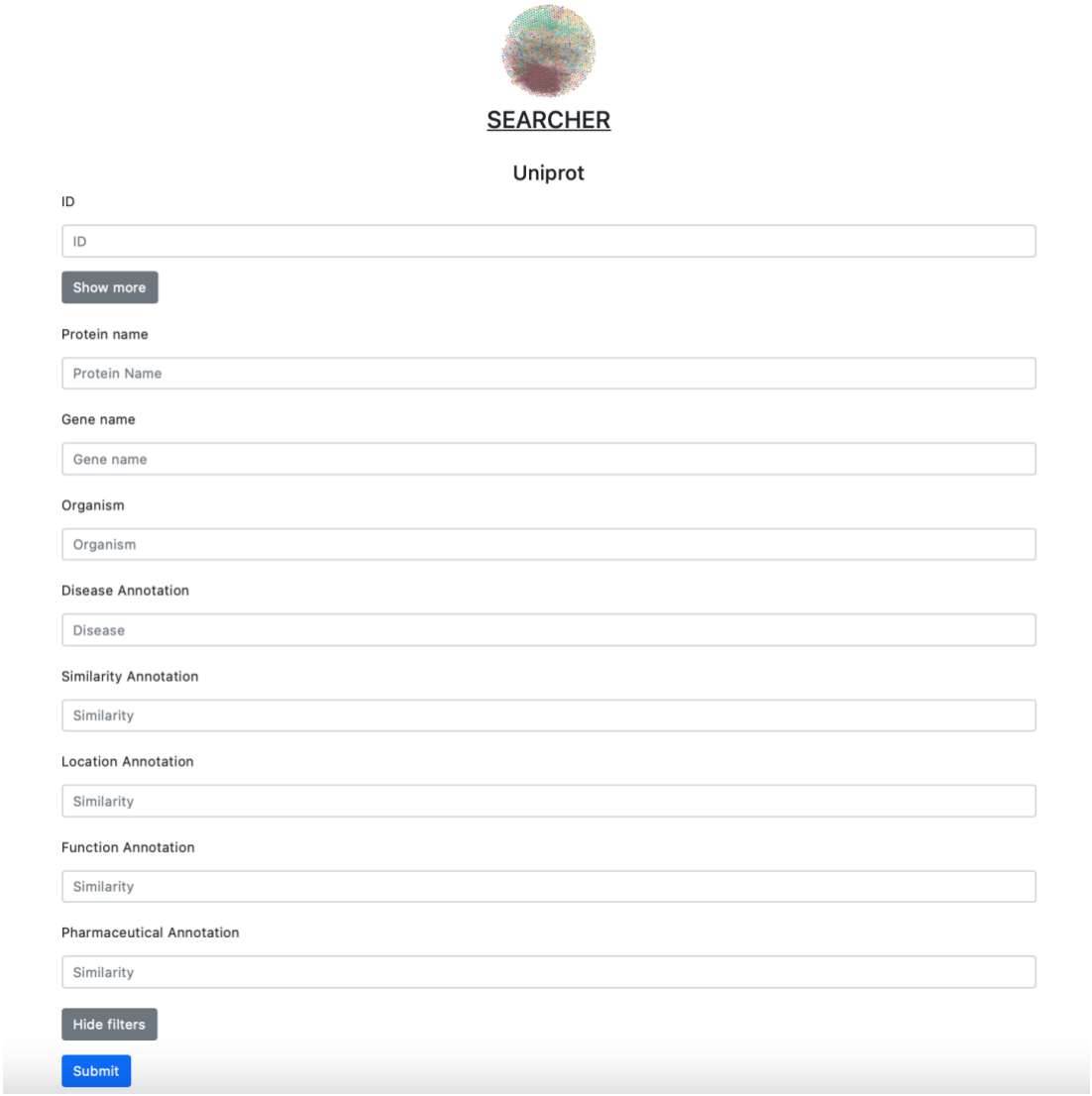


Figura 48: *Filtros de búsqueda en Uniprot.*

Vamos a buscar proteínas que estén asociadas con la *insulina* (insulin). Introducimos este concepto en el campo *Disease Annotation*.

Parte del resultado, porque no cabe todo en la imagen, se visualiza en la figura 49. Aparecen tres filas que corresponden a tres entradas de Uniprot asociadas a la insulina. Las representadas en esta imagen se refieren a la misma proteína, *insulin*, pero con distintas anotaciones de enfermedad.

