Trabajo de Fin de Grado

Detección automática de expresiones y locuciones en textos

TITULACIÓN: Grado en Ingeniería Informática

AUTOR: Alejandro Matías Cruz de la Cruz

TUTORIZADO POR:
Francisco Javier Carreras Riudavets

Fecha 06/2024

Agradecimientos

Agradezco a mi familia, en especial a mi madre, por brindar guía y apoyo incondicionales a lo largo de toda mi vida. Este trabajo está dedicado a ellos.

Agradezco a Francisco Carreras, tutor y autor de la idea detrás de este trabajo, por su tiempo y disposición a resolver dudas y realizar aclaraciones.

Le agradezco también a usted, lector, que haya decidido dedicar tiempo a la lectura de este trabajo. Espero que le sea provechosa.

Resumen

Las expresiones y locuciones son secuencias de palabras cuyo significado no se deriva de las palabras que las forman. Son construcciones lingüísticas de gran importancia, pues se utilizan de manera frecuente, especialmente en el registro coloquial. A pesar de ello, no existe, en estos momentos, ninguna herramienta de acceso público que permita la detección automática de las mismas, al menos no con un margen de error razonable. El objetivo principal de este Trabajo de Fin de Título será, precisamente, suplir esta carencia, desarrollando una aplicación web que permita a sus usuarios obtener un listado de las expresiones y locuciones de la lengua española contenidas en cualquier texto de su elección, junto a sus respectivos significados y ejemplos de uso.

Abstract

Expressions and idiomatic phrases are sequences of words whose meaning is not derived from the individual words that comprise them. They are linguistic constructions of great importance, since they are used very frequently, especially in informal speech. Nonetheless, at the moment, there is no publicly available tool that allows for the automatic detection of these expressions, at least not with a reasonable margin of error. The main objective of this End of Degree Project will be precisely to address this gap, by developing a web application that enables users to obtain a list of the expressions and idiomatic phrases in the Spanish language contained within any text of their choice, along with their respective meanings and usage examples.

Índice general

Índice de figuras

Índice de fragmentos de código

Capítulo 1

Introducción

Las expresiones idiomáticas y locuciones son secuencias de palabras cuyo significado no es composicional [2], es decir, el significado de las mismas no se deriva de las palabras que las forman [1]. Son, por tanto, construcciones lingüísticas considerablemente desafiantes, tanto para aquellas personas que no están familiarizadas con ellas, como para los sistemas de traducción automática, que a menudo se muestran imprecisos al analizar el significado de expresiones y locuciones, resultando en pérdida del sentido de las oraciones. No obstante, a pesar de su inherente dificultad, son uno de los aspectos más esenciales de la lengua, siendo utilizadas constantemente, en especial en las interacciones coloquiales y cotidianas.

Por estos motivos, se pretende desarrollar un software capaz de detectar, automáticamente, todas las expresiones y locuciones de la lengua española localizadas en cualquier texto. Dicho sistema debería ser capaz de obtener y proporcionar al usuario la localización en el texto, categorías, significado, o significados, si tuviera varias acepciones, y ejemplos de uso de cada una de ellas.

Con este propósito, se crearán tres aplicaciones diferentes. En primer lugar, una aplicación de escritorio para el sistema operativo Windows. Esta permitirá a un usuario administrador introducir en una base de datos toda la información relativa a las expresiones y locuciones, a partir de diccionarios almacenados en ficheros de texto. En concreto, el diccionario a utilizar será el Diccionario de la lengua española (DLE), editado y publicado por la Real Academia Española [3]. Dicha obra contiene definiciones, categorías y, en ocasiones, ejemplos de uso de todas las expresiones y locuciones de la lengua española. A pesar de que, durante el desarrollo de este trabajo, se utilizará únicamente este diccionario para poblar la base de datos, la aplicación se diseñará para que sea escalable, ofreciendo la posibilidad de incluir más fuentes de información.

Por otra parte, se desarrollará un servicio web que detectará las expresiones y locuciones de la lengua española, a partir de un texto cualquiera, proporcionando toda la información pertinente. Este servicio web podrá ser utilizado, a su vez, por otras aplicaciones, permitiendo integrar el sistema de detección automática de expresiones y locuciones en cualquier otro software ya existente. Esto es particularmente importante, puesto que el IATEXT (Instituto Universitario de Análisis y Aplicaciones Textuales), del que forma parte Francisco Carreras,

tutor y autor de la idea de este Trabajo de Fin de Título, ya cuenta con diferentes aplicaciones donde este sistema podría resultar de utilidad.

Además, se creará una aplicación web, que permitirá interactuar con el sistema a través de la interfaz gráfica del navegador. Por medio de esta, el usuario podrá introducir cualquier texto de su elección, o subir un fichero de texto plano, tras lo cual recibirá toda la información relativa a las expresiones y locuciones localizadas en el mismo.

No obstante, antes de abordar el desarrollo del proyecto, se describe a continuación, en el capítulo 2, el estado actual de la detección automática de expresiones y locuciones, acompañado de los objetivos iniciales del proyecto. Inmediatamente después, se encuentran las competencias específicas del Grado en Ingeniería Informática más estrechamente relacionadas con el trabajo, seguidas de las aportaciones del mismo.

Acto seguido, se establecen los requisitos del software, cuyo desarrollo se encuentra descrito más adelante, en el capítulo 5, el cual está organizado según los distintos módulos del software y sus correspondientes etapas de desarrollo. En dicho capítulo, se halla también el enlace al código fuente del proyecto, junto con el plan de desarrollo y las herramientas empleadas durante el mismo. Tras ello, podrán encontrarse las conclusiones finales del trabajo, la bibliografía y, por último, el glosario de términos.

Capítulo 2

Estado actual y objetivos iniciales

A continuación, se describirá brevemente el estado actual de la detección automática de expresiones y locuciones en textos, tema central y que da su nombre a este Trabajo de Fin de Título. Seguidamente, se presentarán los objetivos iniciales del trabajo y la solución software que se va a desarrollar, con el fin de alcanzar dichos objetivos.

2.1. Estado del arte

Múltiples diccionarios ofrecen información acerca de las expresiones y locuciones de la lengua española. El más notable de ellos es el Diccionario de la lengua española (DLE), de la Real Academia Española (RAE), la obra lexicográfica académica por excelencia. En su página web, se pueden encontrar, en la entrada de cada palabra, las expresiones y locuciones que la contienen. Para cada una de estas, además, se muestran su significado o significados, categorías y, en ocasiones, uno o más ejemplos de uso. [3]

Sin embargo, actualmente no existe ninguna herramienta de acceso público que permita encontrar, de manera automática y con un margen de error razonable, todas las expresiones y locuciones en un texto en español. Sin embargo, algunas herramientas de Inteligencia Artificial, como los muy conocidos modelos grandes de lenguaje [4] o LLM (por sus siglas en inglés, *Large Language Model*), podrían estar cerca de constituir un sistema de estas características. Modelos de este tipo, como ChatGPT o Claude, tienen la capacidad de capturar buena parte de la sintaxis y semántica del lenguaje, por lo que la posibilidad de analizar textos, con el fin de localizar expresiones y locuciones en el mismo, parece prometedora. No obstante, estas herramientas son de propósito general y no han sido diseñadas para este caso de uso en específico, por lo que los resultados son muy imprecisos.

2.2. Objetivos iniciales

El principal objetivo de este Trabajo de Fín de Título es el desarrollo de una aplicación web, que permita a sus usuarios obtener un informe detallado de las expresiones y locuciones de la lengua española, contenidas en cualquier texto de su elección. Los usuarios tendrán la posibilidad de introducir dicho texto, tanto de forma manual, como subiendo un fichero de texto plano, de extensión `.txt`, por ejemplo. Tras ello, podrán acceder a una página que contendrá la siguiente información para cada expresión o locución encontrada en dicho texto:

- Localización de la expresión o locución en el texto.
- Significado, o significados (en caso de que la expresión o locución tuviera distintas acepciones), junto con la categoría de expresión o locución para cada uno de ellos.
- Ejemplos de uso, si los hubiera.

Por supuesto, será primordial que la información sea presentada de manera ordenada e intuitiva para el usuario.

Asimismo, se deberá tener en cuenta que cada expresión o locución puede tomar muy variadas formas, en función de la conjugación verbal (si contuviera uno o más verbos), las distintas formas flexivas de cada término contenido en la misma, y todas las palabras que esta pudiera contener. Estas variaciones se ilustran en los siguientes ejemplos, donde se utiliza de distintas maneras la misma locución verbal, aparentemente simple, *sentarse alguien a la mesa* - "sentarse, para comer, junto a la mesa destinada al efecto" (Diccionario de la lengua española).

- Poco tardó Pedro en sentarse a la mesa.
- Tuvo que sentar al niño a la mesa.
- Si se hubieran sentado con más tranquilidad a la mesa, eso no hubiera ocurrido.

Como se puede observar, en cada ejemplo se utiliza una conjugación diferente del verbo *sentar*. Además, en cada caso la locución posee en medio de la misma un número distinto de palabras: ninguna en el caso de *sentarse a la mesa* y dos en el caso de *sentar al niño a la mesa*, por ejemplo. Si se tienen en cuenta todas las posibles alternativas, la tarea de localizar de manera precisa todas las expresiones y locuciones de la lengua española en cualquier texto no es, desde luego, trivial.

Para poder lograr dicha funcionalidad, será necesario disponer, en primer lugar, de una base de datos con la información pertinente sobre las expresiones y locuciones. Por ello, el segundo gran objetivo de este trabajo es el desarrollo de una aplicación de escritorio, que permita introducir dicha información a partir de diccionarios de la lengua española, almacenados en ficheros de texto.

Por último, cabe mencionar que, además de la aplicación web descrita anteriormente, la cual pondrá a disposición de los usuarios una interfaz gráfica, se creará también un servicio web. Este hará uso de la base de datos, mencionada en el párrafo anterior, con el fin de

detectar las expresiones y locuciones en cualquier texto de entrada. Extrayendo dicha funcionalidad a un servicio independiente, se logra que puedan acceder a esta tanto la aplicación web como cualquier otro software que realice consultas al servicio. De esta manera, otras aplicaciones podrán integrar el sistema de detección automática de expresiones y locuciones, sin necesidad de duplicar código, ni conocer en profundidad los detalles de implementación del mismo.

Capítulo 3

Competencias específicas y aportaciones del trabajo

Se muestran, en este capítulo, las competencias específicas, correspondientes al plan 41 del Grado en Ingeniería Informática, más estrechamente relacionadas con este trabajo. Tras ello, se presentarán los principales campos en que el proyecto podría ser de utilidad, aquellos ámbitos en los que potencialmente podría ser aplicado, suponiendo una aportación considerable.

3.1. Competencias específicas

Las competencias específicas que mayor relación guardan con este Trabajo de Fin de Título son las CI12, TI3 y TI6. Todas ellas se describen a continuación.

3.1.1. CI12

"Conocimiento y aplicación de las características, funcionalidades y estructura de las bases de datos, que permitan su adecuado uso, y el diseño y el análisis e implementación de aplicaciones basadas en ellos."

Realizar un diseño e implementación efectivos y eficientes de la base de datos será uno de los más cruciales y desafiantes aspectos del desarrollo de este proyecto. La base de datos es el eje central del mismo, pues en esta se almacenará y consultará la información correspondiente a las miles de expresiones y locuciones de la lengua española, así como los patrones que permitirán localizarlas en cualquier texto.

Cabe mencionar que la base de datos a diseñar será relacional, siendo Microsoft Access el sistema de gestión de bases de datos a utilizar.

3.1.2. TI3

"Capacidad para emplear metodologías centradas en el usuario y la organización para el desarrollo, evaluación y gestión de aplicaciones y sistemas basados en tecnologías de la información que aseguren la accesibilidad, ergonomía y usabilidad de los sistemas."

Esta competencia resulta fundamental a la hora de desarrollar cualquier interfaz de usuario, pues la usabilidad y accesibilidad de la misma determinará, en gran medida su utilidad final. En otras palabras, aunque el software posea una funcionalidad y rendimiento excelentes, de poco servirá si nadie es capaz de usarlo. Por supuesto, este aspecto será especialmente relevante en el caso de la aplicación web, el sistema con el que podrán interactuar los usuarios finales, aunque también será un factor a tener en cuenta durante el desarrollo de la aplicación de escritorio.

3.1.3. TI6

"Capacidad de concebir sistemas, aplicaciones y servicios basados en tecnologías de red, incluyendo Internet, web, comercio electrónico, multimedia, servicios interactivos y computación móvil."

Dado que la mayor parte del trabajo consiste en la creación de una aplicación web, además de un servicio también basado en tecnologías web, salta a la vista la importancia de la competencia en cuestión. Esta no se aplica, sin embargo, al desarrollo de la aplicación de escritorio, pues todas las operaciones de la misma se llevarán a cabo de manera local, sin hacer uso de Internet.

3.2. Aportaciones

La detección automática de expresiones y locuciones en textos puede ser extremadamente útil en distintos ámbitos, tanto académicos, como profesionales y tecnológicos. A continuación, se describen algunos de los ejemplos más significativos.

3.2.1. Enseñanza del español

Las expresiones idiomáticas y locuciones son uno de los aspectos más desafiantes al aprender una lengua. Dichas construcciones lingüísticas forman unidades léxicas con un significado propio, que no se deriva del significado literal de las palabras que las forman. A pesar de su inherente dificultad, se utilizan constantemente y son una parte esencial del lenguaje, en especial de las interacciones más cotidianas. Por estos motivos, una herramienta que ofrezca la posibilidad de encontrar de forma automática dichas expresiones y locuciones en cualquier texto, mostrando su significado y ejemplos de uso, podría ser de gran utilidad para estudiantes de la lengua española.

3.2.2. Mejora de la escritura y edición

La detección de expresiones y locuciones podría suponer una ayuda significativa para escritores y editores, permitiéndoles identificar expresiones redundantes o que puedan restar claridad o precisión al texto. Asimismo, podrían verificar que todas las expresiones y locuciones utilizadas sean apropiadas según el registro y formalidad del texto en cuestión, descartando aquellas que resulten demasiado coloquiales, por ejemplo.

3.2.3. Procesamiento del Lenguaje Natural

El Procesamiento del Lenguaje Natural, o NLP por sus siglas en inglés (*Natural Language Processing*), es un campo en constante crecimiento, en buena parte debido al reciente surgimiento de revolucionarias tecnologías relacionadas con la IA (Inteligencia Artificial). Existe la posibilidad de integrar la detección automática de expresiones y locuciones en sistemas de NLP, como chatbots o asistentes virtuales, para facilitar la comprensión de dichos grupos de palabras.

3.2.4. Sistemas de traducción automática

La detección automática de expresiones y locuciones, muy probablemente supondría una mejora en la calidad de la traducción automática. Al capturar el significado completo de dichos grupos de palabras, se evitarían traducciones literales que resultan en pérdida del sentido de las oraciones, un problema de lo más común al utilizar sistemas de estas características.

Capítulo 4

Requisitos del software

Los requisitos se han dividido en dos categorías. En primer lugar, los requisitos funcionales, que describen la funcionalidad indispensable que debe estar presente en cada una de las aplicaciones a desarrollar. No obstante, es posible que las versiones finales contengan algunas características adicionales, como se podrá ver en el capítulo 5, correspondiente al desarrollo del trabajo.

Por otra parte, los requisitos no funcionales determinan restricciones o características imprescindibles del sistema en cuestión, no relacionadas con las funciones del programa. En la mayoría de los casos, estos se encuentran motivados por la necesidad de asegurar la compatibilidad con otras herramientas desarrolladas por el IATEXT.

4.1. Aplicación de escritorio

4.1.1. Requisitos funcionales

- El sistema mostrará, en la pantalla principal, un listado con todos los diccionarios de expresiones y locuciones que han sido añadidos a la base de datos.
- El sistema permitirá ordenar los diccionarios almacenados, tanto en sentido ascendente como descendente, por su nombre, descripción, fecha de creación, formato o ruta de fichero.
- El sistema permitirá la eliminación de un diccionario de expresiones y locuciones, requiriendo la confirmación por parte del usuario antes de efectuar la operación.
- El sistema permitirá añadir un diccionario de expresiones y locuciones, indicando un nombre para el mismo, una descripción (opcional), el formato del diccionario en cuestión (seleccionando uno entre los formatos admitidos) y la ruta al fichero correspondiente.
- Al añadir un nuevo diccionario, el sistema leerá del fichero especificado por el usuario, procesará e insertará en la base de datos todas las expresiones y locuciones que éste

contenga.

- Mientras se lleva a cabo el proceso de creación de un diccionario de expresiones y locuciones, el sistema indicará al usuario que dicha acción se está realizando.
- El sistema ofrecerá la posibilidad de cancelar la operación de creación de un diccionario de expresiones y locuciones, antes de que esta se complete.
- El sistema permitirá seleccionar un diccionario, previamente añadido, y acceder a un listado con todas las expresiones y locuciones que este contiene.
- El sistema permitirá buscar las expresiones y locuciones de un diccionario por nombre.
- El sistema ofrecerá la posibilidad de ordenar alfabéticamente, tanto en sentido ascendente como descendente, las expresiones y locuciones de un diccionario, ya sea por su nombre o por su palabra base, es decir, la palabra correspondiente a la entrada del diccionario en que se encuentra la expresión o locución, *cuenta* en el caso de la locución *por su cuenta*, por ejemplo.

4.1.2. Requisitos no funcionales

- El sistema ha de ser compatible, al menos, con Windows 7, 10 y 11.
- El sistema debe utilizar una base de datos Microsoft Access.
- El sistema ha de ser desarrollado en el ecosistema .NET, utilizando el lenguaje C#.

4.2. Servicio web

4.2.1. Requisitos funcionales

- El sistema, tomando como parámetro un texto, devolverá una colección con las expresiones y locuciones de la lengua española contenidas en dicho texto.
- Para cada locución o expresión encontrada en un texto, el sistema devolverá su nombre, palabra base, localización de la coincidencia en el texto introducido, longitud de la coincidencia en el texto, definición, o definiciones, y ejemplos de uso, si los hubiera.

4.2.2. Requisitos no funcionales

- El servicio debe poder ser utilizado en cualquier aplicación desarrollada en .NET Framework, de la versión 4.7.2 en adelante, incluyendo las versiones actuales de .NET, siendo .NET 8 la más reciente en estos momentos.

- El sistema debe ser desarrollado utilizando la herramienta Windows Communication Foundation (WCF) y el lenguaje C#.
- El sistema debe utilizar una base de datos Microsoft Access.

4.3. Aplicación web

4.3.1. Requisitos funcionales

- El sistema permitirá enviar un texto, cuya longitud ha de estar comprendida entre los valores máximos y mínimos establecidos.
- El sistema permitirá subir un fichero de texto plano, cuyo tamaño no supere el máximo establecido, y extraerá el texto contenido por el mismo.
- El sistema mostrará mensajes de validación en caso de que el usuario haya introducido un texto, o subido un fichero, que no cumpla con las reglas de validación, indicando exactamente cuál de ellas se ha infringido.
- En cuanto el usuario haya introducido un texto o subido un fichero adecuado, el sistema mostrará las locuciones y expresiones de la lengua española incluidas en el texto.
- Para cada locución o expresión encontrada en el texto introducido por el usuario, el sistema mostrará el nombre de la misma, su localización en el texto, definición, o definiciones, y ejemplos, si los hubiera.

4.3.2. Requisitos no funcionales

- Los parámetros de validación del texto y del fichero de texto, es decir, longitud máxima y mínima en el caso del texto, y tamaño máximo en kilobytes y longitud mínima de su contenido en el caso del fichero, han de ser determinados en la configuración de la aplicación, permitiendo su rápida modificación.
- El sistema ha de ser desarrollado utilizando la herramienta Razor Pages y el lenguaje de programación C#.
- El sistema ha de ser compatible, al menos, con las versiones publicadas a partir del año 2020 de Google Chrome, Firefox, Safari y Microsoft Edge.

Capítulo 5

Desarrollo

Tal y como se mencionó en las anteriores secciones, este TFT consiste, casi exclusivamente, en el desarrollo de un software para la detección automática de expresiones y locuciones. Este software se divide en tres módulos claramente diferenciados: una aplicación de escritorio para Windows, que permitirá almacenar dichas expresiones y locuciones, un servicio web para la consulta de las mismas y una aplicación web que proporcione la interfaz gráfica para el usuario final. Este capítulo describe en profundidad todo el proceso de desarrollo y las herramientas empleadas durante el mismo. Para un mejor seguimiento de las explicaciones y los fragmentos de código, puede ser de ayuda consultar el código fuente, escrito en su mayoría en el lenguaje C#.

5.1. Código fuente del proyecto

La totalidad del código, que incluye las tres aplicaciones desarrolladas a lo largo de este Trabajo de Fin de Título, se encuentra disponible en un repositorio público en Github. Puede acceder a él haciendo clic en el siguiente enlace:

https://github.com/Alejandro-M-Cruz/PhraseFinder

El código consiste en varios proyectos que componen una solución de .NET, denominada `PhraseFinder`, que en español se traduce como *Buscador de Expresiones* o *Buscador de Locuciones*. Cada uno de estos proyectos se corresponde con un directorio diferente. Se describen brevemente a continuación:

- `PhraseFinder.Domain` - Capa de dominio. Incluye las clases que representan los modelos, además de la lógica que permite la lectura del fichero y procesamiento de las expresiones y locuciones, previas a su almacenamiento en la base de datos.
- `PhraseFinder.Domain.Tests` - Proyecto xUnit que contiene los tests para la capa de dominio.

- `PhraseFinder.Data` - Capa de datos. Contiene los servicios que permiten interactuar con la base de datos. Hace uso de los modelos y servicios declarados en la capa de dominio.
- `PhraseFinder.Data.Tests` - Tests para la capa de datos.
- `PhraseFinder.WPF` - Aplicación de escritorio para Windows, desarrollada en WPF. Utiliza tanto la capa de dominio, como los servicios de la capa de datos que almacenan las expresiones y locuciones.
- `PhraseFinder.WCF` - Servicio web WCF que permite obtener las expresiones y locuciones a partir de cualquier texto. Incluye la lógica de detección.
- `PhraseFinder.WebApp` - Aplicación web, desarrollada en Razor Pages, que proporciona una interfaz de usuario sobre el servicio WCF.

Todos ellos utilizan la versión .NET 8.0, exceptuando el servicio WCF, que ha sido desarrollado en .NET Framework 4.7.2 por motivos de compatibilidad, tal y como se explicó en la sección 5.2.2.2.

Para inspeccionar el código de manera local, es posible descargarlo como fichero ZIP, a través de Github. No obstante, si se dispone de la herramienta Git, basta con ejecutar la siguiente orden en la interfaz de línea de comandos del sistema:

```
git clone https://github.com/Alejandro-M-Cruz/PhraseFinder.git
```

Si se desea probar la aplicación de escritorio, lo ideal es utilizar el editor Visual Studio 2022, habiendo instalado, por medio del instalador de Visual Studio (Visual Studio Installer), las herramientas de desarrollo de aplicaciones de escritorio. Una vez hecho esto, la aplicación puede ser ejecutada tanto haciendo uso de la interfaz gráfica de Visual Studio, como por medio del comando `dotnet run`, que ha de ser ejecutado en la carpeta base del proyecto en cuestión, denominada `PhraseFinder.WPF`.

Tan solo es posible ejecutar la aplicación de escritorio, pues el servicio de detección de expresiones y locuciones utiliza ciertas librerías de código cerrado, desarrolladas por el IATEXT, las cuales no se encuentran en el repositorio. Téngase en cuenta, además, que la aplicación de escritorio **solo es compatible con Windows**.

5.2. Herramientas empleadas

Para el desarrollo de este proyecto, se ha hecho uso de una considerable variedad de herramientas, en su mayoría dentro del ecosistema .NET y utilizando C# como lenguaje de programación principal.

5.2.1. Lenguaje de programación

El lenguaje de programación es probablemente la elección más importante a la hora de comenzar a crear un software, pues es el aspecto más difícil de modificar una vez este se encuentra en una fase avanzada del desarrollo. Aunque se utilice una metodología como la arquitectura hexagonal, cuyo enfoque radica en el desacoplamiento entre el código y los frameworks y librerías utilizados, es imposible, por definición, desacoplar el código del lenguaje en el que se escribe.

No obstante, esto no supone un gran inconveniente al utilizar **C#**, un muy popular lenguaje de programación de alto nivel, propósito general y orientado a objetos, que además destaca por su estabilidad, seguridad y por proporcionar un buen rendimiento. Además, cuenta con un amplísimo ecosistema y una gran comunidad de programadores. No resulta sorprendente, por tanto, que sea una opción muy popular entre las empresas de software [5] y el lenguaje elegido para este proyecto. Como ventaja adicional, el conocimiento de C# es de gran valor en muchos ámbitos dentro de la industria, tanto en el mundo de la web, como en el ecosistema desktop y móvil, e incluso en desarrollo de videojuegos, con el motor Unity.

Se utilizarán la última versión estable del lenguaje, C# 12, excepto en el caso del servicio WCF, en el cual, por motivos de compatibilidad, se hará uso de C# 7.0.

5.2.2. Framework

El entorno **.NET** es tan amplio y variado que puede resultar confuso. En sus primeras versiones, era conocido como .NET Framework, un software propiedad de Microsoft y de código cerrado, lanzado a comienzos de 2002 [6]. Sin embargo, este proyecto ha pasado a un segundo plano en favor de .NET, aunque todavía recibe actualizaciones de seguridad. .NET es un framework de código abierto, también desarrollado por Microsoft. Fue publicado en 2016, inicialmente bajo el nombre de .NET Core [7]. Su versión más reciente es la 8.0.2 (lanzada en febrero de 2024) y cuenta con múltiples ventajas respecto a .NET Framework, cuya última versión es la 4.8.1 (publicada en septiembre de 2022). Entre dichas ventajas se encuentran un mejor rendimiento y, la más notable, la posibilidad de ejecutarse en MacOS y Linux, mientras que .NET Framework solo funciona en sistemas Windows. Las herramientas utilizadas en este TFT, exceptuando WCF, pertenecen al ecosistema .NET, específicamente a su versión 8.0, aunque todas ellas tienen su equivalente en .NET Framework. Se detallan, a continuación, dichas herramientas.

5.2.2.1. Aplicación de escritorio

En primer lugar, se va a desarrollar una aplicación Windows nativa. Windows Presentation Foundation (**WPF**) es probablemente el framework más utilizado para este propósito, siendo una tecnología muy madura y que cuenta con una gran cantidad de paquetes que permiten extender sus funcionalidades. Además, tiene un buen soporte para la arquitectura MVVM (Model-View-ViewModel), la más utilizada en esta plataforma. Esta permite mantener una clara separación de responsabilidades entre las vistas, declaradas en código XAML, y las clases del modelo, al mismo tiempo que expone las interfaces necesarias a la vista para que esta muestre y actualice correctamente la información deseada.

No obstante, durante la primera etapa de análisis del proyecto, se consideró la posibilidad de utilizar la librería WinUI 3 para el desarrollo de la aplicación de escritorio. Se trata de un framework mucho más reciente que WPF y forma parte del Windows App SDK [8]. Sin embargo, esta librería está en pleno desarrollo y no ofrece tantas herramientas, robustez y estabilidad como WPF. Lo mismo ocurre, por tanto, con .NET MAUI, una herramienta multiplataforma que utiliza WinUI 3 en su versión para Windows.

También se tuvieron en cuenta otros entornos para el desarrollo de aplicaciones multiplataforma, en especial Avalonia UI, un framework de código abierto que permite desplegar la misma aplicación de escritorio en Windows, MacOS y Linux. Sin embargo, la app en cuestión solo va a ser utilizada en el sistema operativo Windows, lo que permite esquivar la complejidad añadida que conlleva desarrollar para distintas plataformas.

5.2.2.2. Servicio web

Con el fin de desarrollar el servicio web, que permitirá encontrar las expresiones y locuciones localizadas en un texto cualquiera, se hará uso de Windows Communication Foundation, o **WCF**, un framework diseñado específicamente para construir aplicaciones orientadas a exponer servicios, y al envío de mensajes asíncronos. Esta tecnología es algo antigua y, a día de hoy, poco popular en comparación con otras opciones dentro del ecosistema .NET, como ASP.NET Web API o gRPC [9].

No obstante, presenta una muy atractiva ventaja en este caso: la compatibilidad y facilidad de integración, tanto en proyectos desarrollados en la implementación antigua del entorno, .NET Framework, como en la más reciente, .NET. Esta característica es crucial, dado que el servicio va a ser utilizado por otras aplicaciones del IATEXT, las cuales han sido construidas en .NET Framework, principalmente en su versión 4.7.2. Por tanto, será esta versión la utilizada para crear la aplicación WCF, que expondrá un servicio web IIS (*Internet Information Services*, servidor web desarrollado por Microsoft para sistemas Windows). Dicho servicio será consumido por la aplicación web, que se comentará en la siguiente subsección.

Por último, cabe comentar que este sistema hará uso de un servicio de *lematización*, desarrollado por el IATEXT, que será una pieza fundamental en la lógica de detección de expresiones y locuciones, pues aportará información imprescindible acerca de las categorías

gramaticales y formas canónicas de las palabras en el texto introducido. Además de este, se utilizará una librería, también del IATEXT, para la separación del texto en párrafos, oraciones y palabras.

5.2.2.3. Aplicación web

Se utilizará **ASP.NET Core**, tecnología que permite el desarrollo de aplicaciones web dentro del entorno .NET. En concreto, se creará un proyecto **Razor Pages**, siendo Razor la sintaxis que utiliza .NET para integrar código C#, o Visual Basic, dentro de las páginas HTML. Razor Pages tiene un enfoque orientado hacia la rapidez de desarrollo de webs centradas en páginas, ofreciendo una mayor productividad para proyectos de poca complejidad que su principal alternativa, ASP.NET Core MVC [10].

La web a desarrollar proporcionará una intuitiva interfaz gráfica, desde la cual el usuario podrá interactuar con el servicio WCF, lo que le permitirá localizar las expresiones y locuciones de cualquier texto de su elección. El diseño de la misma se construirá utilizando principalmente Boostrap 5.1, pues viene integrado por defecto en el proyecto Razor Pages y ofrece una agradable y rápida experiencia de desarrollo. También se escribirá CSS, cuando sea necesario aplicar estilos muy específicos, o hacer uso de ciertas consultas de medios (*media queries*), que no puedan realizarse con Bootstrap.

Para finalizar, en las secciones de la aplicación que requieran de interactividad en el lado del cliente, se utilizará, por supuesto, Javascript. No obstante, se intentará escribir el mínimo indispensable, manteniendo la mayor parte de la lógica en el lado del servidor.

5.2.2.4. Testing

En el ecosistema .NET, destacan tres frameworks para testing: xUnit, NUnit y MSTest [11]. xUnit es un framework de código abierto que simplifica al máximo la creación de pruebas unitarias. Fue desarrollado por los creadores de NUnit, también muy popular y de código abierto, aunque considerablemente más antiguo. La tercera opción, MSTest, es la utilizada por defecto en Visual Studio. Su versión actual es también de código abierto, recibe actualizaciones frecuentes y sigue siendo muy utilizado, a pesar de ser el más antiguo de los tres [12].

No obstante, xUnit presenta algunas ventajas respecto a las dos alternativas mencionadas. En primer lugar, ofrece un mayor aislamiento, dado que la clase de test vuelve a inicializarse para cada caso, lo que garantiza la independencia de las pruebas unitarias entre sí, aunque también proporciona distintas herramientas para compartir contexto entre varios tests, como los *test fixtures*. Además, su uso es algo más simple y menos verboso, aunque las tres herramientas son muy similares. Por ello, **xUnit** ha sido la herramienta utilizada para el desarrollo de pruebas unitarias, las cuales se han centrado en verificar el correcto funcionamiento de la capa de dominio del proyecto y la capa de datos, los módulos más fundamentales del código, pues de estos dependen la aplicación de escritorio y, en gran medida, la calidad de la detección de las expresiones y locuciones.

5.2.3. Base de datos

La base de datos empleada para el almacenamiento de todas las expresiones y locuciones ha sido **Microsoft Access**. Esto es debido, principalmente, a que la mayor parte del software ya desarrollado por el IATEXT utiliza este sistema. Además, cuenta con otra gran ventaja: permite la interacción con los datos mediante interfaz gráfica, proporcionando también una gran variedad de opciones para ordenar, filtrar, seleccionar y realizar búsquedas complejas sobre la información almacenada. Aparte de esto, Access es una base de datos relacional, lo cual es tremendamente conveniente para este proyecto, pues existirán múltiples relaciones *uno a muchos* entre las distintas tablas, como la existente entre cada expresión o locución y sus correspondientes definiciones y ejemplos.

Por último, este sistema almacena la base de datos al completo en un único fichero, con extensión `.accdb`. Dicha característica es muy deseable para este caso de uso, pues clonar la base de datos para utilizarla en diferentes aplicaciones es verdaderamente sencillo, requiriendo tan solo copiar un archivo.

Como no podía ser de otra manera, Access presenta también algunos inconvenientes. En primer lugar, no es una base de datos muy ampliamente utilizada en el entorno .NET, como sí lo son SQL Server, MySQL, PostgreSQL o SQLite, por lo que la documentación no es tan exhaustiva como cabría desear. Asimismo, las extensiones que permiten interactuar con la misma desde código C# no reciben actualizaciones tan frecuentes, aunque esto no supone un gran problema, pues las últimas versiones estables son compatibles con la versión de .NET utilizada, la 8.0.

Por otra parte, para la interacción con la base de datos no se realizarán, de manera directa, consultas SQL, sino que se hará uso de un ORM (*Object-Relational Mapper*). Este tipo de herramienta presenta numerosas ventajas, como una considerable aceleración del proceso de desarrollo y una mayor seguridad y fiabilidad [13], al evitar construir las sentencias SQL manualmente, una práctica que puede exponer el software a errores al teclear y a ataques como la inyección SQL. Concretamente, se empleará la última versión estable (8.0) de Entity Framework Core, el ORM de uso más extendido en el entorno .NET, también desarrollado por Microsoft. Además, es fácil de integrar, siendo, incluso, la opción por defecto al crear una aplicación en ASP.NET.

5.2.4. Editor

El principal editor de código utilizado ha sido **Visual Studio**, un IDE (*Integrated Development Environment* o Entorno de Desarrollo Integrado) orientado al desarrollo en .NET, y que soporta los lenguajes C#, F# y Visual Basic, entre otros. Específicamente, he utilizado la versión de 2022, la más reciente a la fecha de realización de este TFT.

La interfaz gráfica de Visual Studio puede resultar algo confusa, pues proporciona una enorme variedad de opciones. Muchas de ellas son de gran utilidad; las comento a continuación:

- **IntelliSense.** El editor proporciona autocompletado del código y sugerencias sumamente precisas. Esto se debe, en buena parte, al sistema de tipos estático de C#.
- **Gestión del administrador de paquetes.** Visual Studio permite gestionar fácilmente las dependencias del proyecto y agregar bibliotecas de terceros utilizando el repositorio NuGet.
- **Integración con Github Copilot.** El editor ofrece integración con las principales características del asistente Github Copilot, como las sugerencias de código y la función de *chatbot*.
- **Ejecución automática de pruebas.** Facilita la escritura y ejecución de pruebas unitarias directamente desde el entorno de desarrollo.
- **Análisis de rendimiento.** Visual Studio ofrece herramientas para analizar el rendimiento de la aplicación y detectar cuellos de botella, lo que ayuda a optimizar el código y mejorar la experiencia del usuario.
- **Extensiones.** Se puede personalizar el entorno de desarrollo según las necesidades específicas del proyecto y utilizar una amplia gama de extensiones disponibles en el Visual Studio Marketplace.

5.2.5. Sistema de Control de Versiones

El sistema de control de versiones es una tecnología muy útil a la hora de gestionar los cambios realizados en un software. En concreto, se hará uso de **Git**, la herramienta de control de versiones más utilizada por programadores. Asimismo, como repositorio remoto se empleará la plataforma Github. Esta presenta diversas ventajas, como su gratuidad y la posibilidad de automatizar flujos de trabajo por medio de Github Actions, que permite, por ejemplo, implementar procesos de integración continua y despliegue del software.

Existen distintas metodologías para trabajar con un sistema de este tipo. Una de las más conocidas es *Feature Branch Workflow*, que consiste en crear ramas separadas para cada tarea, a partir de la rama principal (comúnmente denominada *main* o *master*). Una vez dicha tarea ha sido completada y validada, esta rama se funde o *mergea* con la rama principal y se lanzan los nuevos cambios al entorno de producción [14].

Si bien este enfoque es excelente para un equipo de programadores, es innecesario para este trabajo, pues se trata de un proyecto individual y la rama principal no va a ser puesta en producción hasta finalizar el mismo. En este caso no supone un gran inconveniente trabajar únicamente en dicha rama. Aún así, ante cualquier incidente, Git proporciona diversas órdenes que permiten deshacer cambios o volver a un *commit* anterior. El ejemplo más ilustrativo es posiblemente la orden *revert*, la cual toma como parámetro el ID de un commit y deshace por completo los cambios realizados en el mismo, incluso si dicho commit ya ha sido subido al repositorio remoto.

5.3. Plan de desarrollo

El desarrollo del software en cuestión se encuentra dividido en tres aplicaciones diferentes: la aplicación de escritorio WPF, el servicio web WCF y la aplicación web Razor Pages. Primero se trabajará en la aplicación de escritorio, pues esta permitirá introducir toda la información correspondiente a las locuciones y expresiones en la base de datos. Por supuesto, esto requerirá crear, previamente, los modelos y servicios necesarios en la capa de dominio para leer las expresiones y locuciones de un fichero de texto. El fichero de texto en cuestión es un archivo de texto plano, con extensión `.txt`, que contiene todas las entradas del Diccionario de la lengua española, en un formato determinado. También será imprescindible desarrollar distintos servicios en la capa de datos, que permitan almacenar la información en la base de datos.

Seguidamente, se desarrollará la aplicación web. Inicialmente, se utilizarán datos de ejemplo, pues el objetivo será diseñar la interfaz de usuario, con un especial enfoque en la simplicidad y usabilidad. Esta aplicación solo dependerá de la interfaz expuesta por el servicio web WCF, que tomará como parámetro un texto y devolverá las expresiones y locuciones de la lengua española encontradas en el mismo.

Por último, se creará la lógica de detección de expresiones y locuciones, dentro del servicio WCF. Este hará uso de la base de datos, creada por medio de la aplicación de escritorio. Una vez se haya desarrollado un prototipo funcional del mismo, será posible conectar este servicio con la aplicación web, y utilizar finalmente la herramienta.

En cada uno de los tres casos, el desarrollo se llevará a cabo siguiendo una metodología basada en prototipos. En cuanto la aplicación en cuestión incluya toda la funcionalidad imprescindible, listada en sus correspondientes requisitos, se considerará terminado el primer prototipo de la misma, y se procederá a su validación. Posteriormente, si el tiempo de desarrollo no se extiende más de lo esperado, estas primeras versiones funcionales recibirán mejoras y características adicionales, desarrollándose subsiguientes prototipos hasta alcanzar las versiones finales.

5.4. Desarrollo de la capa de dominio

La capa de dominio contendrá los modelos de la aplicación, que incluirán la información pertinente de los diccionarios y de las locuciones y expresiones, junto con sus definiciones y ejemplos, además de los patrones utilizados para la detección de las mismas.

Asimismo, la capa de dominio contendrá la lógica de obtención de las expresiones y locuciones, a partir de un fichero con la información del Diccionario de la lengua española, seguida de los servicios necesarios para el procesamiento de las mismas, previo a su almacenamiento en la base de datos.

5.4.1. Obtención de las expresiones y locuciones

En primer lugar, se deben desarrollar los módulos necesarios en la capa de dominio para obtener la información de las expresiones y locuciones de la lengua española, a partir de un fichero de texto. Tal y como se mencionó en la sección 5.3, el fichero a utilizar es un archivo de texto plano, con extensión `.txt`, que contiene todas las entradas del Diccionario de la lengua española de la Real Academia Española. Las entradas están separadas por una línea vacía (`"\r\n"`) y tienen el siguiente formato:

```
zarpa#zarpa[1]
[Etim]Del ant. farpa 'pingajo, jirón', infl. por el sinónimo zarria.
1. f. Mano de ciertos animales cuyos dedos no se mueven con independencia
unos de otros, como en el león y el tigre.
[Sin]garra, mano, pezuña.
2. f. Lodo o barro que se queda en la parte baja de la ropa.
[loc6]echar alguien la zarpa
1. loc. verb. coloq. Agarrar o asir con las manos o las uñas.
2. loc. verb. coloq. Apoderarse de algo por violencia, engaño o sorpresa.
[Ejem]Le echó la zarpa al último dulce.
[loc6]hacerse alguien una zarpa
1. loc. verb. coloq. desus. Mojarse o enlodarse mucho.
```

En total, hay unas 13500 expresiones y locuciones incluidas en el fichero. Inmediatamente a la izquierda de cada una de ellas, se halla la etiqueta `[loc6]`, encontrándose sus categorías y definiciones en las líneas inmediatamente posteriores. Por último, los ejemplos tienen la etiqueta `[Ejem]`. Algunas definiciones cuentan uno o varios ejemplos. Por tanto, el primer modelo a crear será una clase que contenga todos estos datos. Esta se denomina `PhraseEntry` y cada una de sus instancias representa una entrada de expresión o locución en el diccionario. Se muestra la clase a continuación:

Fragmento de código 5.1: Clase `PhraseEntry`, que representa una entrada de expresión o locución en el diccionario.

```
1 public class PhraseEntry
2 {
3     public required string Name { get; set; }
```

```
4       public required string BaseWord { get; set; }
5       public ISet<string> Categories { get; set; } = new HashSet<string>();
6       public IDictionary<string, ICollection<string>> DefinitionToExamples {
            get; } =
7           new Dictionary<string, ICollection<string>>();
8
9       public Phrase ToPhrase()
10      {
11          return new Phrase
12          {
13              // ...
14          };
15      }
16
17      public override bool Equals(object? obj)
18      {
19          if (obj is not PhraseEntry other)
20          {
21              return false;
22          }
23
24          return Name == other.Name && BaseWord == other.BaseWord &&
25              Categories.SequenceEqual(other.Categories) &&
26              DefinitionToExamplesEquals(other.DefinitionToExamples);
27      }
28
29      private bool DefinitionToExamplesEquals(IDictionary<string,
            ICollection<string>> other)
30      {
31          return DefinitionToExamples
32              .All(d => other[d.Key].SequenceEqual(d.Value));
33      }
34
35      public override int GetHashCode()
36      {
37          return HashCode.Combine(Name, BaseWord, DefinitionToExamples);
38      }
39  }
```

Como se puede observar, esta primera clase del modelo contiene toda la información proporcionada por cada entrada de expresión o locución del diccionario. Además, incluye un método `ToString()`, con el fin de obtener una salida por pantalla más legible a la hora de ejecutar los tests de la capa de dominio, y los métodos `Equals()`, `GetHashCode()` y `DefinitionToExamplesEquals()`, que permiten comprobar la igualdad de distintas instancias de la clase, lo cual también será de utilidad en los tests. Por último, contiene un método denominado `ToPhrase()`, que devuelve una instancia correspondiente del modelo `Phrase`, el cual se comentará en la subsección 5.4.2. Este método se utilizará más adelante, al momento de almacenar las locuciones y expresiones en la base de datos.

A continuación, se desarrolla el servicio necesario para realizar la lectura de las expresiones y locuciones del fichero. El diseño elegido contempla la posibilidad de extender la aplicación para soportar distintos formatos de diccionario, aunque el utilizado para este proyecto sea

el Diccionario de la lengua española (DLE), concretamente en el formato de texto plano con extensión `.txt`, descrito anteriormente. Para ello, se añade una enumeración denominada `PhraseDictionaryFormat`, con una única constante: `DleTxt`, que representa el formato del DLE. Seguidamente, se añade la interfaz `IPhraseDictionaryReader`, la cual será implementada por todas las clases cuyo propósito sea leer diccionarios con un determinado formato. Dicha interfaz contiene el siguiente método:

Fragmento de código 5.2: Firma del único método de la interfaz `IPhraseDictionaryReader`.

```
1 public IAsyncEnumerable<PhraseEntry> ReadPhraseEntriesAsync();
```

Por último, se añade el servicio que lee las expresiones y locuciones del DLE, cuya implementación se muestra, algo simplificada, a continuación:

Fragmento de código 5.3: Implementación simplificada del servicio `DleTxtPhraseDictionaryReader`, que lee las entradas de expresión y locución de un fichero con la información del DLE.

```
1  public class DleTxtPhraseDictionaryReader(string filePath) :
       IPhraseDictionaryReader
2  {
3      private const string PhraseTag = "[loc6]";
4      private const string PhraseExampleTag = "[Ejem]";
5
6      public static readonly Regex EntryRegex = new(@"^.+#\w+", RegexOptions
           .Compiled);
7      public static readonly Regex PhraseDefinitionRegex = new(
8          @"^\d+\. ((?:loc\.|locs\.|expr\.|exprs\.) (?:[a-z]+\. )*)",
9          RegexOptions.Compiled);
10     private static readonly Regex EntryNumberRegex = new(@"\[\d+\]",
           RegexOptions.Compiled);
11
12     public async IAsyncEnumerable<PhraseEntry> ReadPhraseEntriesAsync(
13         [EnumeratorCancellation]
14         CancellationToken cancellationToken = default)
15     {
16         using var reader = new StreamReader(filePath);
17         string? currentLine;
18         string? currentWord = null;
19         PhraseEntry? currentPhraseEntry = null;
20         string? currentPhraseDefinition = null;
21         Match phraseDefinitionMatch;
22
23         while (
24             (currentLine = await reader.ReadLineAsync(cancellationToken))
                   != null)
25         {
26             var phraseDefinitionMatch = PhraseDefinitionRegex
27                 .Match(currentLine);
28
29             if (string.IsNullOrWhiteSpace(currentLine) &&
30                 currentPhraseEntry != null)
31             {
32                 // Fin de entrada del diccionario
```

```
33                  // ...
34              }
35              else if (EntryRegex.IsMatch(currentLine))
36              {
37                  // Nueva entrada de palabra en el diccionario
38                  // ...
39              }
40              else if (currentLine.StartsWith(PhraseTag))
41              {
42                  // Nueva expresión o locución en la entrada actual
43                  // ...
44              }
45              else if (
46                  (phraseDefinitionMatch = PhraseDefinitionRegex.Match(
                       currentLine)).Success &&
47                  currentPhraseEntry != null)
48              {
49                  // Definición de la expresión o locución actual
50                  currentPhraseDefinition = currentLine;
51                  currentPhraseEntry.DefinitionToExamples.Add(
52                      currentPhraseDefinition,
53                      []);
54                  currentPhraseEntry.Categories.Add(
55                      phraseDefinitionMatch.Groups[1].Value.TrimEnd());
56              }
57              else if (currentLine.StartsWith(PhraseExampleTag) &&
58                  currentPhraseDefinition != null)
59              {
60                  // Ejemplo de uso de la expresión o locución
61                  currentPhraseEntry?
62                      .DefinitionToExamples[currentPhraseDefinition]
63                      .Add(currentLine[PhraseExampleTag.Length..]);
64              }
65          }
66          if (currentPhraseEntry?.Categories.Count > 0)
67          {
68              yield return currentPhraseEntry;
69          }
70      }
71  }
```

El método `ReadPhraseEntriesAsync()` consiste en un bucle que recorre el fichero del diccionario línea por línea, determinando si la línea actual corresponde a una nueva palabra, a una expresión o locución, a una definición de la misma, o a un ejemplo de uso. Estos datos se recopilan en un objeto de tipo `PhraseEntry` y, cuando no queda más información por leer para la entrada actual, se devuelve dicho objeto. Por tanto, el tiempo de ejecución del método crece de forma lineal con el número de líneas del fichero, es decir, la complejidad temporal del algoritmo, en notación de Landau, conocida coloquialmente como Notación O Grande [15], es O(n), donde n es el número de líneas en el fichero de entrada.

La complejidad espacial (la memoria requerida por el algoritmo) es, en este caso, más difícil de analizar a simple vista. En primer lugar, el método no declara ningún tipo de

colección. Esto es posible gracias a la sentencia `yield return`, la cual convierte el método en un iterador, o `Iterator`. En C#, los iteradores y la sentencia `yield return` permiten crear métodos que generan secuencias de valores según estos son solicitados, sin necesidad de almacenar todos los valores en la memoria al mismo tiempo [16]. Conceptualmente, son muy similares a los generadores y la sentencia `yield` en el lenguaje Python [17].

En esencia, `yield return` devuelve el siguiente elemento de la secuencia, tras lo cual el método no finaliza, tan solo se pausa su ejecución y se guarda su estado actual (variables locales, instrucción actual, ...). Una vez que el código que llama al método en cuestión ha recibido el nuevo valor, este puede solicitar el siguiente, momento en el cual el iterador continúa su ejecución donde se pausó anteriormente [18]. Por lo general, esta secuencia de valores se lee por medio de la instrucción `foreach`, que automáticamente obtiene el siguiente valor y lo almacena en una variable, hasta que finaliza el iterador. En este caso, para hacer uso del método `ReadPhraseEntriesAsync`, se podría ejecutar el siguiente código:

Fragmento de código 5.4: Código que itera sobre los valores generados por el método `ReadPhraseEntriesAsync()`.

```
1 await foreach (var phraseEntry in dleTxtReader.ReadPhraseEntriesAsync())
2 {
3     /* ... */
4 }
```

Nótese el uso de la palabra reservada `await`. Dado que la secuencia generada por el método es asíncrona, cada valor ha de ser *esperado* por el código que lo solicita. Esta es una característica muy relevante, dado que este método va a ser utilizado por la aplicación de escritorio. De no ser asíncrono, la interfaz de usuario y otros procesos de dicha aplicación quedarían bloqueados durante la ejecución del método, que suele tomar algo más de un segundo en el equipo utilizado para el desarrollo, puesto que el fichero tiene un tamaño considerable, unos 28 MB. Además, dado que la aplicación tendrá la posibilidad de cancelar la operación, se ha incluido un parámetro opcional al método, que consiste en un *token* de cancelación [19], el cual es suministrado a la llamada asíncrona a `ReadLineAsync()`.

No obstante, aunque no se declare ninguna, sí que se utilizan colecciones en el método `ReadPhraseEntriesAsync()`, como se puede observar en las líneas líneas 51 a 63 del fragmento de código 5.3. En concreto, se añaden elementos al diccionario `DefinitionToExamples` y al conjunto `Categories`, propiedades de la instancia de la clase `PhraseEntry` almacenada en la variable `currentPhraseEntry`. Debido a esto, en cualquier momento de la ejecución del método, la memoria utilizada por el mismo aumenta linealmente con el número de definiciones y ejemplos que tenga la expresión o locución actual. En teoría, por tanto, la complejidad espacial podría ser lineal. Sin embargo, en la práctica, el diccionario contiene un número muy limitado de definiciones y ejemplos para cada expresión y locución, es decir, la complejidad espacial tenderá a ser constante, O(1) en Notación O Grande.

Por último, es pertinente mencionar que la clase `DleTxtPhraseDictionaryReader`, junto con cualquier otra clase cuya función sea obtener las expresiones y locuciones de un diccionario, independientemente del formato, implementará la interfaz `IPhraseDictionaryReader` y será instanciada a través de un método en una clase independiente, denominado `CreateReader`

`()`. A este método se le suministran como argumentos el formato del diccionario a leer y la ruta del fichero correspondiente, tras lo cual este devuelve un objeto que permite obtener las locuciones y expresiones de dicho diccionario. De esta manera, el resto la lógica de la aplicación es totalmente independiente de las particularidades de la lectura de cada formato, permitiendo agregar nuevos tipos de diccionario fácilmente. Esta estrategia de desacoplamiento sigue el patrón de diseño *Factory*, una versión simplificada del también muy popular *Factory Method* [20]. El código correspondiente a la clase en cuestión se encuentra a continuación, en el fragmento 5.5.

Fragmento de código 5.5: Clase `PhraseDictionaryReaderFactory`, cuyo único método devuelve una instancia de la clase que permite leer las expresiones y locuciones de un diccionario, en base al formato del mismo..

```
1  public static class PhraseDictionaryReaderFactory
2  {
3      public static IPhraseDictionaryReader CreateReader(
4          PhraseDictionaryFormat format,
5          string filePath)
6      {
7          return format switch
8          {
9              PhraseDictionaryFormat.DleTxt =>
10                 new DleTxtPhraseDictionaryReader(filePath),
11             _ => throw new InvalidEnumArgumentException(
12                 nameof(format),
13                 (int)format,
14                 typeof(PhraseDictionaryFormat))
15         };
16     }
17
18 }
```

5.4.2. Modelo de expresión o locución

Una vez que la información de las expresiones y locuciones ha sido extraída del fichero, se construyen los objetos de la clase `Phrase`, que representa las locuciones y expresiones tal y como se almacenan en la base de datos, siendo, por tanto, el modelo central de todo el proyecto. Dicha clase se muestra en el fragmento 5.6.

Fragmento de código 5.6: Clase `Phrase`, correspondiente al modelo de expresión o locución.

```
1  [Table("Locuciones_y_expresiones")]
2  public class Phrase
3  {
4      [Column("ID_Locucion")]
5      public int PhraseId { get; set; }
6
7      [Column("Locucion_o_expresion")]
8      public required string Value { get; init; }
9
10     [Column("Palabra_base")]
```

```
11      [MaxLength(255)]
12      public required string BaseWord { get; init; }
13
14      [Column("Categorias")]
15      [MaxLength(255)]
16      public required string Categories { get; init; }
17
18      [Column("Revisado")]
19      public bool Reviewed { get; set; } = false;
20
21      [Column("ID_Diccionario")]
22      public int PhraseDictionaryId { get; set; }
23
24      public ICollection<PhraseDefinition> Definitions { get; set; } = [];
25
26      public ICollection<PhrasePattern> Patterns { get; set; } = [];
27
28      // ...
29  }
```

5.4.3. Procesamiento de las expresiones y locuciones

Una vez obtenida la información de las expresiones y locuciones, a partir del fichero correspondiente, es momento de procesarlas, de cara a su detección en cualquier texto. Para ello, es necesario procesar el valor de la expresión o locución, es decir, el nombre de la misma según aparece en la entrada del diccionario, que no siempre coincidirá con la manera en que esta puede encontrarse en un texto. Algunos ejemplos sencillos podrían ser los siguientes:

- **a mesa puesta**
- **por cierto**
- **por qué**

Buscar estas tres locuciones en un texto es una tarea de lo más simple, puesto que no varían en absoluto, es decir, ninguno de los términos que estas contienen puede flexionarse en género o número. No obstante, existen muchas otras expresiones y locuciones que acarrean una complejidad mucho más considerable, como las listadas a continuación:

- **sentarse alguien a la mesa**
- **oír, sentir, o ver alguien crecer, o nacer, la hierba**
- **meterse alguien donde no lo llaman, o donde nadie lo llama, o en lo que no le importa, o en lo que no le toca, o en lo que no le va ni le viene**

Todas ellas pueden tomar numerosas formas diferentes, dependiendo de sus distintas variantes, la conjugación de los verbos que contienen y las palabras con las que se combinan, entre otros parámetros. En la sección 2.2 se mostró un ejemplo de estas variaciones, utilizando la locución verbal *sentarse alguien a la mesa*. Si bien es cierto que a partir del nombre de la expresión o locución no es posible, por supuesto, conocer las conjugaciones adecuadas

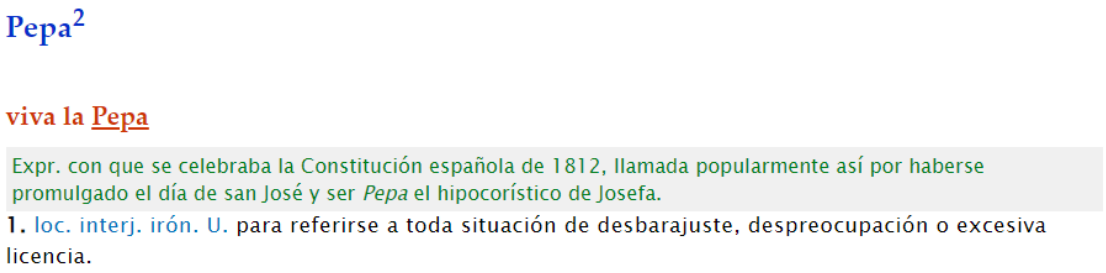

Figura 5.1: Entrada de la palabra Pepa en el Diccionario de la lengua española de la Real Academia Española [3].

de los verbos, o las categorías correspondientes a las palabras que contienen, sí es posible, al menos, obtener las distintas variantes que se encuentren especificadas. Por ejemplo, en el caso de la locución verbal **quitar, o recoger, la mesa**, se pueden obtener **quitar la mesa** y **recoger la mesa**.

Por ello, se han desarrollado dos módulos de la capa de dominio, destinados a procesar estas expresiones y locuciones, obteniendo sus diferentes variantes y eliminando los elementos no deseados. En las siguientes subsecciones, se mostrarán los distintos casos que han sido resueltos.

5.4.3.1. Expresiones y locuciones que incluyen información adicional

En primer lugar, algunas locuciones y expresiones contienen elementos adicionales, que no deben tenerse en cuenta a la hora de buscarlas en un texto. En ocasiones, el Diccionario de la lengua española incluye breves anotaciones que aportan más información sobre la expresión o locución en cuestión, como en el caso del texto de color verde en la figura 5.1.

Estas anotaciones se encuentran también en el nombre de las locuciones y expresiones del fichero, por lo que es necesario descartarlas. Para ello, se ha creado una interfaz, denominada `IPhraseCleaner`, que contiene un único método, `CleanPhrase()`. Este toma como parámetro la cadena de caracteres correspondiente al nombre de la expresión o locución, del cual deben extraerse los elementos no necesarios para su detección. A su vez, se ha creado una clase, llamada `DleTxtPhraseCleaner`, que implementa dicha interfaz. Tal y como se comentó en anteriores secciones, es posible que en un futuro se decida añadir más diccionarios, con un formato diferente al que se está utilizando en este trabajo. Es por ello que se utilizan este tipo de interfaces, que desacoplan al resto de la aplicación del formato utilizado en cada momento.

En el caso del fichero de texto con la información del Diccionario de la lengua española (formato `DleTxt`), existen distintas maneras en que se incluye la información que se pretende eliminar de las expresiones y locuciones. Se muestran, a continuación, algunos ejemplos:

- **dar algo, especialmente la ropa, de sí** - En este caso, se incluye una especificación o aclaración en medio de la locución.

- **valer algo un perú Escr. t. con may. inicial** - Tras la expresión, se incluye información acerca de una posible variación de la misma.
- **dares y tomares De dar y tomar.** - Se especifica la procedencia de la locución en cuestión.

No obstante, esta información adicional, no deseada, puede extraerse fácilmente, por medio de expresiones regulares, como se puede observar en el fragmento 5.7, que muestra el código de la clase `DleTxtPhraseCleaner`. El método `CleanPhrase()` *limpia* la expresión o locución, devolviendo una versión de la misma que no incluye las especificaciones o secciones adicionales, pues no son relevantes para la detección.

Fragmento de código 5.7: Clase `DleTxtPhraseCleaner`, que retira la información adicional, no relevante para su detección, de las locuciones y expresiones.

```
1  public class DleTxtPhraseCleaner : IPhraseCleaner
2  {
3      private static readonly Regex ExtraSectionsRegex = new(
4          @"(?: U\.| V\.| Por alus\.| La var\.| En acep\.| Del| Expr\.| Escr
               \.| De| Falsa| Quizá) .+\.",
5          RegexOptions.Compiled);
6
7      private static readonly Regex SpecificationRegex = new(
8          @",\sespecialmente.*?,|,\sespecialmente.*?$",
9          RegexOptions.Compiled);
10
11     public string CleanPhrase(string phrase)
12     {
13         phrase = RemoveSpecification(phrase);
14         return RemoveExtraSections(phrase);
15     }
16
17     private static string RemoveSpecification(string phrase)
18     {
19         var matches = SpecificationRegex.Matches(phrase);
20
21         foreach (var match in matches.AsEnumerable())
22         {
23             phrase = phrase.Replace(match.Value, "");
24         }
25
26         return phrase;
27     }
28
29     private static string RemoveExtraSections(string phrase)
30     {
31         var match = ExtraSectionsRegex.Match(phrase);
32         return match.Success ? phrase.Remove(match.Index) : phrase;
33     }
34 }
```

5.4.3.2. Expresiones y locuciones con dos variantes

En torno a 1400 expresiones y locuciones, del total de casi 13500, tienen una estructura similar a las siguientes:

- **de cuenta, o de cuenta y riesgo, de alguien**
- **echar aceite al fuego, o en el fuego**

Como se puede apreciar, el diccionario lista, en las entradas correspondientes a estas locuciones, dos maneras diferentes de utilizarlas. El objetivo en este caso es separar dichas variantes, para poder buscar por separado ambos patrones. Continuando con los ejemplos anteriores, las locuciones se separarían de la siguiente forma:

- **de cuenta de alguien** y **de cuenta y riesgo de alguien**
- **echar aceite al fuego** y **echar aceite en el fuego**

Esto presenta una complejidad considerablemente mayor que en el caso descrito en la subsección 5.4.3.1, pues no solo es necesario eliminar los caracteres sobrantes (`", o"` y `","`), sino que además, se debe determinar cómo separar ambas variantes en cada caso, y qué palabras se deben sustituir. Mientras que en el primer ejemplo la segunda variante reemplaza por completo a la primera (**de cuenta, o de cuenta y riesgo, de alguien**), en el segundo, se deben reemplazar las palabras *al fuego* por *en el fuego*, manteniendo el fragmento *echar aceite* (**echar aceite al fuego, o en el fuego**).

Dado que el diccionario, como se puede ver en estos dos ejemplos, no es consistente y no sigue una disposición uniforme a la hora de especificar las dos distintas variantes, no ha sido posible resolver todos y cada uno de los casos, pero sí la gran mayoría de ellos. Se ha logrado por medio de la creación de una nueva clase, `TwoVariantPhraseSplitter` (que se podría traducir al español como *separador de locuciones de dos variantes*). Esta clase implementa, a su vez, una nueva interfaz, denominada `IPhraseSplitter` (*separador de locuciones*), con un único método, `SplitPhrase()`, que toma como parámetro la cadena de caracteres correspondiente a una expresión o locución y devuelve un arreglo con sus posibles variantes. Esta interfaz se utilizará también más adelante, para resolver más casos en los que surge la necesidad de separar las expresiones y locuciones, pues cuentan con distintas variaciones.

5.4.3.3. Expresiones y locuciones con más de dos variantes

Existen también locuciones y expresiones que contienen más de dos variantes en su nombre, como las siguientes:

- **pan y circo, o pan y fútbol, o pan y toros**
- **al, o con, o con el, objeto de**
- **allá se las haya, o se las hayan, o se lo haya, o se lo hayan, o te las hayas, o te lo hayas**

Nuevamente, para resolver estos casos se ha creado una clase que implementa la interfaz `PhraseSplitter`, denominada `MultipleVariantPhraseSplitter`, que utiliza a su vez la clase anterior, `TwoVariantPhraseSplitter`, de forma iterativa, para extraer las diferentes variaciones. De esta manera, se obtienen los siguientes arreglos (o *arrays*), respectivamente:

- `["pan y circo", "pan y fútbol", "pan y toros"]`
- `["al objeto de", "con objeto de", "con el objeto de"]`
- `["allá se las haya", "allá se las hayan", "allá se lo haya", "allá se lo hayan", "allá te las hayas", "allá te los hayas"]`

Tal y como ocurre con las locuciones con dos variantes, el Diccionario de la lengua española Española no utiliza un mismo formato en todas sus entradas para especificar las múltiples variantes que puede tener cada una de las expresiones y locuciones, por lo que no ha sido posible resolver todas ellas de manera automática, aunque sí la gran mayoría de las mismas.

5.4.3.4. Ciertas expresiones y locuciones con tres variantes

Algunas locuciones o expresiones con tres variantes diferentes no se resuelven de manera efectiva utilizando la clase `MultipleVariantPhraseSplitter`, cuyo propósito se describió en la anterior subsección. Por ello, se ha implementado también una clase `ThreeVariantPhraseSplitter`, que resuelve la mayor parte de estos casos. Se listan, a continuación, algunos ejemplos de estas locuciones y expresiones:

- **a buen seguro, al seguro, o de seguro**
- **a peso de dinero, de oro, o de plata**
- **menear, sacudir, o zurrar a alguien el bálago**

Al separarlas, se obtienen los siguientes resultados:

- `["a buen seguro", "al seguro", "de seguro"]`
- `["a peso de dinero", "a peso de oro", "a peso de plata"]`
- `["menear a alguien el bálago", "sacudir a alguien el bálago", "zurrar a alguien el bálago"]`

5.4.3.5. Expresiones y locuciones que incluyen *Tb.*

Otra manera en que se encuentran definidas diferentes variantes de una locución o expresión en el Diccionario de la lengua española, además del uso de `", o"`, es mediante una nota tras la expresión o locución que comienza con `"Tb."`. Se muestran, a continuación, algunos ejemplos:

- **a maltraer Tb. a mal traer.**
- **a rajatabla Tb. a raja tabla, p. us.**

- **a espetaperro o espetaperros Tb. a espetaperros; a espeta perros, desus.**

Como se puede observar, en ocasiones esta nota también contiene información adicional, como en el caso del último ejemplo, en que se indica que la expresión está en desuso (*desus.*).

Estas variantes son mucho más simples de distinguir, pues tan solo es necesario utilizar una sencilla expresión regular y descartar la información adicional. Es por ello que las expresiones y locuciones de este tipo sí se han resuelto de manera automática, por medio de un nuevo servicio, `TbPhraseSplitter`.

5.4.3.6. Expresiones y locuciones que incluyen *etc.*

Existe, además de la anterior, un segundo formato que el Diccionario de la lengua española utiliza para identificar múltiples variantes en una expresión o locución, el uso de comas y *etc.* Véanse los siguientes ejemplos:

- **por mi, tu, su, etc., cuenta**
- **tener medido a palmos un terreno, un lugar, etc.**

El uso de *etcétera* es la manera en que el diccionario indica que la locución o expresión tiene más variantes de las que se encuentran listadas explícitamente, por lo que aún separando adecuadamente estas últimas, no se están teniendo en cuenta todas las posibilidades. Nuevamente, en este punto sería necesaria la revisión manual de estas expresiones y locuciones. No obstante, para poder obtener las variantes que sí se encuentran especificadas, se ha hecho uso de una nueva clase que denominada `EtcPhraseSplitter`, en consonancia con las utilizadas en las anteriores subsecciones.

5.4.3.7. Expresiones y locuciones con género

Por último, en el Diccionario de la lengua española es posible encontrar algunas locuciones y expresiones cuyo género se encuentra desdoblado, como en los siguientes casos:

- **alguno, na que otro, tra**
- **mal dispuesto, ta**

En estos casos, el diccionario sí utiliza un criterio uniforme, incluyendo siempre una coma, un espacio y la última sílaba de la palabra cuyo género se desdobla, por lo que ha sido posible resolver todas las expresiones y locuciones de este tipo, obteniendo los siguientes resultados, respectivamente:

- `["alguno que otro", "alguna que otra"]`
- `["mal dispuesto", "mal dispuesta"]`

5.4.4. Generación de patrones

Tras aplicar iterativamente todas las etapas de procesamiento de las expresiones y locuciones, previamente extraídas del fichero, se obtienen uno o más *patrones* por cada locución o expresión, los cuales serán utilizados para la detección de la misma. Estos se corresponden con el modelo `PhrasePattern`, que se muestra a continuación:

Fragmento de código 5.8: Clase `PhrasePattern`, correspondiente al modelo de patrón de detección para una expresión o locución.

```
1  [Table("Patrones")]
2  public record PhrasePattern
3  {
4      [Column("ID_Patron")]
5      public int PhrasePatternId { get; set; }
6
7      [Column("Locucion_o_expresion")]
8      [MaxLength(255)]
9      public required string Phrase { get; set; }
10
11     [Column("Variante")]
12     [MaxLength(255)]
13     public required string Variant { get; set; }
14
15     [Column("Patron")]
16     [MaxLength(255)]
17     public required string Pattern { get; set; }
18
19     [Column("Palabra_base")]
20     [MaxLength(255)]
21     public required string BaseWord { get; set; }
22
23     [Column("ID_Locucion")]
24     public int PhraseId { get; set; }
25 }
```

Como se puede observar, el modelo incluye tanto el nombre de la locución (una vez que se ha eliminado la información no relevante, como se mostró en la subsección 5.4.3.1), por medio de su propiedad `Phrase`, como la variante (`Variant`) y el patrón de detección de la misma (`Pattern`). Si bien esto crea un cierto grado de redundancia en la base de datos, se ha optado por este diseño debido a que facilita la inspección y modificación manual de los patrones de detección de las expresiones y locuciones.

Nótese, por otro lado, que el modelo `PhrasePattern` no ha sido declarado con la palabra clave `class`, sino con `record`. Las clases `record` o *registros* son una característica muy interesante de C#, pues proporcionan diversas utilidades y métodos predefinidos al declararlas [21]. Por ejemplo, es posible crear una copia de una instancia de registro existente, con ciertos campos y propiedades modificados, por medio de la expresión `with`, como se muestra a continuación:

Fragmento de código 5.9: Ejemplo de uso de la sentencia `with`, que crea una copia de una instancia de registro existente, con las propiedades especificadas modificadas.

```
var expectedPattern = new PhrasePattern
{
    Phrase = "estar, o ir, aviado, da",
    Variant = "estar aviado",
    Pattern = "estar aviado",
    BaseWord = "aviado, da",
    PhraseId = phrase.PhraseId
};
HashSet<PhrasePattern> expectedResults =
[
    expectedPattern,
    expectedPattern with
    {
        Variant = "estar aviada",
        Pattern = "estar aviada"
    },
    expectedPattern with
    {
        Variant = "ir aviado",
        Pattern = "ir aviado"
    },
    // ...
];

var results = _patternGenerator.GeneratePatterns(phrase).ToHashSet();

Assert.Equal(expectedResults, results);
```

En este ejemplo, extraído de los tests unitarios correspondientes a la clase generadora de patrones, se hace uso de la expresión `with` sobre una instancia del modelo `PhrasePattern`, con el fin de facilitar y reducir el código necesario para definir los resultados esperados del test. En concreto, al declarar el `HashSet expectedResults`, se crean varias copias de la instancia `expectedPatern`, con todas sus propiedades idénticas excepto `Variant` y `Pattern`.

5.4.5. Pruebas unitarias

La capa de dominio cuenta con una amplia suite de pruebas unitarias, desarrolladas con xUnit y localizadas en el proyecto `PhraseFinder.Domain.Tests`. Los tests en cuestión verifican el correcto funcionamiento de todos los servicios de la capa de dominio, cuyas principales funciones son la lectura de las expresiones y locuciones de fichero y el procesamiento de las mismas, previos a su inserción en la base de datos. Los tests se encuentran organizados en clases, correspondiendo cada una de estas clases a un servicio diferente de la capa de dominio. Por ejemplo, se muestra en el fragmento 5.10 la clase `TbPhraseSplitterTests`, que contiene las pruebas del servicio `TbPhraseSplitter`, cuyo propósito se describió en la subsección 5.4.3.5.

Fragmento de código 5.10: Pruebas unitarias del servicio `TbPhraseSplitter`, perteneciente a la capa de dominio.

```
public class TbPhraseSplitterTests
```

```
2  {
3      private readonly TbPhraseSplitter _splitter = new();
4
5      [Fact]
6      public void SplitPhrase_WhenPhraseDoesNotContainTb_ReturnsPhrase()
7      {
8          var phrase = "de cuenta, o de cuenta y riesgo, de alguien";
9
10         var result = _splitter.SplitPhrase(phrase);
11
12         Assert.Single(result);
13         Assert.Equal(phrase, result[0]);
14     }
15
16     [Theory]
17     [InlineData(
18         "hacer fuerarropa Tb. hacer fuera ropa.",
19         new[] { "hacer fuerarropa", "hacer fuera ropa" })]
20     [InlineData(
21         "a rajatabla Tb. a raja tabla, p. us.",
22         new[] { "a rajatabla", "a raja tabla" })]
23     // ...
24     public void SplitPhrase_WhenPhraseContainsTb_ReturnsAllPossiblePhrases
           (string phrase, string[] expected)
25     {
26         var result = _splitter.SplitPhrase(phrase);
27
28         Assert.Equal(expected, result);
29     }
30 }
```

Como se puede observar, el segundo test tiene anotados los atributos `[Theory]` e `[InlineData]`, lo que significa que el test se encuentra *parametrizado* y se ejecutará una vez con cada una de las parejas de datos suministradas en los atributos `[InlineData]`. Dichos conjuntos de datos se asignan automáticamente a los parámetros del método, `phrase` y `expected`, durante la ejecución del test. Esto permite verificar que el servicio en cuestión funciona adecuadamente para un amplio rango de entradas, de forma sencilla y sin necesidad de duplicar código.

5.5. Desarrollo de la capa de datos

La capa de datos incluirá los servicios necesarios para interactuar con la base de datos, lo cual es absolutamente esencial para el desarrollo de la aplicación de escritorio. Esta hará uso de la capa de dominio, para lo cual será necesario crear una referencia al proyecto correspondiente. Las referencias, que pueden ser configuradas desde la interfaz de Visual Studio, son una de las principales ventajas de organizar distintos proyectos, relacionados entre sí, en una misma solución, pues permiten, en este caso, utilizar las clases y modelos de la capa de dominio desde la capa de datos.

Asimismo, el sistema de referencias se utiliza también para añadir librerías externas al proyecto. En el caso de la capa de datos, se agregarán distintos paquetes relacionados con Entity Framework Core, una herramienta de acceso a datos, que actúa, entre otras funciones, como ORM (Object-Relational Mapper), transformando los registros de la base de datos en objetos C# y viceversa. Además, EF Core crea y ejecuta automáticamente las sentencias SQL que consultan o escriben en la base de datos, a partir de código C#, lo cual agiliza y facilita notablemente el desarrollo.

Cabe mencionar que, al igual que con la capa de dominio, para la capa de datos se ha creado un proyecto xUnit de pruebas unitarias, denominado `PhraseFinder.Data.Tests`. Los tests en cuestión verifican el correcto funcionamiento de los servicios que interactúan con la base de datos, pues se trata de un aspecto fundamental del sistema.

5.5.1. Estructura de la base de datos

Una de las características más convenientes de Entity Framework Core es la posibilidad de inferir y generar automáticamente la estructura de la base de datos, a partir de los modelos presentes en la aplicación y las relaciones que existen entre ellos. No obstante, antes de llevar a cabo este proceso, es necesario crear una clase que herede de `DbContext`. Esta representará una sesión o conexión con la base de datos de la aplicación y se denominará `PhraseFinderDbContext`. Se muestra, a continuación, el código correspondiente:

Fragmento de código 5.11: Implementación de la clase `PhraseFinderDbContext`, que representa una sesión con la base de datos de la aplicación.

```
1  public class PhraseFinderDbContext : DbContext
2  {
3      public DbSet<Phrase> Phrases { get; set; }
4      public DbSet<PhraseDictionary> PhraseDictionaries { get; set; }
5
6      public PhraseFinderDbContext(DbContextOptions<PhraseFinderDbContext>
           options) : base(options) { }
7
8      protected override void OnModelCreating(ModelBuilder modelBuilder)
9      {
10         modelBuilder.Entity<PhraseDictionary>()
11             .Property(pd => pd.AddedAt)
12             .HasDefaultValueSql("Now()");
```

```
13
14          modelBuilder.Entity<PhraseDictionary>()
15              .Property(pd => pd.Format)
16              .HasConversion<string>();
17      }
18  }
```

Como se puede observar, esta clase permite acceder directamente tanto a las locuciones y expresiones (propiedad `Phrases`), como a los diccionarios (propiedad `PhraseDictionaries`), almacenados en la base de datos. Además, en el método `OnModelCreating()` se realiza una simple configuración adicional, que añade la fecha actual a la columna correspondiente a la fecha de creación, cada vez que se inserta un diccionario, además de añadirse una conversión a la columna de formato del diccionario, para que sus valores sean almacenados como cadenas de caracteres, en lugar de números enteros. El resto de las características de la base de datos es deducido automáticamente por EF Core. Para ilustrar el funcionamiento de esta herramienta de manera sencilla, cabe examinar las clases `PhraseDefinition` y `PhraseExample`, que representan una definición y un ejemplo de uso de una locución o expresión, respectivamente.

Fragmento de código 5.12: Clases `PhraseDefinition` y `PhraseExample`, modelos de la capa de dominio.

```
1  // PhraseDefinition.cs
2  [Table("Definiciones")]
3  public class PhraseDefinition
4  {
5      [Column("ID_Definicion")]
6      public int PhraseDefinitionId { get; set; }
7
8      [Column("Definicion")]
9      [MaxLength(1000)]
10     public required string Definition { get; set; }
11
12     [Column("ID_Locucion")]
13     public int PhraseId { get; set; }
14
15     public ICollection<PhraseExample> Examples { get; set; } = [];
16 }
17
18 // PhraseExample.cs
19 [Table("Ejemplos")]
20 public class PhraseExample
21 {
22     [Column("ID_Ejemplo")]
23     public int PhraseExampleId { get; set; }
24
25     [Column("Ejemplo")]
26     [MaxLength(1000)]
27     public required string Example { get; set; }
28
29     [Column("ID_Definicion")]
30     public int PhraseDefinitionId { get; set; }
31 }
```

Utilizando la información aportada tanto por las propiedades de la clase `PhraseExample`, como en los atributos `[Column]` y `[Table]` presentes en la misma, EF Core creará una tabla denominada `Ejemplos`, con tres columnas. En primer lugar, una columna `ID_Ejemplo` de tipo entero, que además será una clave primaria autoincrementada, pues por convención la propiedad `PhraseExampleId`, por llamarse igual que la clase a la que pertenece, más el sufijo `Id`, se interpreta de dicha manera. Seguidamente, una columna denominada `Ejemplo`, de tipo `Long Text` (tipo de datos de Access que permite almacenar textos de longitud variable, hasta un máximo de en torno a 1GB [22]).

Por último, una tercera columna `ID_Definicion`, de tipo entero al igual que la primera, que será, además, una clave foránea que hará referencia a un registro en la tabla de `Definiciones`. Esta última habrá sido creada a partir de la clase `PhraseDefinition`, siguiendo la misma lógica que con la clase `PhraseExample` y su correspondiente tabla `Ejemplos`. Por supuesto, la clave foránea modela la relación entre ambos conceptos, pues cada definición puede contener cero o más ejemplos, mientras que cada ejemplo se corresponde con una sola definición.

No obstante, cabría preguntarse cómo es posible acceder a las definiciones y ejemplos, puesto que la clase `DbContext`, mostrada en el fragmento 5.11, no posee propiedades que lo permitan. Lo que ocurre es que estos modelos no necesitan ser accedidos directamente, sino a través de la expresión y locución a la que pertenecen, en el caso de las definiciones, y a través de la definición a la que corresponden, en el caso de los ejemplos. Es por ello que la clase `Phrase`, que modela las expresiones y locuciones y se mostró en el fragmento 5.6, contiene una propiedad, de tipo colección, denominada `Definitions`. Al acceder a dicha propiedad, EF Core realizará una consulta a la base de datos y, de manera automática, incluirá en la colección todas las definiciones de la locución o expresión en cuestión. Lo mismo sucede con la clase `PhraseDefinition` y su propiedad `Examples` (línea 15 del fragmento de código 5.12).

5.5.2. Creación de la base de datos y migraciones

Para realizar el proceso de creación de la base de datos, en el cual Entity Framework Core determinará la configuración a partir de la clase `DbContext` y los modelos incluidos en esta, se ha de generar una migración, por medio de la siguiente orden:

```
dotnet ef migrations add [Nombre_migración]
```

Tras ello, será necesario ejecutarla, utilizando el siguiente comando:

```
dotnet ef database update
```

Una migración no es más que una clase C#, que contiene una serie de instrucciones que le indican a EF Core cómo crear o modificar la base de datos, independientemente del motor de base de datos utilizado. Como ejemplo de ello, se muestra a continuación un fragmento de la migración inicial, en concreto del código encargado de la creación de la tabla de diccionarios de expresiones y locuciones, correspondiente al modelo `PhraseDictionary`, junto con su respectiva clave primaria,.

Fragmento de código 5.13: Fragmentos del código de la migración inicial, el cual se encuentra en el fichero `PhraseFinder.Data\Migrations\20240331212926_InitialSchema.cs`.

```
public partial class InitialSchema : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Diccionarios",
            columns: table => new
            {
                ID_Diccionario = table
                    .Column<int>(type: "integer", nullable: false)
                    .Annotation("Jet:Identity", "1, 1"),
                Nombre = table.Column<string>(
                    type: "varchar(255)",
                    maxLength: 255,
                    nullable: false),
                // ...
            },
            constraints: table =>
            {
                table.PrimaryKey(
                    "PK_Diccionarios",
                    x => x.ID_Diccionario);
            });
        // ...
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        // ...
        migrationBuilder.DropTable(name: "Diccionarios");
    }
}
```

Cada vez que se produzca una modificación en las clases del modelo, que deba verse reflejada en la estructura de la base de datos, se podrá generar y ejecutar una nueva migración, que la actualizará automáticamente. Las migraciones se encuentran en el directorio `Migrations` del proyecto de librería de clases, correspondiente a la capa de datos. Este directorio sirve también como un histórico de los cambios realizados en la base de datos, ofreciendo la posibilidad de recrearla en cualquier momento, de forma rápida y sin errores.

Asimismo, las migraciones son reversibles, lo cual permite devolver la base de datos a un estado previo en caso de equivocación. Al revertir una migración, se ejecuta su respectivo método `Down()`, que deshace automáticamente todas las operaciones realizadas por la misma, como se puede apreciar también en el fragmento 5.13.

5.5.3. Servicios de la capa de datos

La capa de datos contiene los servicios encargados de la interacción con la base de datos, junto con las interfaces que estos implementan. En concreto, se han desarrollado dos servicios diferentes: `PhraseDictionaryService`, que incluye los métodos necesarios para consultar, añadir, eliminar y modificar diccionarios, y `PhraseService`, que contiene un único método para consultar las expresiones y locuciones de un diccionario, con distintas opciones para el filtrado, búsqueda y ordenamiento de las mismas. A continuación, se muestra el código de este último:

Fragmento de código 5.14: Clase `PhraseService`, cuyo único método consulta de la base de datos las locuciones y expresiones del diccionario suministrado como argumento.

```
1  public class PhraseService(PhraseFinderDbContext dbContext) :
       IPhraseService
2  {
3      public IQueryable<Phrase> GetPhrases(
4          PhraseDictionary phraseDictionary,
5          PhraseQueryOptions options)
6      {
7          var phrases = dbContext.Phrases
8              .AsNoTracking()
9              .Where(p =>
10                 p.PhraseDictionaryId == phraseDictionary.
                       PhraseDictionaryId)
11             .SearchPhrasesBy(options.SearchText)
12             .OrderPhrasesBy(options.OrderByOption);
13         options.UpdatePagination(phrases);
14         return phrases.Paginate(options.Page, options.PageSize);
15     }
16 }
```

Como se puede observar, esta clase hace uso de una instancia de `PhraseFinderDbContext`, que representa una sesión o conexión con la base de datos de expresiones y locuciones. A través de una propiedad de este objeto, denominada `Phrases`, se puede realizar una consulta a la tabla de expresiones y locuciones.

Como se aprecia en el fragmento de código, tras acceder a esta propiedad, se encadenan una serie de métodos, cuyo propósito es filtrar, buscar, ordenar y paginar las expresiones y locuciones, según las opciones especificadas en los argumentos del método `GetPhrases()`. Mientras que algunos de estos métodos los proporcionan C# y Entity Framework Core, `SearchPhrasesBy()`, `OrderPhrasesBy()` y `Paginate()` son específicos de este proyecto, y se encuentran declarados en clases independientes.

En concreto, estos tres son métodos de extensión, una característica muy útil de C# que permite agregar métodos a los tipos ya existentes, sin necesidad de utilizar herencia [23]. El método `Paginate()`, por ejemplo, extiende la interfaz `IQueryable`, que forma parte de la librería estándar de C#. Este método, junto a `GetTotalPages()`, también relacionado con la paginación, se encuentra definido en una clase estática, único lugar en que se pueden declarar métodos de extensión. Esta se denomina `PaginationExtensions` y se puede ver en

el siguiente fragmento de código:

Fragmento de código 5.15: Clase estática `PaginationExtensions`, que contiene dos métodos de extensión relacionados con la paginación de sequencias de objetos.

```
1  public static class PaginationExtensions
2  {
3      public static IQueryable<T> Paginate<T>(
4          this IQueryable<T> items,
5          int page,
6          int pageSize)
7      {
8          return items.Skip((page - 1) * pageSize).Take(pageSize);
9      }
10
11     public static int GetTotalPages<T>(
12         this IQueryable<T> items,
13         int pageSize)
14     {
15         return Math.Max(
16             1,
17             (int)Math.Ceiling((double)items.Count() / pageSize));
18     }
19 }
```

Nótese también que ambos métodos son génericos, gracias al uso de un parámetro de tipo, denominado `T`, en este caso. Esto resulta muy conveniente, dado que permite compartir la lógica de paginación, independientemente del tipo de datos que contenga la secuencia que se está procesando.

5.6. Desarrollo de la aplicación de escritorio WPF

Se describe a continuación el desarrollo de la aplicación de escritorio para Windows. Esta debe hacer uso de la capa de dominio, tanto para leer la información relativa a las expresiones y locuciones, contenida en ficheros de texto, como para realizar todas las etapas de procesamiento de las mismas, previas a su inserción en la base de datos. Por supuesto, para realizar las operaciones en la base de datos, será necesario utilizar, también, los servicios expuestos por la capa de datos de la solución. Tal y como se explicó en la sección anterior, para poder hacer uso del código declarado en otros proyectos, se deben añadir referencias a los mismos. En este caso, el proyecto WPF, es decir, la aplicación de escritorio, tendrá dos referencias, que apuntarán a los proyectos de librería de clases correspondientes a la capa de dominio y la capa de datos, respectivamente.

Además de las dos ya mencionadas, que apuntan hacia proyectos dentro de la misma solución, se han añadido referencias hacia varias librerías externas. La más notoria, por ser la más frecuentemente utilizada en la aplicación, es `MVVM Toolkit`, que facilita el uso de la arquitectura MVVM y reduce considerablemente la cantidad de código estándar o *boilerplate*. Otra librería destacable es `HandyControls`, que proporciona componentes para la interfaz de usuario (denominados *controles* en WPF), junto con otras convenientes utilidades que permiten construir la interfaz de la aplicación de manera más eficiente. Por último se han importado distintos paquetes relacionados con `Entity Framework Core`, el ORM usado en la capa de datos, además de la librería `Microsoft.Extensions.Hosting`, que facilita el uso de la inyección de dependencias.

5.6.1. Prototipo de la aplicación de escritorio

A continuación, se puede observar la funcionalidad del prototipo de la aplicación de escritorio para Windows. A pesar de que el software cumple con los requisitos definidos inicialmente, listados en la sección 4.1, tan solo incluye las características indispensables y este ha sido ampliado en la versión final, que se comentará más adelante.

5.6.1.1. Listado de diccionarios

La pantalla inicial de la aplicación contiene un listado con los diccionarios de expresiones y locuciones que se han añadido. Se muestra, para cada diccionario, su identificador único en la base de datos, su fecha de creación, el formato en que se encuentra, la ruta del fichero correspondiente y una descripción que el usuario pudo introducir al agregar el diccionario. Esto se puede observar en la figura 5.2

Destacan también los botones localizados arriba a la derecha, de los cuales dos están deshabilitados. Esto se debe a que, mientras que el primero tiene el propósito de redirigir a la pantalla de creación de diccionarios, que se comentará más adelante, los otros dos botones llevan a cabo acciones sobre el diccionario seleccionado: redirigir a la pantalla de expresiones

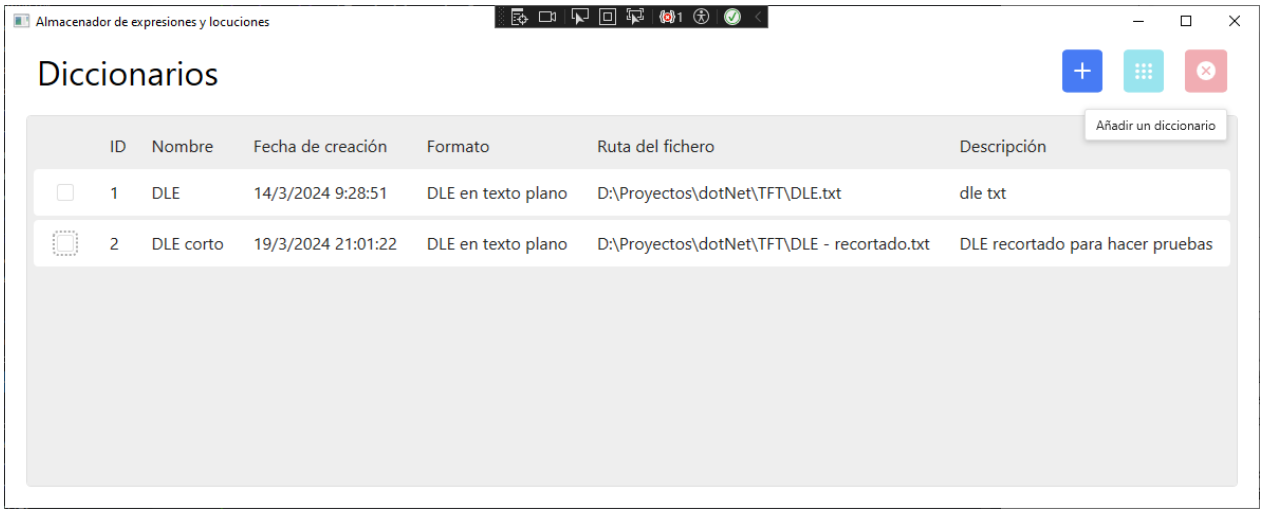

Figura 5.2: Pantalla inicial del prototipo de la aplicación de escritorio, que muestra un listado de los diccionarios de expresiones y locuciones que han sido añadidos.

de dicho diccionario, o eliminarlo, respectivamente. En este caso no hay ninguno seleccionado, por lo que dichas funciones permanecen deshabilitadas.

Al seleccionar un diccionario y pulsar el botón rojo, se desplegaría el diálogo de la figura 5.3. En el caso de confirmar su eliminación, se borrarían junto a este todas las expresiones y locuciones que contenga.

Cabe mencionar, además, que al colocar el cursor sobre cada botón, se muestra una breve descripción de su función, como se puede ver bajo el botón para añadir un diccionario en la figura 5.2.

5.6.1.2. Creación de un diccionario

Para la creación de un diccionario de expresiones y locuciones, se hace uso del botón azul en la pantalla principal, los cual redirige al usuario al formulario de la figura 5.4. Como se puede apreciar, el usuario debe introducir un nombre para el diccionario, opcionalmente una descripción y seleccionar el formato del fichero que se va a leer (en este momento solo se soporta el formato del DLE en texto plano, por lo que se selecciona por defecto). Por último, el usuario elige el fichero que contiene las expresiones y locuciones, en el formato seleccionado, de entre los archivos de su sistema. Al seleccionar dicho fichero, la ruta del mismo aparece bajo el botón, que ahora está deshabilitado. Esto permite descartar dicho fichero para poder volver a elegirlo, como se puede ver en la figura 5.5.

Una vez que la información requerida ha sido introducida, es posible confirmar la creación del diccionario, momento en que se lee el fichero seleccionado y se extraen, procesan y almacenan todas las locuciones y expresiones encontradas en el mismo, junto con sus definiciones, ejemplos y patrones de detección. Dado que este proceso puede tomar de unos segundos a varios minutos, dependiendo del tamaño del fichero, se muestra un mensaje y una barra de

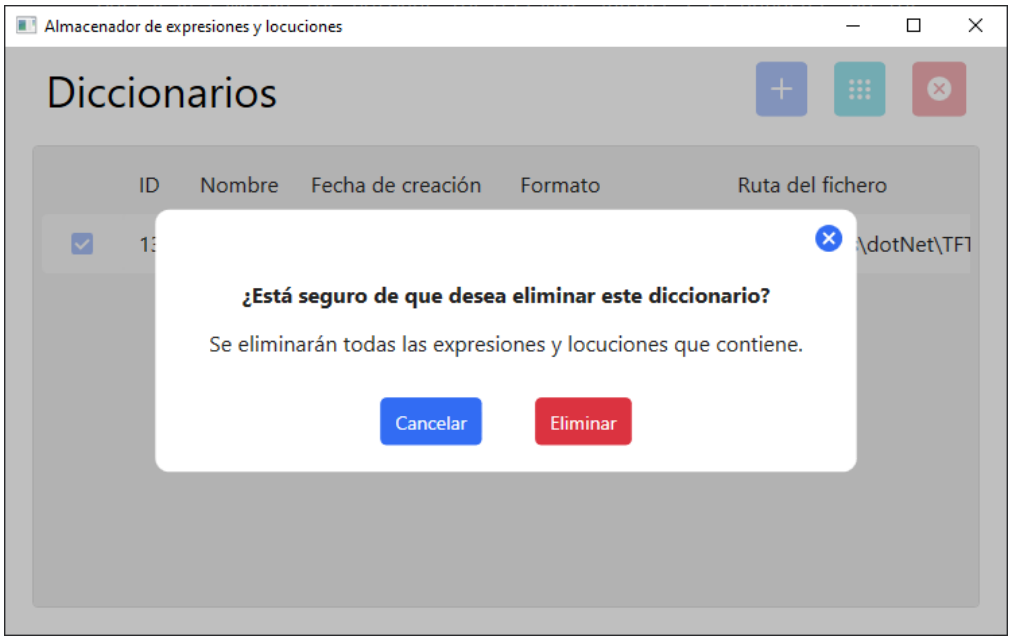

Figura 5.3: Diálogo de confirmación, previo a la eliminación de un diccionario.

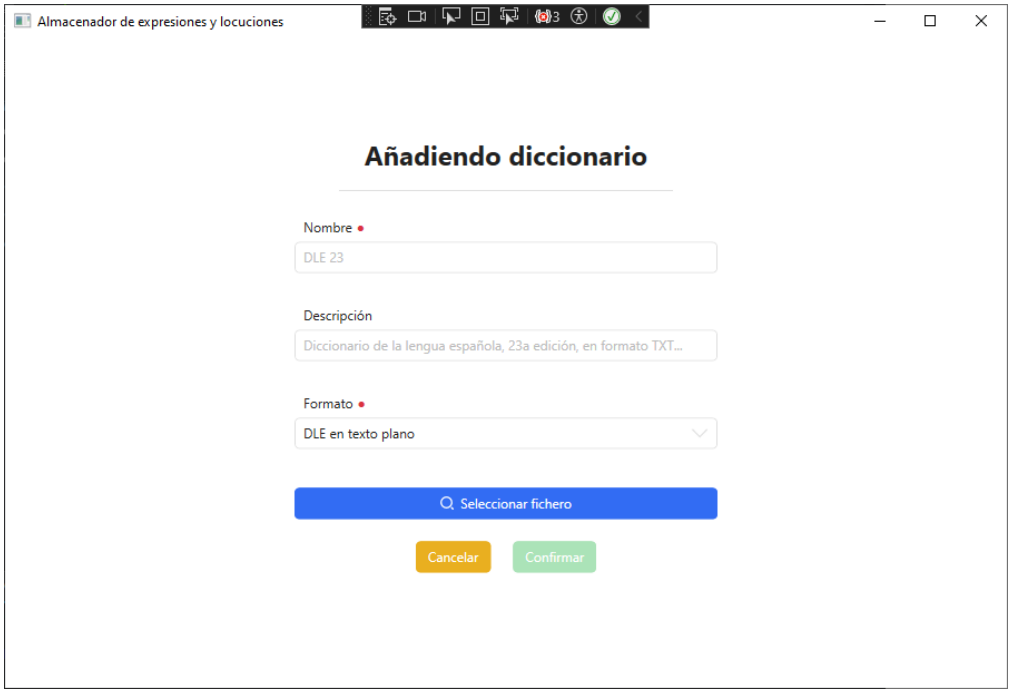

Figura 5.4: Pantalla de creación de un diccionario de expresiones y locuciones.

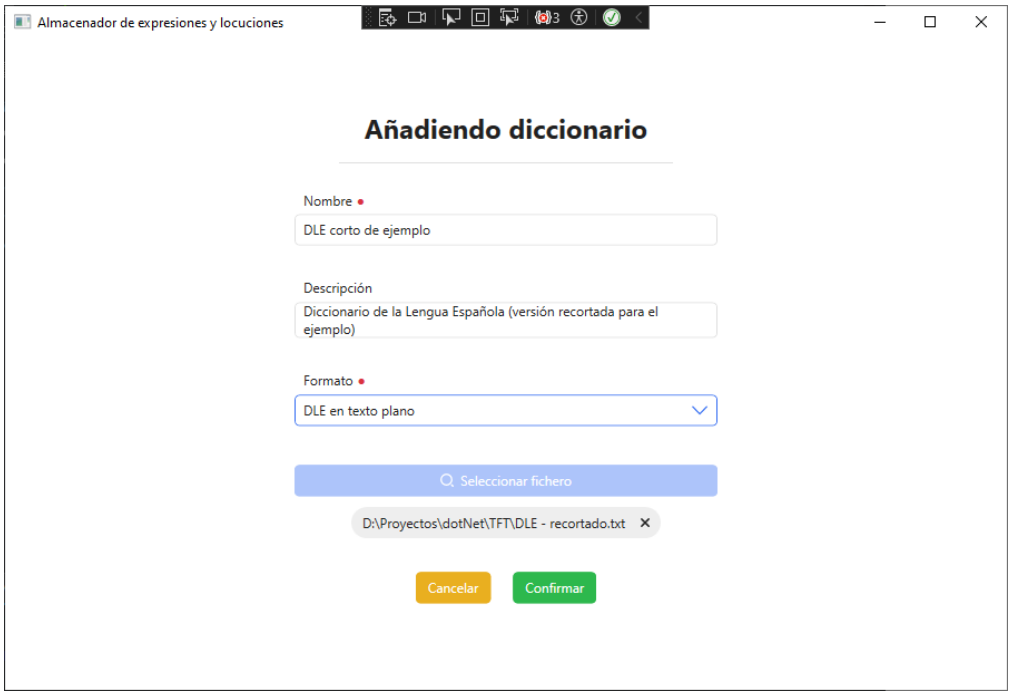

Figura 5.5: Pantalla de creación de un diccionario, con todos sus campos cumplimentados.

carga en la zona inferior de la pantalla (figura 5.6).

Una vez se ha añadido el diccionario, este pasa a formar parte del listado presente en la pantalla principal, a la cual se redirige la aplicación.

5.6.1.3. Listado de expresiones y locuciones

Para finalizar con el prototipo de la aplicación de escritorio, se muestra en la figura 5.7 la pantalla de expresiones y locuciones, donde se listan todas aquellas contenidas por el diccionario seleccionado. A esta se accede al pulsar el botón azul celeste en la pantalla principal (figura 5.2), habiendo seleccionado previamente un diccionario de la lista. Al igual que en esta última, las expresiones y locuciones se pueden ordenar, tanto en sentido ascendente como descendente, por distintas propiedades: ID, nombre, palabra base, categorías y revisado, en este caso. Además, dado que la cantidad de expresiones y locuciones es demasiado elevada como para mostrarlas simultáneamente en la misma pantalla, se ha implementado una barra de búsqueda, localizada en la parte superior, y un sistema de paginación, localizado en la zona inferior derecha. La función de paginación permite seleccionar la página actual, tanto mediante botones como introduciendo directamente el número de página deseado, además de cambiar la cantidad de expresiones y locuciones que se muestran en cada página, que puede ser 10, 20, 50 o 100.

Si bien la aplicación podría incluir funciones de filtrado y búsqueda mucho más avanzadas, o mostrar las definiciones, ejemplos y patrones de cada expresión o locución, dichas características no son imprescindibles, precisamente porque la base de datos se almacena en

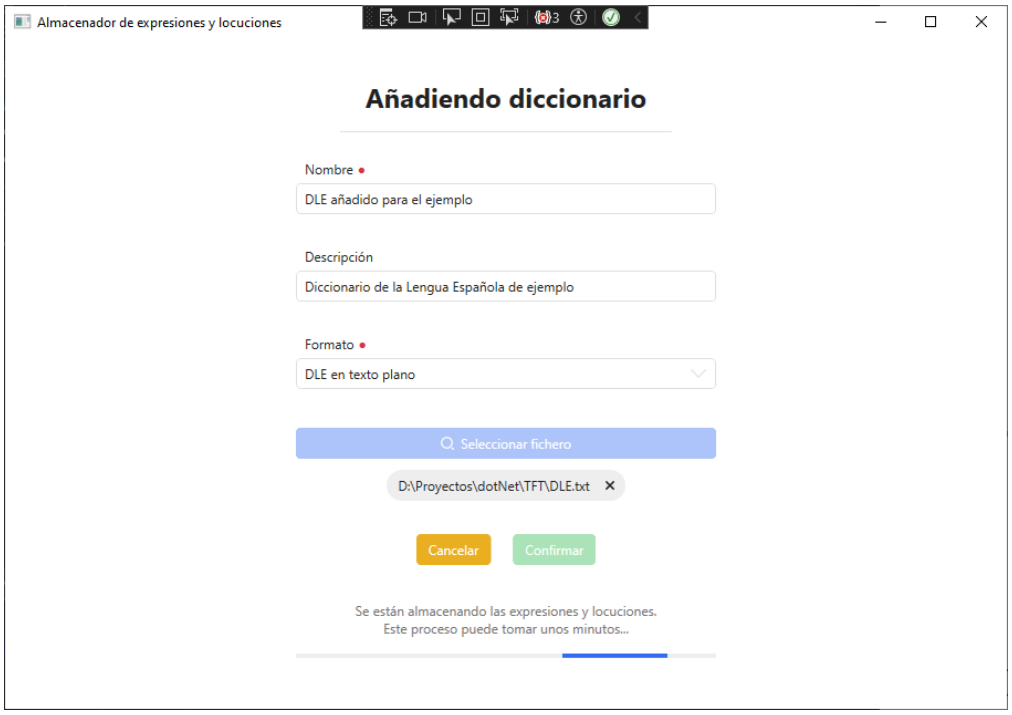

Figura 5.6: Pantalla de creación de un diccionario, en el momento en que este está siendo añadido a la base de datos.

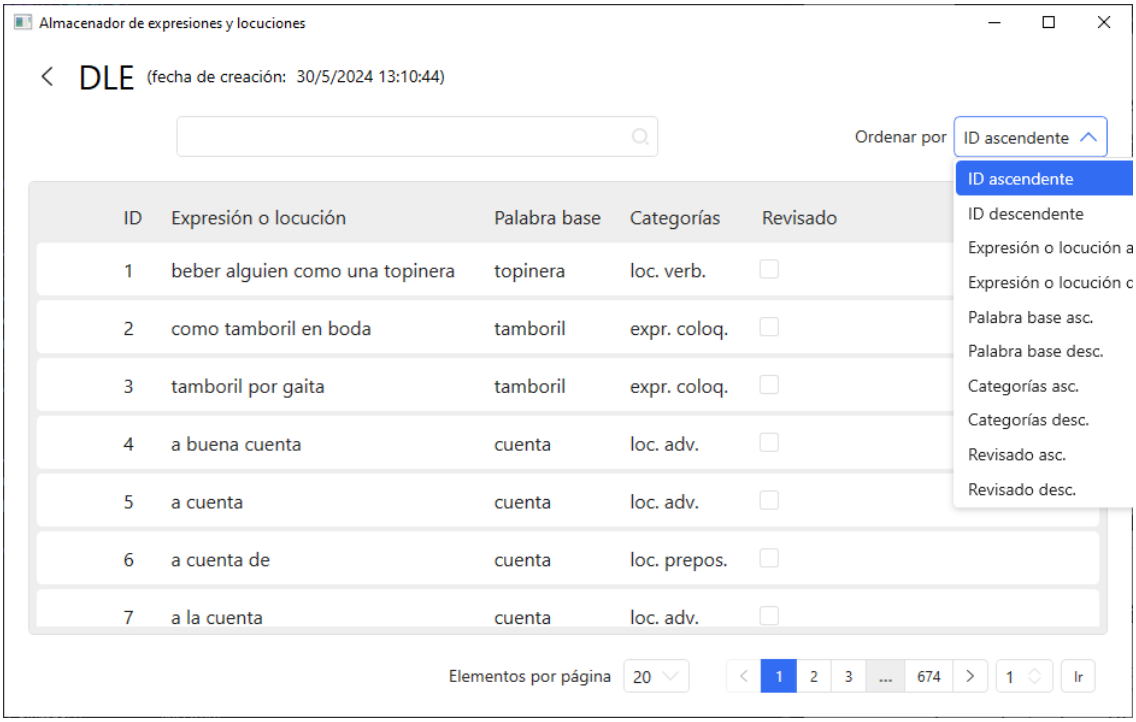

Figura 5.7: Pantalla que contiene el listado de expresiones y locuciones de un diccionario, en el primer prototipo de la aplicación de escritorio.

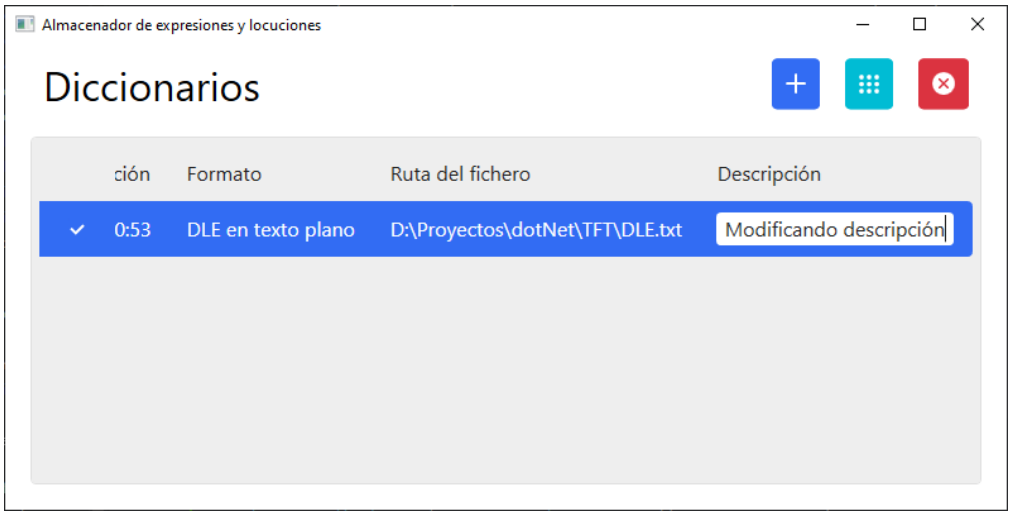

Figura 5.8: Pantalla principal de la versión final de la aplicación de escritorio, mientras se modifica la descripción de un diccionario.

un fichero Access en el sistema, pudiendo el administrador acceder a todas ellas desde la interfaz gráfica de Microsoft Access.

5.6.2. Versión final de la aplicación de escritorio

Para la versión final de la aplicación de escritorio, ha sido posible mejorar algunos aspectos visuales de la aplicación. El más notable es el aumento del tamaño de la fuente y los botones en la pantalla para añadir un diccionario. Además, se ha añadido la posibilidad de modificar los detalles de un diccionario, desde el propio listado donde estos se encuentran. Concretamente, se pueden actualizar tanto el nombre como la descripción de cualquier diccionario. Esta nueva funcionalidad puede apreciarse en la figura 5.8.

Por último, como se puede ver en la figura 5.9, se han añadido mensajes de error al listado de diccionarios y a la pantalla de creación de los mismos. Estos se muestran si se ha producido una excepción al momento de añadir o actualizar un diccionario, como en el caso de que el usuario introdujera un nombre o descripción demasiado largo para el mismo.

5.6.3. Arquitectura MVVM

Como se mencionó anteriormente, para el desarrollo de la aplicación de escritorio se ha aplicado la arquitectura MVVM (*Model-View-ViewModel*). Por tanto, la vista se encuentra completamente desacoplada de la lógica de la aplicación, y por supuesto, de las operaciones realizadas en la base de datos. Tanto la información que se muestra, como las acciones, o *comandos*, que pueden ejecutarse desde la interfaz de usuario, se encuentran definidos en la respectiva clase ViewModel y se enlazan con la vista por medio del *data binding*, el proceso que establece la conexión entre los datos y la interfaz de usuario. Como ejemplo de ello, se muestra

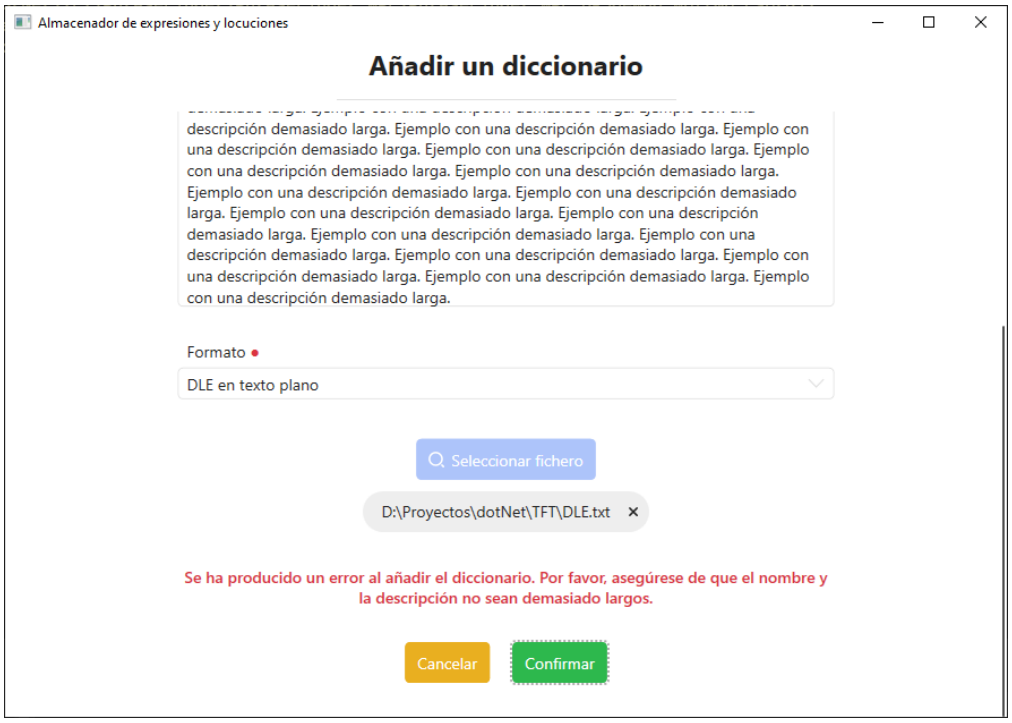

Figura 5.9: Mensaje de error al intentar añadir un diccionario de expresiones y locuciones con una descripción demasiado extensa.

en el fragmento 5.16 una fracción del código XAML de la vista correspondiente al listado de expresiones y locuciones de un diccionario, en concreto los controles y atributos relacionados con el selector de página del sistema de paginación (`<hc:Pagination>`). Seguidamente, en el fragmento 5.17, se puede observar parte de la clase ViewModel correspondiente a dicha vista.

Fragmento de código 5.16: Fragmento del código XAML correspondiente a la vista del listado de expresiones y locuciones de un diccionario.

```
1  <!-- ... -->
2  <hc:Pagination
3    Margin="30,0,30,20"
4    HorizontalAlignment="Right"
5    VerticalAlignment="Center"
6    AutoHiding="False"
7    FontSize="14"
8    IsJumpEnabled="True"
9    MaxPageCount="{Binding CurrentTotalPages, Mode=OneWay,
       UpdateSourceTrigger=PropertyChanged}"
10   PageIndex="{Binding CurrentPage, Mode=TwoWay}">
11   <hc:Interaction.Triggers>
12     <hc:EventTrigger EventName="PageUpdated">
13       <hc:EventToCommand Command="{Binding LoadPhrasesCommand}"
           PassEventArgsToCommand="False" />
14     </hc:EventTrigger>
15   </hc:Interaction.Triggers>
16 </hc:Pagination>
17 <!-- ... -->
```

Fragmento de código 5.17: Fragmentos de la clase ViewModel correspondiente a la vista del listado de locuciones y expresiones de un diccionario, denominada `PhrasesViewModel`.

```
1  internal partial class PhrasesViewModel : ObservableObject
2  {
3      public PhraseDictionary PhraseDictionary { get; }
4      private readonly IPhraseService _phraseService;
5      private readonly INavigationService _navigationService;
6
7      public ObservableCollection<Phrase> DisplayedPhrases { get; } = [];
8
9      [ObservableProperty]
10     private PhraseQueryOptions _options = new();
11
12     [ObservableProperty]
13     private int _currentPage;
14
15     [ObservableProperty]
16     private int _currentTotalPages;
17
18     public PhrasesViewModel(
19         IPhraseService phraseService,
20         INavigationService navigationService)
21     {
22         _navigationService = navigationService;
23         PhraseDictionary = WeakReferenceMessenger.Default
24             .Send<PhraseDictionaryRequestMessage>();
25         _phraseService = phraseService;
26         LoadPhrases();
27     }
28
29     [RelayCommand]
30     public void LoadPhrases()
31     {
32         Options.Page = CurrentPage;
33         Options.TotalPages = CurrentTotalPages;
34         DisplayedPhrases.Clear();
35         var phrases = _phraseService.GetPhrases(PhraseDictionary, Options)
               ;
36         DisplayedPhrases.AddRange(phrases);
37         CurrentPage = Options.Page;
38         CurrentTotalPages = Options.TotalPages;
39     }
40
41     // ...
42 }
```

Como se puede observar, algunos campos del ViewModel, como es el caso de `_currentPage`, tienen asignados el atributo `[ObservableProperty]`, proporcionado por el paquete `MVVM Toolkit`. Este atributo genera el código necesario para que dichos campos puedan ser *observados* por la vista, la cual se actualizará ante un cambio en los datos. En el caso de `_currentPage`, que representa la página actual en el sistema de paginación, se puede ver cómo, en la línea 10 del fragmento 5.16, esta propiedad se enlaza con la vista, por medio de

un *data binding* bidireccional (`{Binding CurrentPage, Mode=TwoWay}`), lo cual se traduce en que los cambios que tengan lugar en la vista, como en el caso de que el usuario seleccione una nueva página, se verán reflejados en el ViewModel y viceversa.

Además, ciertos métodos, como `LoadPhrases()`, poseen el atributo `[RelayCommand]`, que permite asignar un comando a distintas acciones que el usuario puede llevar a cabo en la vista, desencadenando la ejecución del método cuando estas se producen. Esto se puede ver en las líneas 12 y 13 del fragmento 5.16, donde el comando `LoadPhrasesCommand`, correspondiente al método `LoadPhrases()`, es asignado al evento de actualización de página (`PageUpdated`) del sistema de paginación.

De esta manera, se logra desacoplar por completo el ViewModel de la vista. A su vez, las clases ViewModel son también independientes de las operaciones realizadas en la base de datos, pues de estas se encarga la capa de datos de la aplicación, que expone los servicios necesarios para ello.

Asimismo, otro aspecto interesante del código de la clase `PhrasesViewModel` es el uso de un `Messenger` [24] para la obtención del diccionario, cuyas expresiones y locuciones se muestran en la pantalla en cuestión, tras ser consultadas de la base de datos por medio del servicio `PhraseService`. Todo ello ocurre en el constructor de la clase, donde se encuentra una llamada al método `Send()` de la instancia por defecto (`Default`) de la clase `WeakReferenceMessenger`. Este método envía un *mensaje* que solicita el diccionario de expresiones y locuciones que ha sido seleccionado. Previamente, en el constructor de la clase ViewModel correspondiente a la pantalla principal de la aplicación, que contiene el listado de diccionarios almacenados, se ha registrado un método que responde a dichos mensajes, proporcionando el diccionario actualmente seleccionado en el listado.

A través de este sistema de *mensajes*, se logra desacoplar por completo las clases ViewModel entre sí. En este caso, por ejemplo, `PhrasesViewModel` y `PhraseDictionariesViewModel` se desconocen mutuamente, y sin embargo, esta última es capaz de transmitir información a la primera.

Por último, cabe destacar el uso de algunos de los controles proporcionados por la librería `HandyControls`, mencionada al comienzo de la sección 5.6. Este es el caso, por ejemplo, del componente `<hc:Pagination>`, que se corresponde con el selector de página del sistema de paginación, incluido en el código XAML del fragmento 5.16.

5.6.4. Inyección de dependencias y navegación entre pantallas

La inyección de dependencias es, probablemente, el patrón de diseño de software más ampliamente utilizado a lo largo de este trabajo. Es, además, especialmente fácil de implementar en .NET, pues los servicios que se desean inyectar pueden ser añadidos al *host* de la aplicación, un objeto que encapsula los recursos y la funcionalidad de vigencia de una aplicación [25]. Una vez que han sido agregados, pueden solicitarse al host en cualquier momento e inyectarse en el constructor de la clase que precise de ellos, de forma muy simple.

Un ejemplo, muy ilustrativo de ello, se encuentra en la implementación de la navegación

entre pantallas en la aplicación de escritorio, que consiste en un servicio que toma como parámetro la función que inicializa las distintas clases ViewModel de la aplicación. Dicha función no es más que un método en el proveedor de servicios del host de la aplicación, que devuelve el servicio apropiado según el tipo que se le solicita, ya sea el tipo de la clase en sí, o una interfaz que esta implementa. Todos estos servicios son agregados al host al iniciar la aplicación, por medio de una serie de métodos de extensión. Como ejemplo de ello, el siguiente fragmento muestra el método que añade los servicios relacionados con la navegación entre pantallas:

Fragmento de código 5.18: Método de extension `AddNavigation()`, que agrega al *host* de la aplicación los servicios necesarios para la navegación entre pantallas.

```
1  private static void AddNavigation(this IHostApplicationBuilder builder)
2  {
3      builder.Services.AddSingleton<Func<Type, INotifyPropertyChanged>>(
4          serviceProvider =>
5      {
6          return viewModelType =>
7              (INotifyPropertyChanged)serviceProvider
8                  .GetRequiredService(viewModelType);
9      });
10     builder.Services
11         .AddSingleton<INavigationService, NavigationService>();
12 }
```

Como se puede observar, el método añade a la colección de servicios del host de la aplicación una función que instancia las clases ViewModel de la misma, además del servicio de navegación, `NavigationService`, cuyo constructor toma como argumento dicha función. De esta manera, para navegar a una nueva pantalla, basta con inyectar el servicio de navegación, por medio del constructor del ViewModel, y llamar al método `NavigateTo()`, suministrando como argumento el tipo del ViewModel a cuya vista se desea navegar. Precisamente de esta forma se lleva a cabo la navegación desde la pantalla principal de la aplicación, el listado de diccionarios, a la pantalla de locuciones y expresiones, como se puede observar en el fragmento 5.19.

Fragmento de código 5.19: Constructor y método `NavigateToPhrases()` de la clase `PhraseDictionariesViewModel`.

```
1  // ...
2
3  public PhraseDictionariesViewModel(
4      IPhraseDictionaryService phraseDictionaryService,
5      INavigationService navigationService)
6  {
7      _navigationService = navigationService;
8      // ...
9  }
10
11 public bool IsPhraseDictionarySelected() =>
12     SelectedPhraseDictionary != null;
13
14 [RelayCommand(CanExecute = nameof(IsPhraseDictionarySelected))]
```

```
15 public void NavigateToPhrases()
16 {
17     _navigationService.NavigateTo<PhrasesViewModel>();
18 }
19
20 // ...
```

En lo que respecta a la vista, para poder mostrar la pantalla correspondiente al ViewModel actual, se modifica la ventana principal de la aplicación, denominada `MainWindow`, que encapsula a todas las pantallas o vistas de la aplicación. El código de la misma se halla en el fichero `MainWindow.xaml`, que se encuentra en el directorio raíz de la aplicación y cuyo contenido se muestra en el fragmento 5.20. Inmediatamente después, en el fragmento de código 5.21, se puede observar la clase ViewModel correspondiente a la ventana principal, llamada `MainViewModel`, que contiene la navegación de la aplicación.

Fragmento de código 5.20: Código XAML correspondiente a la ventana principal de la habitación, contenido en el fichero `MainWindow.xaml`.

```
1  <Window
2    x:Class="PhraseFinder.WPF.MainWindow"
3    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
6    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
7    xmlns:viewModels="clr-namespace:PhraseFinder.WPF.ViewModels"
8    Title="Almacenador de expresiones y locuciones"
9    Width="800"
10   Height="600"
11   d:DataContext="{d:DesignInstance viewModels:MainViewModel}"
12   xml:lang="es-ES"
13   mc:Ignorable="d">
14
15   <ContentControl Content="{Binding Navigation.CurrentViewModel, Mode=
        OneWay, UpdateSourceTrigger=PropertyChanged}" />
16
17 </Window>
```

Fragmento de código 5.21: Clase ViewModel correspondiente a la ventana principal de la aplicación, denominada `MainViewModel`.

```
1  internal partial class MainViewModel : ObservableObject
2  {
3      [ObservableProperty]
4      private INavigationService _navigation;
5
6      public MainViewModel(INavigationService navigation)
7      {
8          _navigation = navigation;
9          _navigation.NavigateTo<PhraseDictionariesViewModel>();
10     }
11 }
```

Como se puede observar en la línea 11 del fragmento 5.20, el ViewModel de la vista

en cuestión se encuentra definido en el código XAML por el atributo `d:DataContext`. Este ViewModel contiene una propiedad denominada `Navigation`, que a su vez incluye una propiedad `CurrentViewModel`, que determina el ViewModel actual. En cuanto el ViewModel actual cambia, es decir, cuando se navega a otra pantalla, el elemento `ContentControl` del `MainWindow` muestra la vista correspondiente a dicho ViewModel. Las diferentes vistas de la aplicación, que se encuentran en el directorio `Views`, están definidas como componentes, o *controles*, independientes, llamados `UserControl`, cada uno con su ViewModel correspondiente.

5.7. Desarrollo de la aplicación web Razor Pages

Dado que el servicio WCF de detección de expresiones y locuciones aún no se encontraba terminado, durante el desarrollo del prototipo de la aplicación web se utilizaron datos de prueba. Para ello, se hizo uso del sistema de inyección de dependencias de .NET, tal y como se hizo también en la aplicación de escritorio. Concretamente, se declaró un servicio sustituto, denominado `PhraseFinderServiceDev`, en el punto de partida de la aplicación web, el fichero `Program.cs`, como se puede observar a continuación.

Fragmento de código 5.22: Uso del sistema de inyección de dependencias para declarar el servicio sustituto, `PhraseFinderServiceDev`.

```
1 var builder = WebApplication.CreateBuilder(args);
2
3 // ...
4
5 builder.Services
6     .AddSingleton<IPhraseFinderService, PhraseFinderServiceDev>();
7
8 // ...
```

Por medio de este sistema, en lugar de utilizar el servicio real, aún sin terminar, se inyecta la clase `PhraseFinderServiceDev`, que implementa la misma interfaz que dicho servicio pero devuelve datos de prueba, es decir, una secuencia fija de expresiones y locuciones encontradas. Esta se utiliza en la clase correspondiente al modelo de la página de expresiones y locuciones encontradas, `PhrasesModel`, siendo inyectado a través del constructor de la misma.

Fragmento de código 5.23: Declaración y constructor principal de la clase que representa el modelo de la página de expresiones y locuciones.

```
1 public class PhrasesModel(IPhraseFinderService phraseFinder) : PageModel
2 {
3     // ...
4 }
```

El modelo de página, o `PageModel`, es un elemento fundamental en Razor Pages, pues es la clase que define los datos que consumirá la vista, representada en HTML, además de las acciones que se llevan a cabo al recibir distintos tipos de peticiones HTTP (GET, POST, PUT, DELETE, ...). Dichas acciones se definen en unos métodos específicos de la clase `PageModel`, denominados *handlers*, que por convención se nombran de la siguiente manera: `OnGet()` u `OnGetAsync()` (si realiza operaciones asíncronas) para el método de petición GET, `OnPost()` u `OnPostAsync()` para el método POST, etc [10]. Un ejemplo muy ilustrativo de estas clases `PageModel` y los métodos *handler* se encuentra más adelante, en el fragmento de código 5.24, correspondiente al modelo de la página principal, denominado `IndexModel`

En este caso, la aplicación web dispone de tres páginas diferentes, la página principal, que incluye un área de texto donde el usuario puede enviar el contenido que quiere procesar, una segunda pantalla, alternativa a la primera, donde puede subir un fichero de texto y, finalmente, la página de expresiones y locuciones encontradas. También cuenta con una página de error,

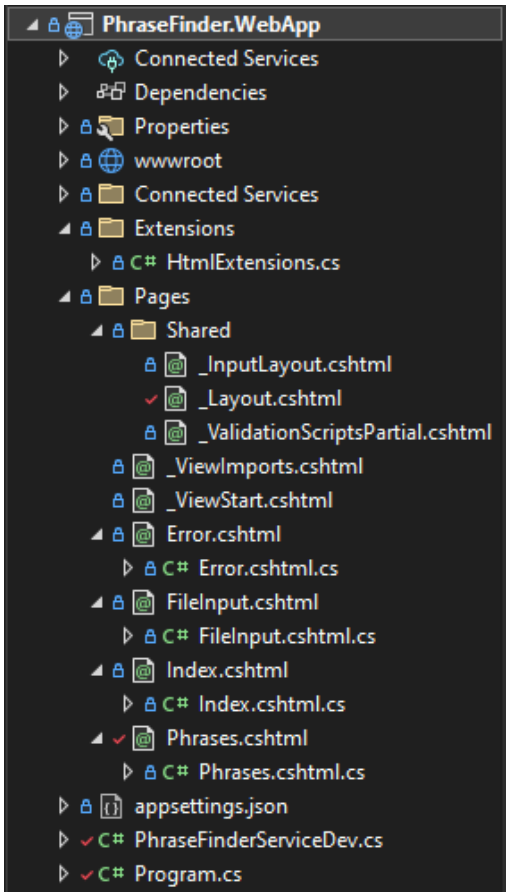

Figura 5.10: Organización de archivos del proyecto de aplicación web Razor Pages, denominado `PhraseFinder.WebApp`.

que solo se muestra en caso de que la aplicación lance una excepción. Cada una de estas páginas cuenta, a su vez, con una clase `PageModel` asociada. Estas clases se encuentran en los ficheros con extensión `.cshtml.cs`, mientras que las páginas en sí, la vista, se encuentra declarada en los ficheros `.cshtml`. Todo esto se puede ver de forma mucho más clara en la figura 5.10, que muestra la organización del proyecto de aplicación web.

5.7.1. Primer prototipo de la aplicación web

Como se podrá comprobar en las páginas siguientes, el prototipo inicial de la aplicación web cuenta con toda la funcionalidad imprescindible, especificada en los requisitos listados en la sección 4.3.

5.7.1.1. Entrada de texto o fichero de texto plano

Inicialmente, el usuario dispone de una página principal, donde puede introducir un texto de forma manual a través de un área de texto. En lugar de ello, también puede subir un

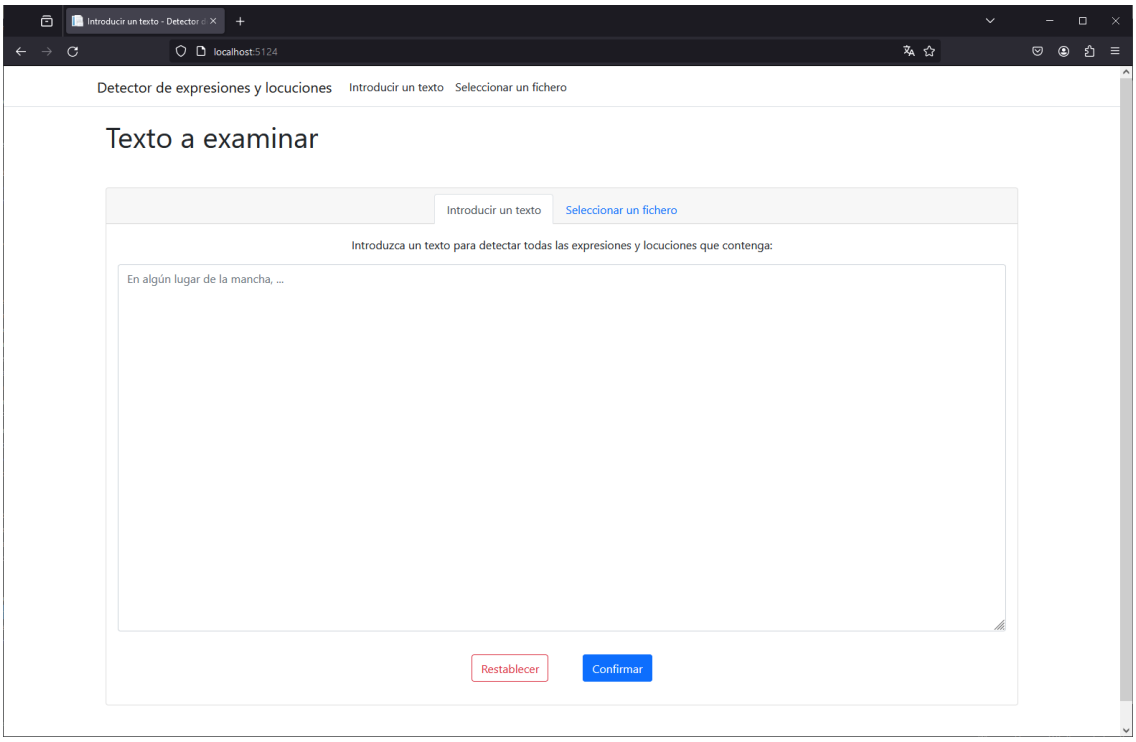

Figura 5.11: Página principal del primer prototipo de la aplicación web.

fichero de texto plano, por medio del `input` presente en la página de selección de fichero. Como se puede observar en la imagen 5.11, ambos formularios contienen dos botones. El botón de la izquierda, de color rojo, restablece el formulario, mientras que el azul envía la información introducida al servidor. Se puede alternar entre ambas páginas utilizando tanto la barra de navegación de la parte superior de la pantalla, como la que se encuentra justo encima del formulario.

5.7.1.2. Validación de entradas

En caso de que no se haya introducido un texto, o su longitud no se encuentre entre los límites mínimo y máximo, al tratar de enviar el formulario, se mostrarán los errores correspondientes, como se aprecia en la figura 5.13. Lo mismo ocurre si, al seleccionar un fichero, su tamaño supera el máximo, o su contenido no supera la longitud mínima.

La validación es notablemente fácil de configurar en Razor Pages. En el caso del área de texto, esta se realiza en la clase `IndexModel`, modelo de la página principal de la aplicación, que se muestra, algo simplificado, en el fragmento 5.24.

Fragmento de código 5.24: Modelo de la página principal de la aplicación web.

```
1 public class IndexModel : PageModel
2 {
3     [BindProperty]
4     [Required(ErrorMessage = "Por favor, introduzca un texto.")]
5     public string Text { get; set; } = string.Empty;
```

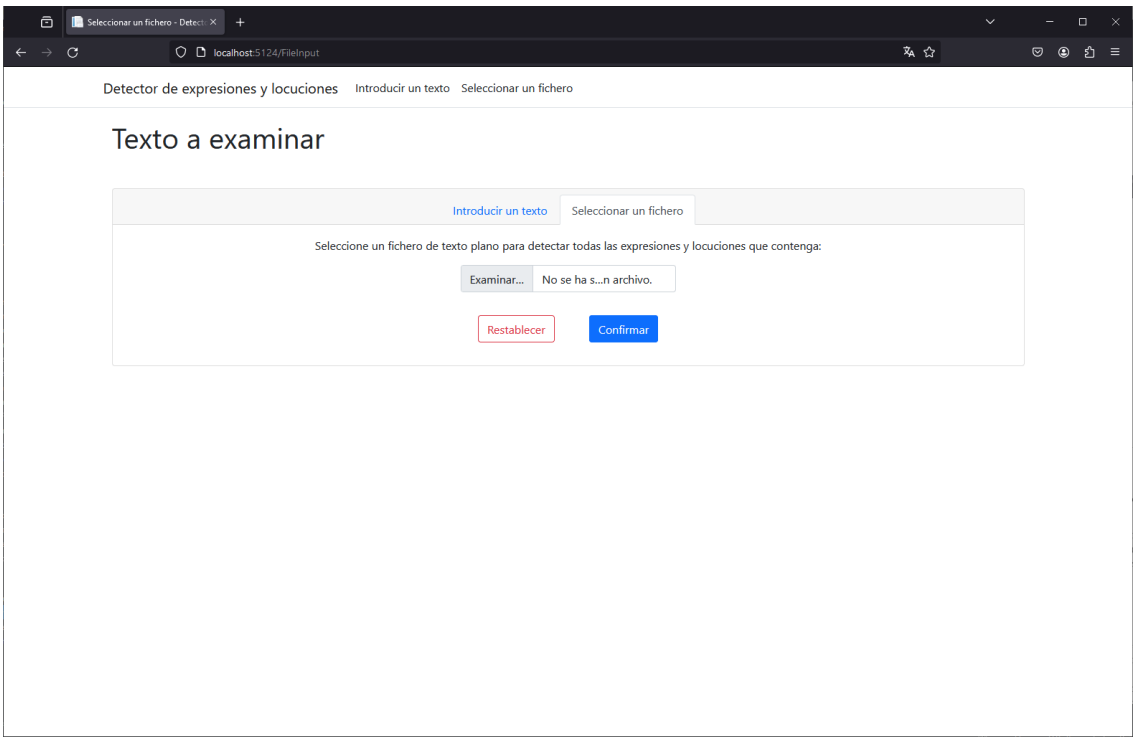

Figura 5.12: Página de selección de fichero del primer prototipo de la aplicación web.

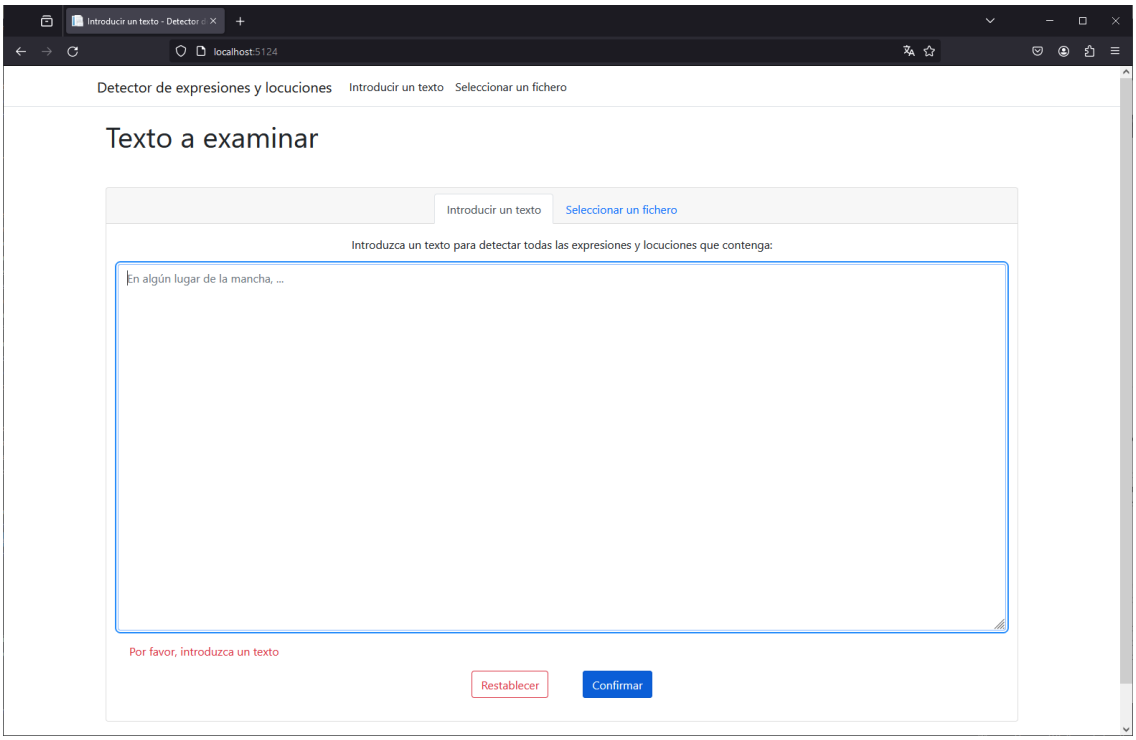

Figura 5.13: Primer prototipo de la aplicación web. Formulario con mensaje de validación, presente cuando el usuario trata de enviar un texto vacío.

```
6
7       private readonly TextValidationOptions _options;
8
9       public IndexModel(
10          IOptions<TextValidationOptions> textValidationOptions)
11      {
12          _options = textValidationOptions.Value;
13      }
14
15      public IActionResult OnPost()
16      {
17          if (!ModelState.IsValid)
18          {
19              return Page();
20          }
21
22          if (Text.Length < _options.MinLength)
23          {
24              var characters = _options.MinLength == 1 ? "carácter" : "
                    caracteres";
25              ModelState.AddModelError(
26                  nameof(Text),
27                  $"El texto es demasiado corto (mínimo {_options.MinLength}
                        {characters}).");
28              return Page();
29          }
30
31          // ...
32
33          TempData["Text"] = Text;
34          return RedirectToPage("/Phrases");
35      }
36  }
```

Se valida, tanto en el lado del cliente como en el servidor, que el texto tenga algún valor, gracias al atributo `[Required]`, donde se especifica el mensaje de validación (`ErrorMessage`). También es necesario el atributo `[BindProperty]`, que enlaza la propiedad `Text` del modelo de la página con el valor que el usuario introduce en el área de texto, para lo cual se hace uso de un atributo específico de ASP.NET en el área de texto, denominado `asp-for`, como se puede apreciar en el siguiente fragmento:

Fragmento de código 5.25: Código HTML simplificado, correspondiente al área de texto de la aplicación web.

```
1  <textarea
2    id="text"
3    asp-for="Text"
4    <!-- ... -->
5  ></textarea>
```

Además de verificarse que el usuario ha introducido un texto, en el método responsable de la acción POST, denominado `OnPost()` por convención, se comprueba que la longitud del mismo se encuentra entre unos determinados límites. En las líneas 22 a 29 del fragmento

5.24, por ejemplo, se verifica que el texto sea mayor que la longitud mínima establecida. En caso contrario, se muestra un mensaje de validación correspondiente.

Por otra parte, como se puede observar también en el fragmento 5.24, las opciones de validación del texto, que incluyen su longitud mínima y máxima, se inyectan en el constructor de la clase (línea 10 del fragmento 5.24), puesto que estos parámetros se encuentran definidos en la configuración de la aplicación, un fichero en formato JSON denominado `appsettings.json`, y que contiene lo siguiente:

Fragmento de código 5.26: Contenido del fichero `appsettings.json`, que define la configuración de la aplicación.

```
1  {
2      "Logging": {
3          "LogLevel": {
4              "Default": "Information",
5              "Microsoft.AspNetCore": "Warning"
6          }
7      },
8      "AllowedHosts": "*",
9      "Validation": {
10         "Text": {
11             "MinLength": 3,
12             "MaxLength": 10000
13         },
14         "TextFile": {
15             "MinContentLength": 3,
16             "MaxSizeKiloBytes": 10
17         }
18     }
19 }
```

Se especifica también la validación del fichero de texto, cuyo tamaño debe ser inferior a 10 kilobytes y cuyo contenido ha de tener una longitud superior a 3 caracteres. Esto se verifica en el método `OnPostAsync()`, encargado de la acción POST. Este *handler* se encuentra, por supuesto, en la clase correspondiente al modelo de la página de selección de fichero, denominada `FileInputModel`. Su estructura es muy similar a la de la clase `IndexModel`. Nuevamente, las opciones de validación han sido inyectadas en el constructor de la misma y se ha hecho uso del atributo `[Required]` para validar, tanto en el lado del cliente como en el servidor, la presencia del fichero. Se muestran a continuación, en el fragmento 5.27, la propiedad correspondiente al fichero de texto, el constructor de la clase `FileInputModel` y parte del método `OnPostAsync()`, en concreto el código que valida el tamaño del fichero.

Fragmento de código 5.27: Constructor y parte del método `OnPostAsync()` de la clase correspondiente al modelo de la página de selección de fichero de la aplicación web.

```
1  [BindProperty]
2  [Required(
3      ErrorMessage = "Por favor, seleccione un fichero de texto.")]
4  public IFormFile? TextFile { get; set; }
5
6  public FileInputModel(
```

```
7       IOptions<TextFileValidationOptions> textFileValidationOptions)
8   {
9       _options = textFileValidationOptions.Value;
10  }
11
12  public async Task<IActionResult> OnPostAsync()
13  {
14      // ...
15      if (TextFile.Length > _options.MaxSizeKiloBytes * 1024)
16      {
17          ModelState.AddModelError(
18              nameof(TextFile),
19              $"El fichero es demasiado grande (máximo {_options.
                    MaxSizeKiloBytes}KB).");
20          return Page();
21      }
22      // ...
23  }
```

5.7.1.3. Página de expresiones y locuciones encontradas

Una vez que este ha introducido el texto, o subido el fichero pertinente, se redirige al usuario a la tercera y última página de la aplicación, que contendrá un listado con todas las expresiones y locuciones encontradas en el texto, o en el contenido del fichero subido. Como se puede apreciar en la figura 5.14, también se muestra el texto en sí, con una serie de enlaces de color rojo en el mismo. Cada uno de estos enlaces se corresponde con una expresión o locución diferente. Al hacer clic en uno de ellos, la página se desplaza verticalmente, hasta mostrar la información de la expresión o locución en cuestión.

5.7.2. Segundo prototipo de la aplicación web

Además de las características imprescindibles, incluidas en el prototipo inicial, ha sido posible implementar funcionalidad adicional. En concreto, se ha añadido un botón que permite descargar un fichero PDF con el listado de locuciones y expresiones, encontradas en el texto previamente introducido por el usuario. Esto se muestra en la figura 5.15.

Además, se han agregado algunas líneas de código Javascript, con el fin de añadir un borde alrededor de la información correspondiente a la locución o expresión cuyo enlace ha sido clicado, tal y como se puede observar en la figura 5.16. Esto facilita notablemente el seguimiento y consulta de las mismas.

Por otro lado, se han llevado a cabo mejoras de la accesibilidad, incluyendo algunos atributos ARIA pertinentes, utilizando las etiquetas HTML semánticas de manera adecuada y añadiendo los metadatos más relevantes. Asimismo, se han implementado algunos cambios visuales, con el propósito de lograr un diseño adaptable y proporcionar una experiencia de usuario agradable en una amplia variedad de dispositivos, incluyendo móviles, tablets, portátiles y ordenadores de escritorio.

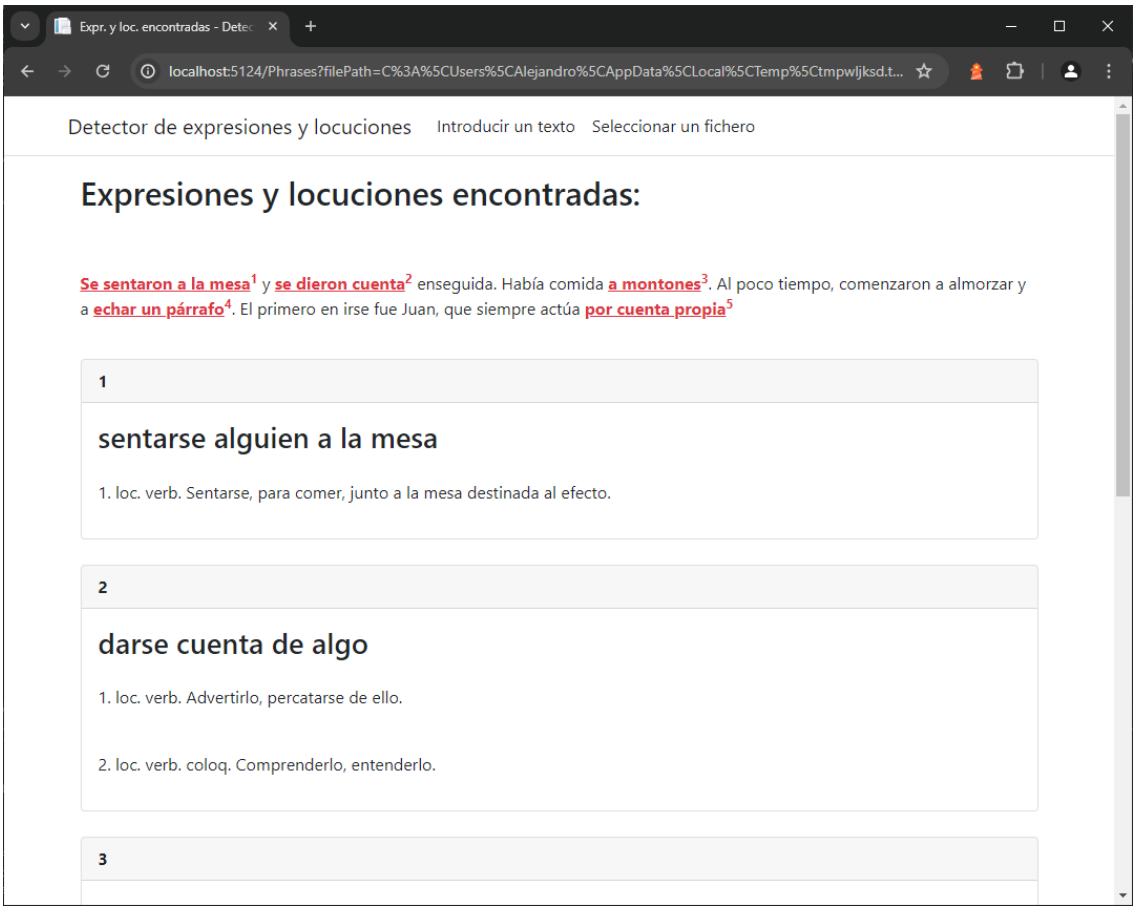

Figura 5.14: Primer prototipo de la aplicación web. Página de expresiones y locuciones encontradas.

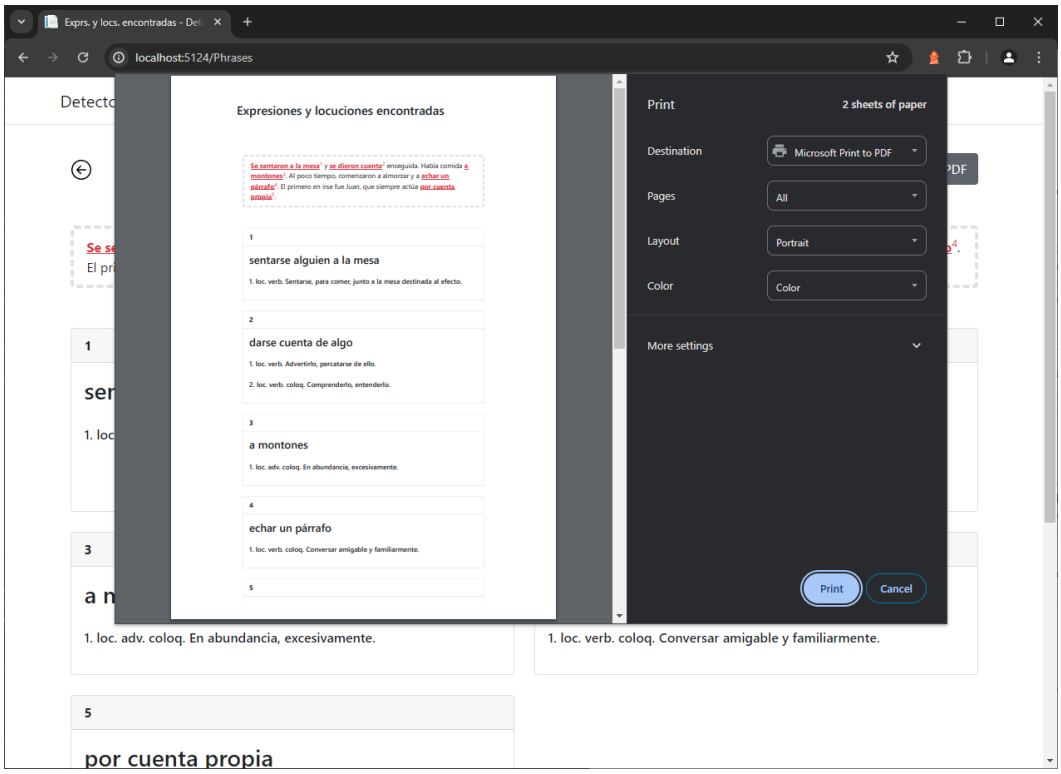

Figura 5.15: Función de guardar como PDF el listado de expresiones y locuciones encontradas en el segundo prototipo de la aplicación web.

Por último, se ha implementado una mejora fundamental en el manejo del texto introducido por el usuario. Previamente, no se realizaba ningún tipo de filtrado o codificación del mismo, lo cual podía conducir a errores al momento de mostrar el texto, en caso de que el usuario hubiera introducido caracteres que se consideran reservados en HTML, como el signo menor que (<) o el signo et (&, también denominado *ampersand*). Un ejemplo muy claro de ello se puede ver en la figura 5.17, donde el diseño del listado de expresiones y locuciones queda totalmente desconfigurado, tras haber introducido el siguiente texto en el formulario inicial:

```
<button> Este texto contiene caracteres reservados en HTML </button>
```

Sin embargo, tras los cambios realizados, el texto se codifica en HTML, logrando que se muestre exactamente lo que el usuario introdujo. Ahora, al introducir el texto que anteriormente causó problemas, el comportamiento es el esperado, como se puede apreciar en la figura 5.18.

Por otra parte, dado que la lógica que construye el HTML que muestra el texto introducido, junto con los enlaces de las expresiones y locuciones que contiene, se ha vuelto demasiado compleja para introducirla directamente en la plantilla HTML de la página, esta se ha desarrollado como un método de ayuda, es decir, un *helper*, de la vista. Concretamente, se trata de un método de extensión de la interfaz `IHtmlHelper`, como se puede ver en el fragmento 5.28.

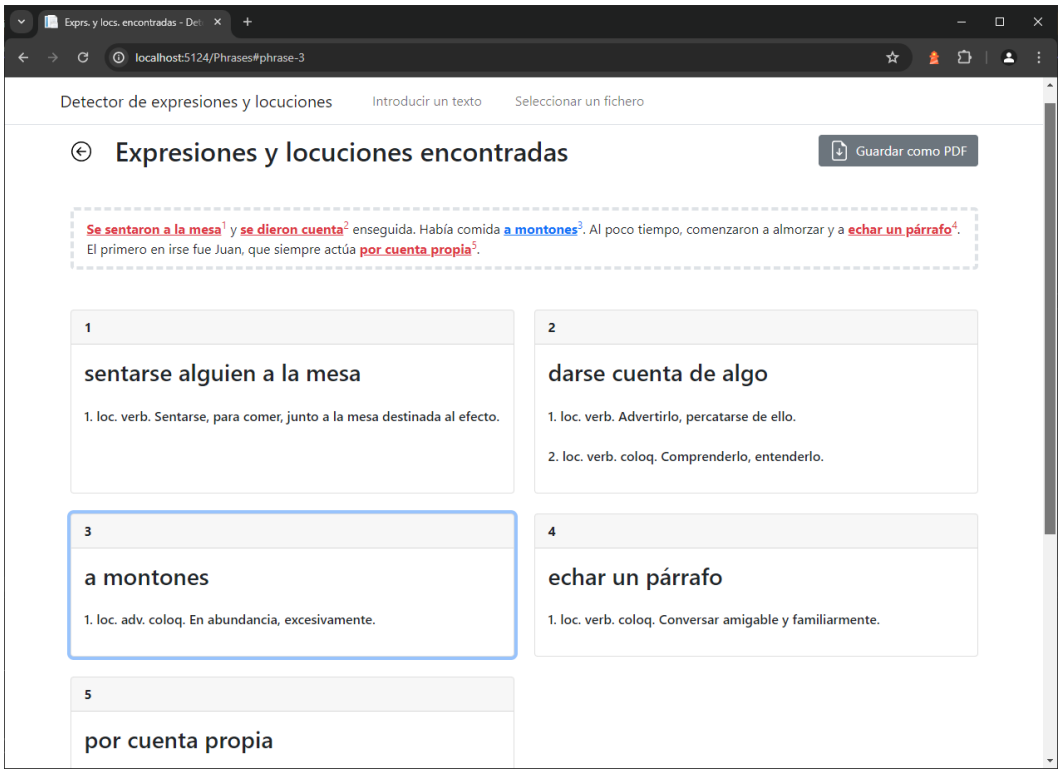

Figura 5.16: Resultado de hacer clic en el enlace correspondiente a la locución *a montones* en el segundo prototipo de la aplicación web.

Fragmento de código 5.28: Firma del método de extensión del *helper* HTML, que construye el código HTML del listado de expresiones y locuciones encontradas.

```
1 public static IHtmlContent HighlightPhrases(
2     this IHtmlHelper htmlHelper,
3     string text,
4     FoundPhrase[] foundPhrases)
5 {
6     // ...
7 }
```

La interfaz `IHtmlHelper` posee un método `Encode()`, utilizado para codificar en HTML el texto introducido por el usuario, lo cual soluciona el problema ilustrado en la figura 5.17. Además, puesto que el *helper* `Html` es accesible por defecto desde la vista, el método `HighlightPhrases()` es muy fácil de utilizar, pudiendo ser invocado directamente dentro del código HTML, como se puede apreciar en el fragmento 5.29.

Fragmento de código 5.29: Uso del método `HightlightPhrases()`, que construye el HTML del texto, junto con los enlaces de las expresiones y locuciones que contiene, en el código HTML de la página de expresiones y locuciones encontradas.

```
1 // ...
2 @if (Model.FoundPhrases?.Length > 0)
3 {
4     <p class="my-5 user-text-paragraph border border-4 rounded-3 fw-medium
```

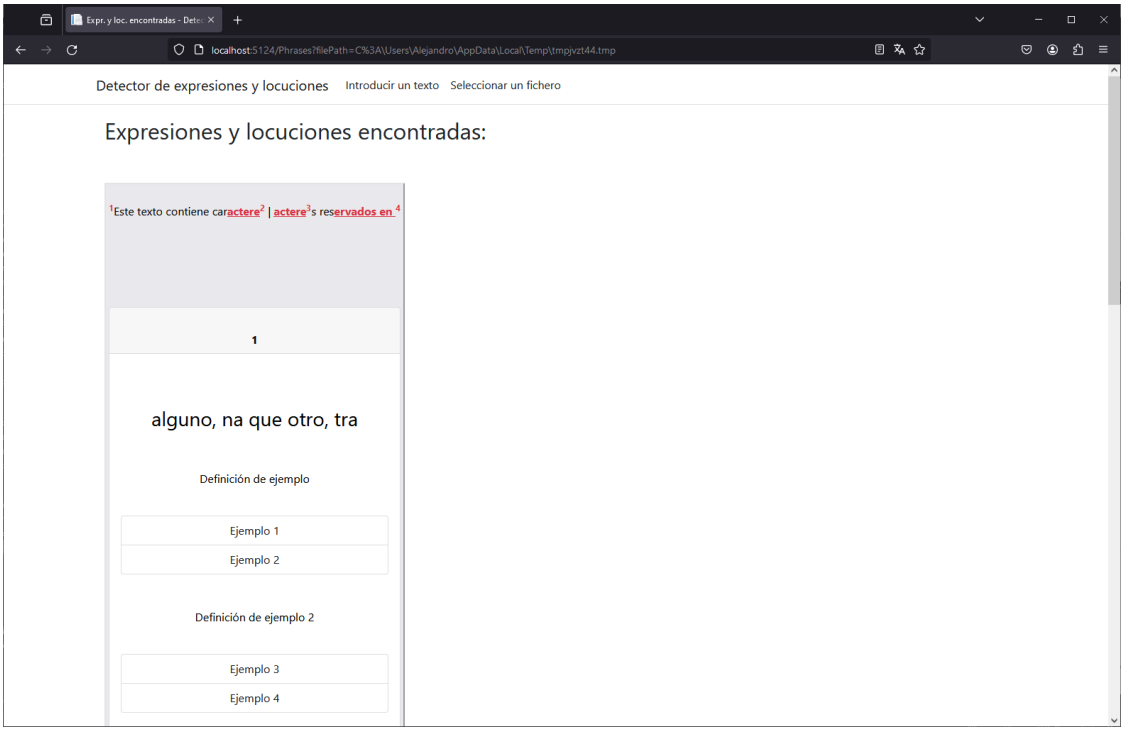

Figura 5.17: Desconfiguración de los estilos del listado de expresiones y locuciones, resultado de introducir un texto que incluye caracteres reservados en HTML, en el primer prototipo de la aplicación web.

Figura 5.18: Resultado de introducir un texto que contiene caracteres reservados en HTML en el segundo prototipo de la aplicación web, tras realizar los cambios en el manejo de la entrada del usuario.

```
            ">
5           @Html.HighlightPhrases(Model.Text, Model.FoundPhrases)
6       </p>
7       // ...
8   }
9   // ...
```

Nótese que el uso de código C# directamente en la página HTML, como en este último fragmento, es posible gracias a la sintaxis `Razor`, característica fundamental de la herramienta Razor Pages.

5.7.3. Versión final

Una vez completado el desarrollo del servicio WCF, se pudo sustituir el servicio de prueba en la aplicación web, utilizando, a partir de este momento, detección real de expresiones y locuciones, de acuerdo con la información presente en la base de datos. Gracias al sistema de inyección de dependencias, este cambio tan solo ha requerido sustituir una sola línea de código en el punto de partida de la aplicación, el fichero `Program.cs`. En concreto, se ha sustituido la línea que añadía el servicio de prueba al contenedor de servicios de la aplicación, por una que agrega el servicio real, como se puede observar en el siguiente fragmento.

Fragmento de código 5.30: Modificación realizada en `Program.cs` para sustituir el servicio de prueba (línea comentada), utilizado durante el desarrollo, por el servicio WCF real, ya completado.

```
1 // builder.Services.AddSingleton<IPhraseFinderService,
      PhraseFinderServiceDev>();
2 builder.Services.AddSingleton<IPhraseFinderService,
      PhraseFinderServiceClient>();
```

De esta manera, se satisface finalmente el principal objetivo de este trabajo, la detección automática de expresiones y locuciones de la lengua española en cualquier texto.

Por otra parte, dado que el servicio toma unos segundos en encontrar todas las expresiones y locuciones en el texto introducido por el usuario, se ha modificado la página de locuciones y expresiones para que, mientras esto sucede, se muestre un mensaje indicándolo, junto al texto que el usuario introdujo, como puede verse en la figura 5.19.

5.7.4. Accesibilidad y usabilidad

La accesibilidad ha sido un aspecto a tener en cuenta durante el desarrollo de la aplicación de escritorio. Tanto es así que, al analizar dicho parámetro con Lighthouse, popular herramienta de análisis de calidad de sitios web, se obtiene una puntuación excelente, como se puede ver en la figura 5.20. Entre los resultados del informe de accesibilidad, destacan el empleo adecuado de las etiquetas HTML semánticas y los atributos ARIA. Sumado a ello, toda la funcionalidad es accesible a través del teclado.

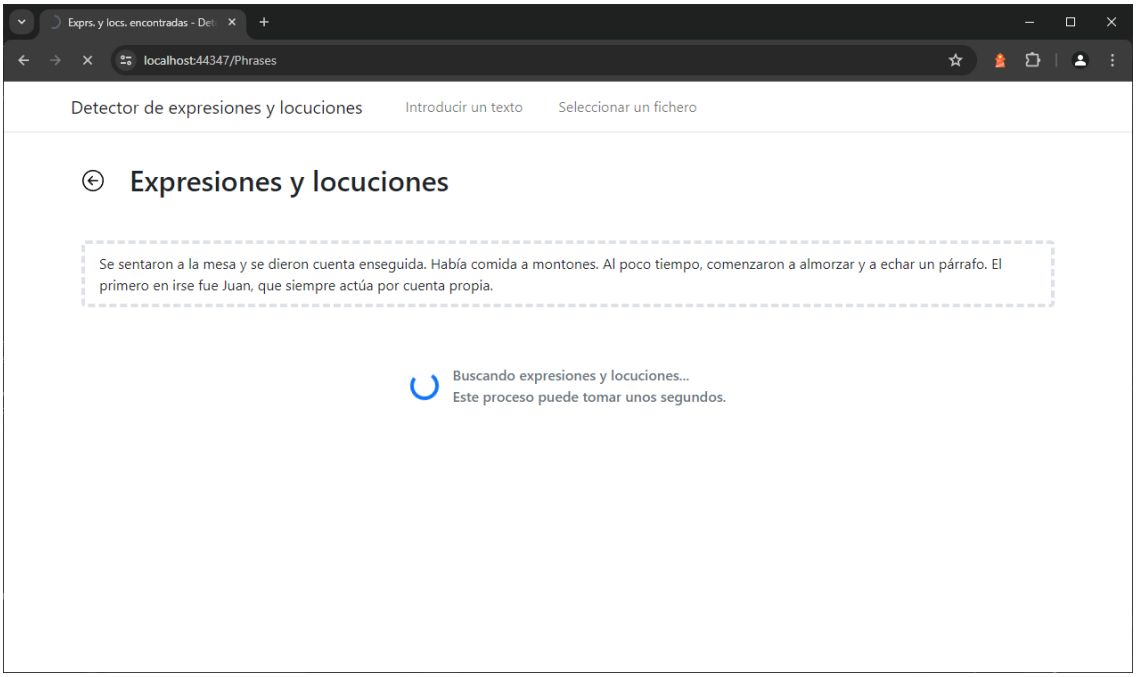

Figura 5.19: Página de expresiones y locuciones, mientras se realiza la detección de las mismas.

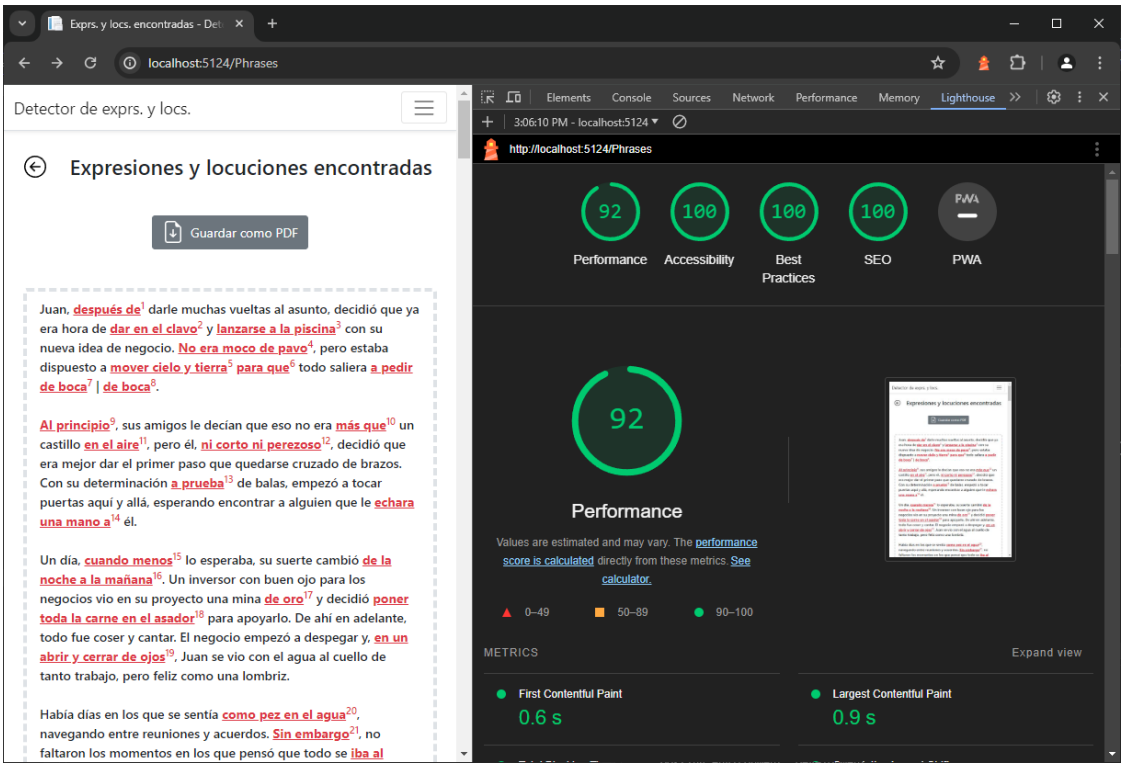

Figura 5.20: Resultado de ejecutar un análisis de calidad de la página de locuciones y expresiones encontradas, utilizando la herramienta Lighthouse con la configuración por defecto.

Asimismo, cabe mencionar que la práctica totalidad del código de la aplicación web se ejecuta en el lado del servidor, por lo que el Javascript que se envía al cliente es mínimo. Esto agiliza considerablemente el tiempo de carga de la web y hace posible que, aún con la ejecución de código Javascript deshabilitada en el navegador, el sitio sea perfectamente utilizable. La única función que no está disponible en este caso es la de enfocar la expresión y locución al clicar en su correspondiente enlace.

Por otra parte, la usabilidad es también una característica destacable de la aplicación, principalmente debido a su simplicidad. Además, los elementos interactivos son fácilmente distinguibles e indican su propósito y estado. Esto se aplica a todos los botones de la aplicación, a ambos formularios, a los enlaces de navegación y a los enlaces a las expresiones y locuciones encontradas. Por último, se ha prestado mucha atención tanto al manejo de errores como a la validación de entradas del usuario. Un ejemplo de ello son los mensajes de validación en ambos formularios de la aplicación.

5.8. Desarrollo del servicio web WCF

El servicio web WCF, encargado de la detección de las expresiones y locuciones, se encuentra en cierto modo segregado del resto del trabajo, dado que, por motivos de compatibilidad con otras aplicaciones ya desarrolladas por el IATEXT, en las cuales se espera poder integrar el servicio, es un requisito indispensable que este sea desarrollado en .NET Framework 4.7.2, una versión considerablemente antigua de dicha herramienta. Es por ello que en este proyecto no es posible reutilizar el código de las capas de dominio y de datos, pues desde .NET Framework no existe la posibilidad de llamar a código desarrollado en .NET (anteriormente denominado .NET Core), aunque aplicaciones desarrolladas en este último sí pueden hacer uso del servicio, como en el caso de la aplicación web.

No obstante, esto no supone un impedimento inabordable, pues el servicio tan solo debe realizar algunas consultas simples a la base de datos, haciendo prescindible el uso de la capa de datos. Además, de la capa de dominio solo será necesario recrear, parcialmente, los modelos correspondientes a los patrones de detección (`PhrasePattern`, fragmento de código 5.8), las definiciones y los ejemplos (`PhraseDefinition` y `PhraseExample`, respectivamente, ambos incluidos en el fragmento 5.12).

Cabe mencionar que el desarrollo del servicio web WCF ha consistido en la creación de un único prototipo funcional, al que se le han ido aplicando mejoras en la lógica de detección, de forma iterativa. No se han implementado características adicionales, más allá de las especificadas en los requisitos, descritos en la sección 4.2.

5.8.1. Consultas a la base de datos

En primer lugar, será necesario obtener los patrones que permiten la detección de las expresiones y locuciones, por medio de consultas a la base de datos. Para facilitar el uso de la misma, se ha empleado Dapper, un intuitivo ORM (Object-Relational Mapper) que, si bien no proporciona tantas herramientas, cuenta con algunas ventajas frente a Entity Framework Core, el utilizado en la capa de datos. En primer lugar, es mucho más sencillo de configurar y de utilizar, pues su uso se basa en la creación directa de sentencias SQL. Tras ejecutarlas, el ORM transforma automáticamente los registros devueltos por la base de datos en objetos C#. Además, ofrece un rendimiento algo superior al de Entity Framework Core. Se muestra a continuación, en el fragmento 5.31, un ejemplo de uso, en particular el método `GetPhrasePatterns()`, que devuelve todos los patrones de detección de expresiones y locuciones contenidos en la base de datos. Este se encuentra en la clase `PhrasePatternService`.

Fragmento de código 5.31: Método que consulta los patrones de detección de la base de datos, denominado `GetPhrasePatterns()` y localizado en la clase `PhrasePatternService`.

```
1 // ...
2
3 private const string PhrasePatternQuery = @"
4 select
5     [Locucion_o_expresion] as [Phrase],
6     [Patron] as [Pattern],
```

```
7       [Palabra_base] as [BaseWord],
8       [ID_Locucion] as [PhraseId]
9   from Patrones;";
10
11  // ...
12
13  public IEnumerable<PhrasePattern> GetPhrasePatterns()
14  {
15      return _dbConnection.Query<PhrasePattern>(PhrasePatternQuery);
16  }
17
18  // ...
```

Cabe mencionar que, una vez obtenidos los patrones de detección, estos se almacenan en un campo del servicio de detección, denominado `_patterns`, con el fin de evitar consultarlos en cada petición.

5.8.2. Librería de procesamiento de textos

El primer paso en el proceso de detección de las expresiones y locuciones en un texto, es la separación del texto en cuestión en párrafos y frases. Esta tarea, por muy simple que pueda parecer, acarrea cierta complejidad, pues en español las frases pueden dividirse de diversas maneras, como por un punto, un interrogante y un signo de exclamación, entre otros. Además, el texto puede contener puntuación ambigua o tener un formato inconsistente. No obstante, esto no supone un problema, pues el IATEXT cuenta con una librería, denominada `ProcesarTexto`, que proporciona toda la lógica necesaria. Dicha librería ha sido agregada al proyecto de aplicación WCF, por medio de un fichero DLL (Dynamic-Link Library), que no se encuentra en el repositorio del trabajo.

En el siguiente fragmento de código, se muestra un ejemplo de uso de esta librería, concretamente el método de extensión que obtiene los párrafos que constituyen el texto suministrado como parámetro.

Fragmento de código 5.32: Método de extensión que divide el texto en párrafos, haciendo uso de la librería `ProcesarTexto`.

```
1  public static Paragraph[] GetParagraphs(this string text)
2  {
3      var textProcessor = new ProcesarTextos.Text(string.Empty, text);
4      return textProcessor.GetParagraphs();
5  }
```

5.8.3. Servicio de *lematización*

Este servicio, desarrollado por el IATEXT, aporta información esencial acerca de las categorías gramaticales y formas canónicas de las palabras en el texto introducido. Esto permite, por ejemplo, comprobar si una palabra localizada en el texto es un verbo y, en

caso de que así sea, comparar su forma canónica, es decir, la forma no conjugada, que no indica tiempo, modo, número ni persona (*decir*, *ir*, *haber*, etc). De esta manera, puesto que las locuciones que contienen verbos suelen incluirlos en forma canónica, estas podrán ser detectadas adecuadamente, independientemente de la conjugación utilizada en el texto.

No obstante, este servicio no es infalible, pues existen, en la lengua española, multitud de conjuntos de palabras diferentes que se escriben igual. Este es el caso, por ejemplo, del condicional del indicativo del verbo *caber*, *cabría*, que puede confundirse con el femenino del adjetivo *cabrío* - "Perteneciente o relativo a las cabras" (Diccionario de la lengua española). Por supuesto, no es posible determinar, en estos casos, cuál es la palabra en cuestión, sin utilizar herramientas de inteligencia artificial o complejísimos análisis del texto.

5.8.4. Interfaz del servicio

Las aplicaciones o *clientes* que se conectan al servicio de detección de expresiones y locuciones lo hacen por medio de la interfaz pública del mismo, denominada `IPhraseFinderService`. Se puede ver en el fragmento de código 5.33.

Fragmento de código 5.33: Interfaz `IPhraseFinderService`, expuesta por el servicio WCF.

```
1 [ServiceContract]
2 public interface IPhraseFinderService
3 {
4     [OperationContract]
5     Task<PhraseAnalysis> FindPhrasesAsync(string text);
6 }
```

Como se puede observar, la interfaz incluye un único método asíncrono, denominado `FindPhrasesAsync()`. Este toma como parámetro una cadena de caracteres cualquiera, que se corresponde con el texto del cual se desean extraer las locuciones y expresiones. El método devuelve un objeto de tipo `PhraseAnalysis`, que incluye el texto tras ser procesado y una colección, concretamente un *array* de instancias de `FoundPhrase`, con todas las locuciones y expresiones encontradas en el texto. La clase `FoundPhrase` contiene toda la información relevante sobre las coincidencias de locución o expresión. Tal y como se especificó en los requisitos del servicio web (sección 4.2), esta incluye localización y longitud de la coincidencia en el texto introducido, nombre de la locución o expresión en cuestión, definición o definiciones de la misma, ejemplos de uso y palabra base.

Por otra parte, cabe mencionar que la interfaz `IPhraseFinderService` hace uso de los atributos `ServiceContract` y `OperationContract`, estándares en el desarrollo de servicios WCF. Estos determinan que el método ha de tener un correspondiente punto final o *endpoint* en la API expuesta por el servicio.

5.8.5. Lógica de detección de locuciones y expresiones

La detección de locuciones y expresiones se realiza en la clase que implementa la interfaz del servicio WCF, denominada `PhraseFinderService`, concretamente en los métodos `FindPhrasesAsync()` y `FindPhrasesInSentences()`, que se muestran en el fragmento 5.34.

Fragmento de código 5.34: Métodos `FindPhrasesAsync()` y `FindPhrasesInSentences`, que detectan las expresiones y locuciones localizadas en el texto.

```
1  // ...
2
3  public async Task<PhraseAnalysis> FindPhrasesAsync(string text)
4  {
5      var paragraphs = text.GetParagraphs();
6      var sentences = paragraphs.SelectAllSentences().ToList();
7      sentences = await _servicioLematizacion.NuevoReconocerFrasesAsync(
8              sentences,
9              idioma: "es",
10             multiPref: false);
11
12     var foundPhrases = FindPhrasesInSentences(sentences, paragraphs)
13         .Distinct()
14         .ToArray();
15     IncludeDefinitions(ref foundPhrases);
16
17     return new PhraseAnalysis
18     {
19         ProcessedText = paragraphs.ReconstructText(
20             ParagraphSeparator,
21             SentenceSeparator),
22         FoundPhrases = foundPhrases
23     };
24 }
25
26 private IEnumerable<FoundPhrase> FindPhrasesInSentences(
27     IReadOnlyCollection<InfoUnaFrase> sentences,
28     IReadOnlyCollection<Paragraph> paragraphs)
29 {
30     // ...
31     foreach (var pattern in _patterns)
32     {
33         var sentenceIndex = 0;
34         // ...
35
36         foreach (var sentence in sentences)
37         {
38             var matches = pattern.FindPhrase(sentence, sentenceIndex);
39
40             foreach (var match in matches)
41             {
42                 yield return match;
43             }
44
45             sentenceIndex += sentence.Frase.Length;
```

```
46                 // ...
47             }
48         }
49     }
50
51     // ...
```

Como se puede observar, la lógica de detección consiste en un bucle principal, que recorre un *array* con todos los patrones, extraídos de la base de datos. A su vez, hay un segundo bucle anidado dentro de este, que itera sobre las frases contenidas en el texto introducido por el usuario, obtenidas previamente gracias a la librería comentada en la subsección 5.8.2. Dentro de este último, se llama al método `FindPhrase()` de la clase `PhrasePattern`, suministrando como parámetro la frase actual, en la cual se buscan coincidencias del patrón en cuestión. Cada coincidencia corresponderá a una instancia de `FoundPhrase`, devuelta a su vez por el método `FindPhrasesInSentences()`.

En el método `FindPhrase()`, se comprueba, para cada locución o expresión, si la primera palabra que esta contiene coincide con alguna palabra del texto. En caso de ser así, se verifica si el resto de la expresión coincide también con las subsiguientes palabras del texto.

Tan solo la lógica que se ha descrito hasta este momento permitiría encontrar una proporción significativa de las locuciones y expresiones de la lengua española, en concreto aquellas que no varían en absoluto, como *por cierto*, o *a duras penas*. No obstante, la mayor parte de las mismas contienen verbos, los cuales pueden encontrarse de diversas formas distintas en el texto, dependiendo de su conjugación. Por ello, se hace uso del servicio de lematización para obtener la forma canónica de los verbos localizados en el texto, para comparar esta con la palabra de la expresión o locución que está siendo buscada.

Esto es sumamente efectivo, pues la gran mayoría de verbos contenidos en las locuciones o expresiones, al menos en los casos en que estos pueden tomar distintas formas, se encuentran en su forma canónica. Una excepción notable a esta regla, sin embargo, son las expresiones en que el verbo se encuentra en forma pronominal, es decir, terminado en *-me*, *-te*, *-le(s)*, *-lo(s)*, *-la(s)*, *-se*, *-nos* u *-os*. Este es el caso, por ejemplo, de la locución verbal *darse cuenta de algo*. Estos casos también se han tenido en consideración, añadiendo a la lógica detección una colección estática con todas los sufijos correspondientes a la forma pronominal, de manera que se compara la palabra extraída de la expresión o locución, con la forma canónica del verbo contenido en el texto, añadiendo cualquiera de estos sufijos. Asimismo, se han tenido en cuenta las formas compuestas de los verbos (*hubieran cantado*, *ha visto*, ...), en los cuales *haber* precede al verbo, así como los casos en los que se utiliza un pronombre átono antes del verbo (*se han sentado*, *te diste*, *nos vamos*, ...).

Por otra parte, las locuciones y expresiones a menudo contienen las palabras *algo*, *alguien*, o incluso *algo y alguien*. Estas se utilizan para determinar que el contenido que la locución o expresión puede albergar en esa posición puede variar, como en la locución vista en el párrafo anterior, *darse cuenta de algo*. Esto se ha contemplado en la lógica de detección, pues se ha ofrecido cierta flexibilidad a la hora de procesarlas, considerándose que en la posición en que se encuentra una de estas palabras *comodín*, pueden hallarse de 0 a 3 palabras cualesquiera.

No obstante, existen locuciones en que, por ejemplo, la palabra *algo* es la palabra base de la misma, como en *algo es algo*. En estos casos, la propia palabra debe encontrarse en el texto, en lugar de servir para indicar variabilidad. Estas situaciones también se han tenido en cuenta.

Teniendo en consideración todos los parámetros mencionados, se obtiene una detección considerablemente precisa de la gran mayoría de locuciones y expresiones de la lengua española. No obstante, se producen algunos falsos positivos, ocasiones en que se detecta una expresión o locución que en realidad no se encuentra en el texto. Por lo general, esto se debe al sentido de la frase u oración donde se ha detectado, puesto que el servicio desarrollado tan solo analiza patrones de palabras, no el significado ni el contexto de las mismas, por lo que se dan casos como el siguiente:

Su collar tiene un corazón de oro.

En esta oración, se utiliza el significado literal de *corazón de oro*, por lo que en realidad no se está haciendo uso de la locución *tener un corazón de oro* (ser una persona destacablemente bondadosa). No obstante, el sistema lo considera una coincidencia de la locución verbal, puesto que el patrón de palabras concuerda.

Por otra parte, existen también casos en que la detección no encuentra locuciones y expresiones que el texto sí contiene. En algunos casos, esto se debe a patrones defectuosos, resultado de la separación de las diferentes variantes de las locuciones y expresiones. Esto se debe a que, como se mencionó en distintas ocasiones en la sección 5.4.3, el diccionario no es consistente al especificar estas variaciones, dificultando enormemente el procesamiento automático de las mismas. Por ejemplo, en el caso de *dormirse alguien sobre los laureles, o en los laureles*, se obtienen los patrones *dormirse alguien sobre los laureles* y *dormirse alguien sobre en los laureles*, siendo el segundo erróneo. No obstante, en otras ocasiones este problema lo causa la lógica de detección, como en el siguiente caso:

Se sentaron todos los comensales muy rápidamente a la mesa.

En este caso, no se produce la detección de la locución verbal *sentarse alguien a la mesa*, debido a que la oración contiene un número de palabras mayor al esperado en la posición de la palabra comodín *alguien*. Sí resultaría en una coincidencia, por ejemplo, la oración *Se sentaron todos rápidamente a la mesa*.

Desafortunadamente, aunque se podrían implementar mejoras en el sistema, sería prácticamente imposible resolver por completo, y de manera automática, los problemas descritos. Esto se debe a que no existe herramienta alguna, ni sería sencillo desarrollarla, que proporcione toda la información necesaria para la correcta detección de las expresiones y locuciones de la lengua española, incluyendo exactamente todas sus posibles variantes y sus correspondientes flexiones, conjugaciones verbales y palabras con las que se combinan, entre otros aspectos. Por ello, esta tarea requiere de un laborioso e intrincado trabajo manual, por parte de expertos lingüistas.

Como ejemplo de este trabajo manual, se muestran a continuación algunos de los patrones de detección para la locución verbal *levantarse alguien de la mesa*, cuidadosamente elaborados

de forma manual por estudiantes de filología, que se encontraban haciendo prácticas en el IATEXT.

- `<Pron. Pers. Át. Refl.> $levantar [<pron. Pers. Tón.>] [<S. Prep.>] [<Loc . Adv.>] de la mesa`
- `<Pron. Pers. Át. Refl.> $levantar [<pron. Pers. Tón.>] [<S. Prep.>] [<adv1 >] [<adv2>] de la mesa`
- `[<Pron. Pers. Át. Refl.>] <$verbo> [<infinitivo>] levantar [<Pron. Pers. Át. Refl.>] [<Pron. Pers. Tón.>] [<S. Prep>] [<Adv>] de la mesa`

Como se puede observar, estos incluyen distintos símbolos, que delimitan *etiquetas* que describen las palabras que ha de contener una coincidencia de la locución o expresión en cuestión. El símbolo de dólar ($) indica que la palabra precedida por este puede encontrarse en cualquiera de sus formas flexivas, mientras que las abreviaturas entre los símbolos menor que (<) y mayor que (>) determinan la categoría a la que puede pertenecer la palabra localizada en la posición correspondiente. Además, el contenido que se sitúa entre corchetes ([ ]) es opcional. El resto de palabras, que no incluyen ninguno de estos símbolos, han de encontrarse de forma exacta en el texto. Cabe mencionar que, solo para esta locución, se encontraron treinta patrones diferentes y, en total, este proceso se ha llevado a cabo con 42 locuciones y expresiones distintas. Por supuesto, se trata de un procedimiento considerablemente complejo y que consume una gran cantidad de tiempo, por lo que aplicarlo a todas y cada una de las casi 13500 expresiones y locuciones de la lengua española, recogidas en el Diccionario de la lengua española, sería una tarea monumental.

Aunque el uso de estos patrones, elaborados manualmente, disminuiría notablemente la frecuencia de los falsos positivos en la detección de las 42 expresiones y locuciones a las que corresponden, por una cuestión de tiempo no ha sido posible desarrollar al completo la lógica necesaria.

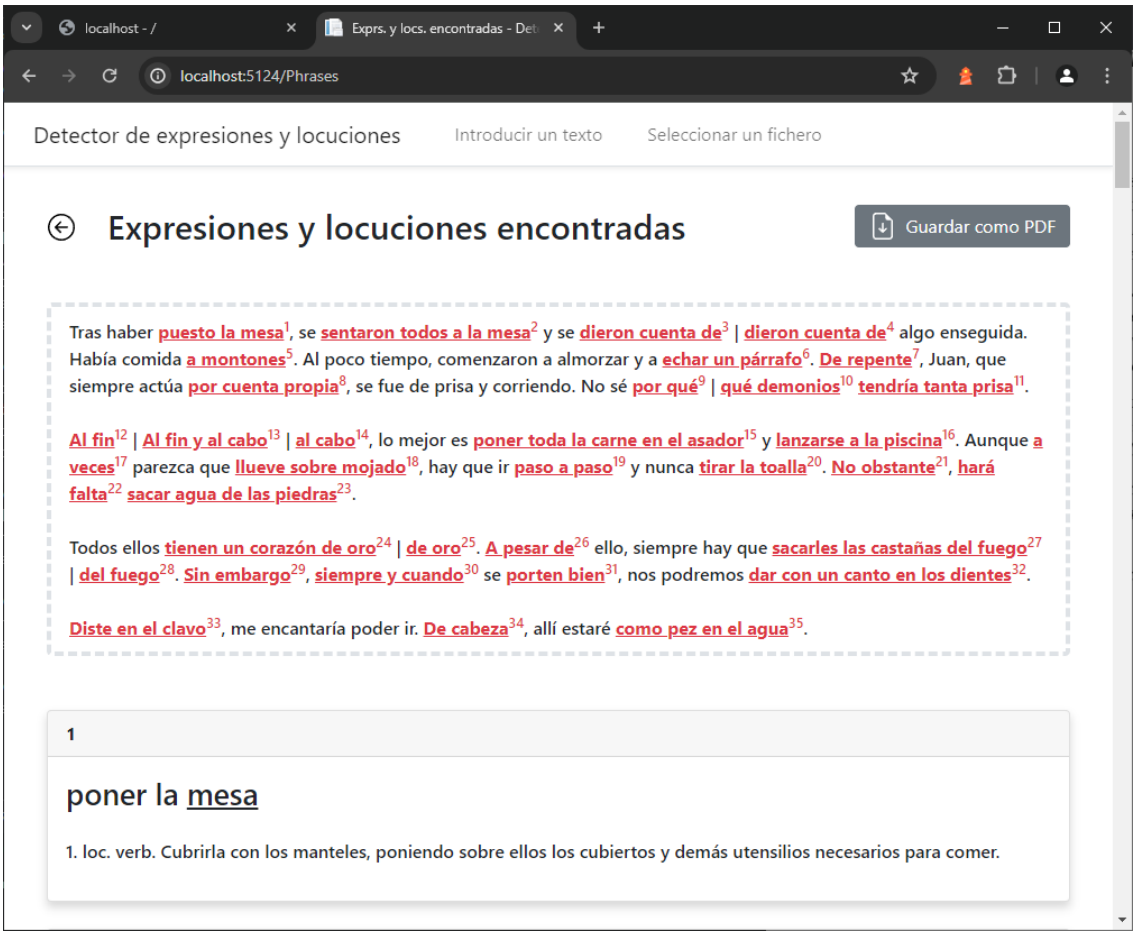

Figura 5.21: Ejemplo de detección de locuciones y expresiones en un texto, utilizando las versiones finales de la aplicación web y el servicio WCF.

5.9. Resultado final

Para concluir el capítulo, se muestra en la figura 5.21, una vez más, la aplicación web, concretamente su versión final haciendo uso de la última iteración del servicio web. Se observa el funcionamiento de la lógica de detección, tras introducir un texto que contiene una gran cantidad de locuciones y expresiones de la lengua española.

Capítulo 6

Conclusiones y trabajo futuro

La lógica de detección proporciona un margen de error razonable para la mayoría de locuciones y expresiones. No obstante, existen unas doscientas expresiones y locuciones, del total de aproximadamente 13.500, que no son detectadas adecuadamente, debido a la dificultad de procesarlas. Además, el sistema sufre, con cierta frecuencia, de falsos positivos, principalmente debidos al sentido de la frase u oración en cuestión, pues en ocasiones se pueden encontrar secuencias de palabras que podrían formar una locución o expresión y, sin embargo, al tener en cuenta la semántica y el contexto del texto, se determina que no es así.

Por tanto, se puede concluir que, aunque este TFT ha servido como un sólido punto de partida en el desarrollo de un sistema de detección automática de expresiones y locuciones, completando la infraestructura software necesaria y obteniendo una primera versión funcional, aún queda trabajo por hacer. Actualmente, no existe diccionario ni herramienta alguna que aporte toda la información necesaria para la adecuada detección en textos de todas las expresiones y locuciones de la lengua española, pues algunas de estas se pueden encontrar de numerosas formas distintas, dependiendo de la conjugación de los verbos y la flexión de las palabras que las componen, entre otros parámetros. Es por ello que, en estos momentos, no es posible automatizar por completo el proceso, el cual requiere de un complejo y laborioso análisis manual de las expresiones y locuciones.

Es más, aunque se analizaran a la perfección los patrones de palabras que componen las locuciones y expresiones, continuaría existiendo cierto margen de error. Esto se debe a que, en un texto, cabe la posibilidad de encontrar secuencias de palabras que podrían formar una locución o expresión y, sin embargo, al tener en cuenta la semántica de la oración, se determina que no es así. Estos casos serían sumamente difíciles de abordar, pues requerirían del uso de herramientas muy avanzadas de inteligencia artificial, o de análisis lingüísticos extremadamente complejos, capaces de interpretar el contexto y significado de las palabras.

Por otra parte, el sistema desarrollado ofrece interesantes posibilidades de expansión. En primer lugar, se podría incorporar la información de distintos diccionarios, además del Diccionario de la lengua española de la Real Academia Española, utilizado a lo largo de este trabajo. De hecho, el sistema se ha desarrollado teniendo en cuenta esta posibilidad, por medio del uso de interfaces y de la aplicación del patrón `Factory`, abstrayendo por completo

las particularidades del formato del diccionario del resto de la lógica del sistema.

Asimismo, sería posible, incluso, añadir diccionarios de otras lenguas. La inglesa, por ejemplo, sería especialmente interesante, pues en dicho idioma las locuciones y expresiones, denominadas *phrases* o *idioms*, son utilizadas constantemente y cobran una importancia mayúscula en las interacciones coloquiales. No obstante, esta ampliación requeriría realizar cambios significativos en la lógica de detección de las expresiones y locuciones, pues las herramientas desarrolladas por el IATEXT, de las cuales depende en gran medida, están orientadas al análisis de textos en español.

Además de los aspectos ya comentados, también podría resultar de utilidad la incorporación, a la aplicación web, de un listado con todas las expresiones y locuciones disponibles, que permitiera a los usuarios ordenar, filtrar y buscar entre todas las locuciones y expresiones almacenadas en la base de datos. Sería una funcionalidad muy similar al listado de expresiones y locuciones disponible en la aplicación de escritorio, descrito en la sección 5.6.1.3.

En lo que respecta a la aplicación de escritorio, aunque se han alcanzado todos los objetivos iniciales relacionados con la misma, esta presenta, por supuesto, algunas oportunidades de mejora. En primer lugar, el procesamiento y almacenamiento de las expresiones y locuciones puede llegar a tomar unos minutos, dependiendo de la carga del sistema. Esto se debe a que el código que ejecuta dichas tareas, localizado en las capas de dominio y de datos, no ha sido optimizado rigurosamente, pues se han priorizado la legibilidad y mantenibilidad del mismo, más que su rendimiento.

Por otra parte, si bien es cierto que no se les ha dado demasiada importancia, pues la interfaz gráfica de Microsoft Access ya proporciona estas características, podrían incluirse más herramientas de filtrado y búsqueda en el listado de expresiones y locuciones, presente en la aplicación de escritorio. Sería particularmente útil, por ejemplo, la posibilidad de filtrar mediante expresiones regulares, pues esta funcionalidad se ha utilizado constantemente durante el desarrollo del proyecto. Algo similar ocurre con las definiciones, ejemplos y patrones de las locuciones y expresiones. Si bien estos pueden ser inspeccionados directamente en la base de datos, podría ser de utilidad disponer de esta característica en la aplicación de escritorio.

Bibliografía

[1] Anónimo. *Glosario de términos gramaticales. locución*. Real Academia Española y Asociación de Academias de la Lengua Española. 2019. URL: https://www.rae.es/gtg/locuci%C3%B3n (visitado 25-02-2024).

[2] Anónimo. *Glosario de términos gramaticales. composicionalidad*. Real Academia Española y Asociación de Academias de la Lengua Española. 2019. URL: https://www.rae.es/gtg/composicionalidad (visitado 25-02-2024).

[3] Anónimo. *Diccionario de la lengua española. Vigesimotercera edición*. Real Academia Española. 2014. URL: https://dle.rae.es/ (visitado 25-02-2024).

[4] Anónimo. *¿Qué es un modelo de lenguaje grande (LLM)?* Elastic. 2023. URL: https://www.elastic.co/es/what-is/large-language-models (visitado 07-04-2024).

[5] Anónimo. *Developer Survery 2023. Most popular technologies*. Stack Overflow. 2023. URL: https://survey.stackoverflow.co/2023/#technology-most-popular-technologies (visitado 27-02-2024).

[6] Contribuidores de Wikipedia. *.NET Framework*. Wikipedia, The Free Encyclopedia. 2024. URL: https://en.wikipedia.org/wiki/.NET_Framework (visitado 25-02-2024).

[7] Contribuidores de Wikipedia. *.NET*. Wikipedia, The Free Encyclopedia. 2024. URL: https://en.wikipedia.org/wiki/.NET (visitado 25-02-2024).

[8] Contribuidores de la documentación de .NET. *An overview of Windows development options*. Microsoft. 2023. URL: https://learn.microsoft.com/en-us/windows/apps/get-started/?tabs=winappsdk-winui%2Cnet-maui (visitado 25-02-2024).

[9] Contribuidores de la documentación de .NET Framework. *What Is Windows Communication Foundation*. Microsoft. 2023. URL: https://learn.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf (visitado 25-02-2024).

[10] Contribuidores de la documentación de .NET. *Introduction to Razor Pages in ASP.NET Core*. Microsoft. 2023. URL: https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-8.0 (visitado 25-02-2024).

[11] Contribuidores de la documentación de .NET. *Testing in .NET*. Microsoft. 2023. URL: https://learn.microsoft.com/en-us/dotnet/core/testing/ (visitado 25-02-2024).

[12] Himanshu Sheth. *NUnit vs. XUnit vs. MSTest: Unit Testing Frameworks*. LambdaTest. 2021. URL: https://www.lambdatest.com/blog/nunit-vs-xunit-vs-mstest/ (visitado 25-02-2024).

[13] Andrew Lock. «ASP.NET Core in Action». En: Tercera Edición. Manning Publications, 2023. Cap. 12.1, págs. 282-287. ISBN: 9781633438620.

[14] Anónimo. *Git feature branch workflow*. Atlassian. 2024. URL: https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow (visitado 25-02-2024).

[15] Shen Huang. *What is Big O Notation Explained: Space and Time Complexity*. freeCodeCamp. 2020. URL: https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/ (visitado 27-02-2024).

[16] Contribuidores de la documentación de C#. *C# guide. Iterators*. Microsoft. 2021. URL: https://learn.microsoft.com/en-us/dotnet/csharp/iterators (visitado 27-02-2024).

[17] Anónimo. *6. Expressions. Yield expressions*. Python Software Foundation. 2024. URL: https://docs.python.org/3/reference/expressions.html#yield-expressions (visitado 27-02-2024).

[18] Jon Skeet. «C# in Depth». En: Cuarta Edición. Manning Publications, 2019. Cap. 2.4, págs. 53-66. ISBN: 9781617294532.

[19] Contribuidores de la documentación de C#. *Task cancellation*. Microsoft. 2022. URL: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-cancellation (visitado 02-06-2024).

[20] Anónimo. *Factory Pattern*. Object Oriented Design. 2006. URL: https://www.oodesign.com/factory-pattern (visitado 01-06-2024).

[21] Contribuidores de la documentación de C#. *Registros (referencia de C#)*. Microsoft. 2023. URL: https://learn.microsoft.com/es-es/dotnet/csharp/language-reference/builtin-types/record (visitado 24-05-2024).

[22] Anónimo. *Data types for Access desktop databases*. Microsoft. 2024. URL: https://support.microsoft.com/en-us/office/data-types-for-access-desktop-databases-df2b83ba-cef6-436d-b679-3418f622e482 (visitado 02-06-2024).

[23] Contribuidores de la documentación de C#. *Métodos de extensión (Guía de programación de C#)*. Microsoft. 2024. URL: https://learn.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/extension-methods (visitado 22-05-2024).

[24] Contribuidores de la documentación de .NET. *Messenger*. Microsoft. 2022. URL: https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/messengery (visitado 31-05-2024).

[25] Contribuidores de la documentación de .NET. *Host genérico de .NET*. Microsoft. 2024. URL: https://learn.microsoft.com/es-es/dotnet/core/extensions/generic-host?tabs=appbuilder (visitado 22-05-2024).

[26] Anónimo. *What is an API (Application Programming Interface)?* Amazon. 2024. URL: https://aws.amazon.com/what-is/api/ (visitado 05-06-2024).

[27] Contribuidores de MDN. *ARIA*. MDN Web Docs. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA (visitado 15-05-2024).

[28] Contribuidores de la documentación de Microsoft. *What is a DLL*. Microsoft. 2024. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-client/setup-upgrade-and-drivers/dynamic-link-library (visitado 29-05-2024).

Glosario

API Application Programming Interface. Interfaz que puede ser entendida como un contrato de servicio entre dos aplicaciones. Dicho contrato define la manera en que ambos sistemas se comunican por medio de solicitudes y respuestas [26]. 69

ARIA Accessible Rich Internet Applications. Colección de atributos que definen cómo realizar contenido y aplicaciónes web (especialmente las desarrolladas con Javascript) más accesibles para las personas con discapacidades [27]. 59, 64

boilerplate Parte del código que se repite en varias ocasiones con poca o ninguna variación. 41

DLE Diccionario de la lengua española, obra editada y publicada por la Real Academia Española.. 1, 3, 22, 42

DLL Dynamic-Link Library. Librería que contiene código y datos que pueden ser utilizados por más de un programa al mismo tiempo. Por ejemplo, en los sistemas operativos Windows la DLL `Comdlg32` lleva a cabo funciones comunes relacionadas con los cuadros de diálogo. Cualquier programa puede acceder a dicha funcionalidad, contenida por la DLL, para implementar un cuadro de diálogo [28]. 68

endpoint Localización específica en una API que acepta solicitudes determinadas y proporciona respuestas a las mismas. 69

forma canónica Forma estándar o más comúnmente aceptada de una palabra o estructura gramatical. Es la forma que se utiliza como referencia en diccionarios y gramáticas y se considera la versión *normal* o *no marcada* de una palabra o construcción. 16, 68, 69, 71

framework Estructura conceptual y tecnológica de asistencia definida, normalmente, con artefactos o módulos concretos de software, que puede servir de base para la organización y desarrollo de software. 14–16

IATEXT Instituto Universitario de Análisis y Aplicaciones Textuales. 1, 9, 13, 15–17, 67, 68, 73, 76

IIS Internet Information Services. Servidor web desarrollado por Microsoft para sistemas Windows, compatible con HTTP, HTTP/2, HTTPS, FTP, FTPS, SMTP y NNTP. 15

inyección SQL Método de infiltración de código intruso que se vale de una vulnerabilidad informática, presente en una aplicación en el nivel de validación de las entradas, para realizar operaciones sobre una base de datos. 17

LLM Large Language Model. Modelo de lenguaje de propósito general, entrenado con ingentes conjuntos de datos, lo que les permite reconocer, traducir, predecir o generar texto u otro contenido. 3

MVVM Model-View-ViewModel o Modelo, Vista, Modelo de la vista. Arquitectura de software que tiene como principal objetivo la separación de la interfaz de usuario (vista) de la lógica de la aplicación (modelo), mediante el uso de servicios que actúan como puente entre ambos (modelos de la vista). 15, 41, 46

ORM Object-Relational Mapper, herramienta que mapea permite al programador manipular y consultar una base de datos mediante objetos y clases en un lenguaje de programación. 17, 35, 41, 67

test fixture Programa utilizado para configurar el estado del sistema y los datos de entrada necesarios para la ejecución de un test. 16

versión estable Versión que ha sido probada meticulosamente, con el fin de alcanzar la máxima fiabilidad posible y permitir su uso en producción. 17

WCF Windows Communication Foundation. Plataforma para desarrollo de aplicaciones orientadas a servicios. 15

WPF Windows Presentation Foundation. Framework de código abierto para desarrollo de aplicaciones de escritorio para Windows, lanzado en 2006 por Microsoft. 15

XAML Extensible Application Markup Language. Lenguaje declarativo basado en XML, diseñado para soportar las clases y métodos de .NET. Es utilizado habitualmente para definir la interfaz de usuario en múltiples herramientas de desarrollo de software dentro del entorno .NET, tales como WPF. viii, ix, 15, 47, 49, 51, 52