Protein	Gene name	Organism Name	Location Annotation	Function Annotation	Similarity Annotation	Disease Annotation	Protein Name
http://purl.uniprot.org/uniprot/P01308	INS	Homo sapiens	Protein located outside the cell membrane(s).	Insulin decreases blood glucose concentration. It increases cell permeability to monosaccharides, amino acids and fatty acids. It accelerates glycolysis, the pentose phosphate cycle, and glycogen synthesis in liver.	Belongs to the insulin family.	A form of diabetes that is characterized by an autosomal dominant mode of inheritance, onset in childhood or early adulthood (usually before 25 years of age), a primary defect in insulin secretion and frequent insulin-independence at the beginning of the disease.	Insulin
http://purl.uniprot.org/uniprot/P01308	INS	Homo sapiens	Protein located outside the cell membrane(s).	Insulin decreases blood glucose concentration. It increases cell permeability to monosaccharides, amino acids and fatty acids. It accelerates glycolysis, the pentose phosphate cycle, and glycogen synthesis in liver.	Belongs to the insulin family.	A multifactorial disorder of glucose homeostasis that is characterized by susceptibility to ketoacidosis in the absence of insulin therapy. Clinical features are polydipsia, polyphagia and polyuria which result from hyperglycemia-induced osmotic diuresis and secondary thirst. These derangements result in long-term complications that affect the eyes, kidneys, nerves, and blood vessels.	Insulin
http://purl.uniprot.org/uniprot/P01308	INS	Homo sapiens	Protein located outside the cell membrane(s).	Insulin decreases blood glucose concentration. It increases cell permeability to monosaccharides, amino acids and fatty acids. It accelerates glycolysis, the pentose phosphate cycle, and glycogen synthesis in liver.	Belongs to the insulin family.	An autosomal dominant condition characterized by elevated levels of serum proinsulin-like material.	Insulin

Figura 49: *Resultado de la búsqueda de la proteína relacionada con la insulina (insulin).*

También tenemos la posibilidad de pulsar sobre cualquier enlace de los genes y realizar la búsqueda directa de ese gen en cuestión.

8

Conclusiones y Líneas Futuras

8.1. Conclusiones

En este Trabajo Fin de Grado se ha investigado a fondo la condición en la que se encuentran tres de los repositorios más utilizados en investigación en Ciencias de la Vida: **GenBank**, **OMIM** y **Uniprot**. Para cada una de estas bases de datos, se ha estudiado toda la información que hay almacenada en Linked Data, todos los datos que ofrecen en RDF y todas las alternativas disponibles para su consulta.

En el caso de Uniprot, el trabajo de búsqueda de un endpoint actualizado y disponible para la consulta ha sido el más fácil, ya que como comentamos en el capítulo 3 ya dispone de un SPARQL Endpoint actualizado. Lo complicado en esta base de datos ha sido el requerir consultas de un tiempo de ejecución muy alto, debido a las numerosas entradas almacenadas de las que dispone y que estas tienen que resolverse en un servidor remoto del que no se dispone de control. Queríamos que hubiese la mayor cantidad posible de opciones de búsqueda y, para ello, las consultas debían ser muy detalladas para poder abarcarlo todo. De ahí que hayamos tenido que optar por reducir el límite de búsquedas.

En cuanto a OMIM, el trabajo inicial de búsqueda de información disponible para la consulta fue mucho más difícil que la que acabamos de relatar. Este repositorio no tiene Endpoint disponible ni otro mecanismo eficaz y actualizado de acceder a los datos. Lo único que pudimos encontrar fue la ontología con la que hemos trabajado, una ontología enorme y difícil de tratar. El proceso de estudio de esa ontología para saber la manera en la que consultarla ha sido el más complicado de este proceso, ya que tampoco disponíamos de un esquema de RDF claro con el que saber qué organización presenta.

Si hablamos de GenBank ha sido el proceso más complejo de todos. No hay ni SPARQL Enpoint disponible ni tampoco ontología. De ahí que optásemos por el mecanismo de las **E-utilities**. Aún así, tampoco es una manera fácil de obtener la información. Primero hemos tenido que empaparnos de cómo funciona esta herramienta, de todos los tipos de descarga de datos que se pueden hacer, del formato que podemos obtener y de la manera de trabajar con esos archivos. Cabe mencionar que también queríamos permitir que el usuario consultase a través de varios campos, lo que ha hecho la investigación y el desarrollo del código aún más difícil. Además, como hemos visto en el capítulo 6 los archivos que obtenemos con la información no siempre tienen los mismos campos, por lo que el número de alternativas es muy grande.

Con todo este estudio previo, hemos desarrollado un demostrador tecnológico de la utilidad que tienen las tecnologías semánticas del Linked Data en Ciencias de la Vida. Para ello, hemos usado como interfaz **Next.js**, un framework que trabaja con JavaScript y que es totalmente nuevo para mí. Además, para la conexión con las bases hemos usado una API en Python, **FastAPI**, con la que tampoco había trabajado anteriormente a esta profundidad.

El aprendizaje de estas dos nuevas herramientas ha sido un reto para mí en este proyecto, al que le sumo la tediosa tarea de investigar el estado de los repositorios.

Este proyecto también me ha hecho reflexionar sobre la importancia de los estándares de datos y de tener una información bien organizada, algo que es crucial para trabajar con esos datos en funciones reproducibles y aún más importante para su representación.

También he podido aprender muchísimos conceptos nuevos y muy relevantes en el desarrollo de aplicaciones. Uno de ellos es el uso de **microservicios**. Al tener una estructura de microservicios, la aplicación o software que estés desarrollando será más fácil de escalar y mucho más rápida de desarrollar, además de que tendrá un código mucho más claro y organizado. Otra cosa importante es el uso de **cachés**, lo que te permite almacenar las consultas que se ejecuten con mayor frecuencia y que si se vuelve a buscar el mismo concepto la respuesta sea más rápida. Todo esto le da eficiencia y fluidez a la aplicación.

Agradezco enormemente todo el conocimiento que me llevo de este Trabajo Fin de Grado porque sé que todo lo aprendido es muy útil y podré aplicarlo en un futuro.

8.2. Líneas Futuras

En cuanto a trabajos futuros que nos gustaría realizar sobre este proyecto, el primero de ellos es extender este demostrador a otras bases de datos relacionadas con las Ciencias de la Vida. Hacer un análisis sistemático del estado de los Linked Data en Ciencias de la Vida e integrar los repositorios de mayor utilidad e interés a nuestra plataforma.

También nos gustaría tener todas estas bases de datos conectadas entre sí. Es decir, que desde todas las secciones de búsqueda haya, al menos, un enlace que reconduzca a una búsqueda en otra de las secciones de la aplicación.

Por último, pretendemos mejorar el funcionamiento ya existente de nuestra aplicación: añadir nuevos campos de búsqueda, ampliar la información que se devuelve tras las consultas, mejorar el aspecto de la aplicación, etc.

Referencias

[1] *European Molecular Biology Laboratory | EMBL*. URl: https://www.embl.org.

[2] *GenBank*. URl: https://www.ncbi.nlm.nih.gov/genbank/.

[3] *OMIM*. URl: https://omim.org.

[4] *Uniprot*. URl: https://www.uniprot.org.

[5] *Worldwide Protein Data Bank*. URl: https://www.wwpdb.org.

[6] *Linked Open Data Cloud*. URl: https://lod-cloud.net.

[7] *Linked Data*. URl: https://es.wikipedia.org/wiki/Datos_enlazados.

[8] *Sparql Language*. URl: https://www.w3.org/TR/rdf-sparql-query/.

[9] *OpenLink Virtuoso Universal Server Documentation*. URl: http://docs.openlinksw.com/virtuoso/.

[10] *StarDog*. URl: https://www.stardog.com.

[11] Kiyoko F. Aoki-Kinoshita y col. "Implementation of linked data in the life sciences at BioHackathon 2011". En: *Journal of Biomedical Semantics* 6.1 (2015), págs. 1-13. ISSN: 20411480. DOI: 10.1186/2041-1480-6-3.

[12] Tim Berners-Lee, James Hendler y Ora Lassila. "The Semantic Web A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities". En: *Scientific American* 284.5 (2001), págs. 1-5.

[13] Juan Manuel' 'González Mañas. *Manual access to GenBank*. URl: http://www.ehu.eus/biofisica/juanma/bioinf/pdf/genbank.pdf.

[14] *Bio2RDF*. URl: https://bio2rdf.org.

[15] *Entrez Programming Utilities Help*. URl: https://www.ncbi.nlm.nih.gov/books/NBK25501/.

[16] *BioPortal*. URl: https://bioportal.bioontology.org.

[17] *BioPortal - OMIM*. URl: https://bioportal.bioontology.org/ontologies/OMIM.

[18] *Sparql Uniprot*. URl: https://sparql.uniprot.org/sparql.

[19] *Next.js*. URl: https://nextjs.org.

[20] *React.js*. URl: https://es.reactjs.org.

[21] Sergio Daniel Xalambrí. *Qué es Next.js, el framework JS de React para construir el futuro*. URl: https://platzi.com/blog/nextjs-el-futuro-de-las-aplicaciones-con-react/.

[22] *Apache*. URl: http://httpd.apache.org.

[23] *FastAPI*. URl: https://fastapi.tiangolo.com.

[24] *FastApi, el framework veloz hecho con Python*. URl: http://www.buscaminegocio.com/cursos-de-python/fastapi-framework-en-python.html.

[25] *Balanceador de cargas*. URl: https://servsoft.com.co/infraestructura-it/balanceador-de-carga/.

[26] ¿Qué es una arquitectura de microservicios? URl: https://decidesoluciones.es/arquitectura-de-microservicios/.

[27] *Kubernetes*. URl: https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/.

[28] *Redis*. URl: https://redis.io.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA