Base de datos de obras de compositores en IMSLP

Luis Sanjuán

marzo, 2026 <ver. 0.1>

Índice

Índice de figuras

Introducción[1]

Muchos de nosotros, los músicos profesionales, y seguramente cualquiera que se dedique a la investigación musical —por no hablar de nuestros propios alumnos— conocemos y visitamos frecuentemente el sitio web IMSLP, una biblioteca musical de partituras de dominio público de inestimable valor.

Además de los ficheros de partituras o de audio, IMSLP contiene información muy interesante acerca de cada una de las obras correspondientes. Por ejemplo, la página de IMSLP dedicada a la composición de Tárrega *Capricho Árabe*, aparte de los enlaces a los ficheros descargables de las partituras, contiene una sección de información general con los detalles que se muestran en la imagen siguiente:

Figura 1: Información sobre la obra Capricho Árabe en IMSLP

En mi trabajo diario he echado de menos disponer de algúna herramienta que me permitiera acceder rápidamente a este tipo de información para el conjunto de todas las composiciones en las que estoy interesado o de cuyas partituras dispongo, una que, por ejemplo, me permitiera filtrar inmediatamente todas los valses para dos guitarras de ciertos compositores y sólo de ellos; una sobre la que, además, pudiera plantearme la posibilidad de realizar ciertos análisis estadísticos, como —digamos— la frecuencia de los tipos de obras según períodos.

La propia interfaz de IMSLP permite realizar ciertas búsquedas básicas sobre el conjunto de todos sus registros, pero no consultas complejas o concretas como aquellas que sólo se refieran a los compositores de mi interés. Sobra decir que sobre su interfaz no es posible realizar ningún tipo de análisis estadístico; para ello, evidentemente, es necesario tener a nuestra disposición los datos mismos.

Pero si los datos son parte integrante e indisolublemente unida a las decenas de miles de páginas de IMSLP, parece improbable ser capaz de obtenerlos y guardarlos como un conjunto estructurado de datos, permanentemente disponible para el estudio, por ejemplo, como una hoja de cálculo o, técnicamente expresado, una base de datos. Se podría, por supuesto, copiar de la página de cada obra los datos de interés y guardarlos en algún fichero para luego elaborar con esos fragmentos una hoja de cálculo. Aunque posible, el tiempo y dedicación que tal procedimiento conllevaría se antoja demasiado elevado como para considerarlo una estrategia viable. Ahora bien, puesto que, en esencia, ese procedimiento es altamente mecánico, resulta concebible que sea un programa el encargado de ejecutar las acciones pertinentes para su realización.

El objetivo inicial de este trabajo es construir tal programa. Pero su objetivo principal es mostrar cómo se construye, qué modos de pensamiento entran en juego en la creación de un programa de esta clase o de cualquier programa en general. Sobra decir que por muy útil que sea la herramienta de software que podamos producir, mucho más provechoso es aprender, o empezar a aprender, el proceso mismo de su construcción, ya que lo aprendido de esa forma podrá aplicarse a cualquier situación o contexto; por no hablar de que los procesos de abstracción y diseño implicados poseen un valor intrínsico, el cual incluye también —y esto es especialmente relevante para cualquiera a quien el arte le concierne— un elemento estético de primer orden: todo programa bien construido es, en un sentido no meramente metafórico, un objeto artístico.

El segundo objetivo principal es mostrar cómo la inteligencia artificial en la forma actual de los modelos extensos de lenguaje (*Large Language Models*) puede servir de ayuda en la construcción de programas. Se mostrará, en particular, cómo un ayudante en la creación de código (*code agent*) es capaz, si se le proporcionan las instrucciones (*prompts*) adecuadas, de producir una interfaz web de usuario completamente funcional para nuestro código base en un tiempo increíblemente breve. Ciertamente, los modelos fundacionales (*foundation models*) y los asistentes de programación creados sobre ellos están demostrando una eficacia sin parangón en el mundo del desarrollo del software.

La exposición se dividirá en tantos capítulos como pasos en el proceso sistemático de construcción del programa. Esta sucesión de fases ha sido diseñada con el fin de resolver el problema *iterativamente*, desde la formulación más simple de las partes elementales del sistema hasta el refinamiento de tales formulaciones y la inclusión progresiva de funcionalidades nuevas, pasando por la generalización de las fases iniciales.

Asismismo, los ficheros de código fuente reflejan el proceso de desarrollo iterativo antes mencionado, de modo que los ficheros cuyo nombre contiene un número de iteración más alto acumulan y, en alguna ocasión, revisan la implementación de aquellos con menores números.

Cada sección corresponde *grosso modo* a cada uno de los ficheros de código fuente, de acuerdo con la siguiente secuenciación:

`imslpdb_01` Obtención de una página IMSLP.
`imslpdb_02` Análisis de un registro de página.
`imslpdb_03`, **`imslpdb_04`** Colecciones de registros.
`imslpdb_05` Ficheros JSON y registros genéricos.
`imslpdb_06`, **`imslpdb_07`** Registros nuevos y antiguos.
`imslpdb_08`, **`imslpdb_09`** Dataframes, ficheros CSV y Excel.
`imslpdb_10` Registros de páginas de compositores.
`imslpdb_11` Funciones envoltorio de lectura y escritura.
`imslpdb_12` Ensamblaje de las piezas.
`imslpdb_ui` La interfaz de usuario.
`imslpdb_main`, **`imslpdb`** El lanzador de la aplicación.

La descripción de cada sección será en cualquier caso muy sucinta. Si el lector es programador experimentado en Python podrá seguirla y entenderla por completo. Si es programador, pero no conoce Python, podrá comprender la secuencia lógica y todas las decisiones y soluciones propuestas, aunque la forma (la implementación) concreta de esas soluciones no le sea del todo transparente. Finalmente, si el lector no es programador podrá —espero— entender la esencia del proceso, aunque no pueda seguir los detalles punto por punto: la intelección de los modos de pensamiento involucrados en la creación de un programa informático es suficiente para captar la esencia de esa actividad y reconocer su valor y su significado y es el primer paso para iniciarse adecuadamente en el ámbito de la programación o, como resulta tan pertinente en el día de hoy, para poder trabajar en la creación de programas asistidos por IA.

Obtención de una página de IMSLP

El primer problema es el más simple de todos. Recordemos que el objetivo inicial es recuperar la información que se presenta en una página de IMSLP sobre una obra dada y hacerlo del modo más automático posible, esto es, sin intervención del usuario: sin que sea éste el que navegue hacia la página de su interés, copie de ella la información relevante y la guarde en un formato estructurado. Se trata pues de crear una herramienta de software que realice esa operación, algo así como un dispositivo que, dado el nombre de la página en cuestión (su título) obtiene su contenido y nos devuelve un registro de él si nada falla en el proceso. Si denominamos a este dispositivo por la acción que realiza, por ejemplo, `obten_página`, podríamos representarlo como una caja por la que entra el dato que introducimos, el título de la página, y de la que sale el resultado de la operación, un registro de la página, o bien nada si algo fallase en el proceso de obtención:

Este dispositivo se denomina *función* y es el componente básico de todo programa. Aquí la función `obten_página` (`fetch_page`) tiene como *argumento* el título de la página (`page_title`), una simple

Figura 2: Diagrama de la función `fetch_page`

cadena de texto (`str`) y produce como resultado un `registro de página` (`PageRecord`), o bien nada (`None`), si el proceso falla por alguna razón, por ejemplo, porque el servidor de IMSLP esté caído o no responda a nuestra solicitud.

En código, esta función se definiría de la siguiente forma:

```
def fetch_page(page_title: str) -> PageRecord | None:
```

En la parte inferior del diagrama de caja anterior aparece una breve descripción sobre su lógica interna: «Recupera [la página] del conjunto completo de páginas de IMSLP (`IMSLP_PAGES`)». Lo que no aparece es la manera exacta en que se implementa esa tarea, los engranajes de ese mecanismo que denominamos la función `fetch_page`.

Véamoslo ahora:

```
def fetch_page(page_title: str) -> PageRecord | None:
    """Fetch the given IMSLP page title and produce its PageRecord.
    Returns None if retrieval fails.
    Assume: page_title exists in IMSLP
    """
    try:
        imslp_page = IMSLP_PAGES[page_title]
    except (APIError, MaximumRetriesExceeded):
        print(f"{page_title} cannot be retrieved")
        return None
    return PageRecord(
        page_title = page_title,
        wikitext = imslp_page.text()
    )
```

En primer lugar, entre comillas triples, se muestra el propósito de la función y algunos otros datos relevantes, esta sección es la documentación interna de la función (*docstring* en jerga de Python). Tras la documentación se encuentra el cuerpo de la función: las expresiones y enunciados que hacen que la función desempeñe su tarea como lo hace. En este caso, el núcleo del procedimiento es seleccionar la página con el título dado de entre todas las páginas de IMSLP. Para poder hacerlo necesitamos una forma de interactuar con el servidor de IMSLP. Veamos esto último con algún detalle.

Cuando abrimos en nuestro navegador la página oficial de IMSLP se nos muestra la página principal, desde la cual podemos buscar la pieza que nos interesa. Pero navegar por el sito web de IMSLP no es la única forma de interactuar con su servidor. IMSLP es un sitio web basado en MediaWiki, un software para la creación de sitios web semejantes en su arquitectura a la Wikipedia, para la cual fue precisamente desarrollado MediaWiki. IMSLP se basa en la misma arquitectura. Lo interesante es que MediaWiki

proporciona una interfaz de programación de aplicaciones (API). A efectos prácticos esto significa que podemos, desde nuestros programas, interactuar con los sitios basados en MediaWiki, por ejemplo, para obtener páginas del sitio. Como programadores interesados en interactuar con un sitio como IMSLP lo habitual y recomendado es utilizar un *cliente* que entienda la API de MediaWiki y nos proporcione una sistema fácil de manejar, sin las complicaciones de un trato directo con la API de MediaWiki. Un cliente popular de ese tipo en Python es `mwclient`.

Las dos líneas siguientes definen mediante `mwclient` la interfaz principal que usaremos en nuestro programa:

```
IMSLP_SITE = mwclient.Site("imslp.org", path="/")
IMSLP_PAGES = IMSLP_SITE.pages
```

En concreto, `IMSLP_PAGES` nos proporciona el acceso a las páginas del sitio `IMSLP_SITE`, el cual apunta al sitio web de IMSLP. Recuperar una página de IMSLP a partir de su título resulta muy sencillo:

```
imslp_page = IMSLP_PAGES[page_title]
```

Puesto que la comunicación con el sitio web de IMSLP puede fallar, ya sea porque se produzca algún tipo de error interno en la API del MediaWiki de IMSLP o porque a consecuencia de un fallo en la red excedamos el número máximo de solicitudes de la página, es conveniente intentar primero la operación de recuperar la página y en caso de que esta falle hagamos que la función devuelva nada (`None`) en lugar de fallar sin resultado alguno:

```
    try:
        imslp_page = IMSLP_PAGES[page_title]
    except (APIError, MaximumRetriesExceeded):
        print(f"{page_title} cannot be retrieved")
        return None
```

Si, por el contario, como es previsible, la obtención de la página tiene éxito, nuestra función devuelve un *registro de página* (`PageRecord`), el cual consta del título de la página y de su contenido (`wikitext`):

```
    return PageRecord(
        page_title = page_title,
        wikitext = imslp_page.text()
    )
```

En el código comentado tenemos que hacer uso de elementos que no forman parte del lenguaje Python o de su biblioteca estándar[2].

En particular, para poder definir el sitio `IMSLP_SITE` tenemos que recurrir a un objeto `Site` de `mwclient` y para establecer qué clase de errores pueden producirse tenemos que utilizar las clases de errores `APIError` y `MaximumRetriesExceeded`, definidas en el módulo de errores de `mwclient`. Cuando en la construcción de un programa hemos de recurrir a herramientas que no vienen incorporados de fábrica por el lenguaje de programación, las incluimos explicitamente, en Python mediante enunciados `import`. En nuestro programa importamos `mwclient` en su totalidad y, de su módulo de errores, las clases `APIError` y `MaximumRetriesExceeded`:

```
import mwclient
from mwclient.errors import (
    APIError,
    MaximumRetriesExceeded
)
```

`PageRecord` es otro elemento del código que no forma parte ni de Python ni de ninguna biblioteca externa que podamos importar. Se trata de un recurso de programación que nosotros mismos definimos para diseñar adecuadamente nuestro programa. `PageRecord` no es otra cosa que una abstracción cuyo propósito es representar dentro del programa la información en bruto de la página de IMSLP de cada caso. Un

[2] La distribución de un lenguaje de programación, el software que descargamos cuando lo instalamos, incluye habitualmente un conjunto de recursos y herramientas, aparte de la propia definición del lenguaje, que permiten llevar a cabo las tareas más comunes. A este conjunto se le denomina convencionalmente *biblioteca estándar* (*standard library*).

registro de página (`PageRecord`) es un modelo —técnicamente hablando, una *clase*— que funciona como una plantilla, a partir de la cual se crean las instancias concretas de registros de página. Nuestra definición establece que todo registro de página, `PageRecord`, consta de un título de página (`page_title`), que es una cadena de texto (`str`), y del texto en el wiki de IMSLP que esa página contiene (`wikitext`), una cadena de texto igualmente:

```
class PageRecord(BaseModel):
    page_title: str
    wikitext: str
```

Por lo demás, para poder siquiera escribir estas definiciones de clase en la manera sucinta que acabamos de describir necesitamos de una herramienta como `pydantic`, de ahí que tengamos que importar `pydantic` —aquí, en concreto, la definición `BaseModel` sobre la que se base `PageRecord`—.

```
from pydantic import BaseModel
```

Para terminar, nótese la diferencia entre la definición de la clase `PageRecord` que acabamos de dar y una instancia concreta de esa clase, que es el valor que devuelve `fetch_page` si tiene éxito. La clase es el modelo a partir del cual se construyen las *instancias* de esa clase.

Una vez completado el código podemos usarlo. Una manera de hacerlo consiste en activar el entorno de Python e iniciar desde ahí una sesión en el intérprete de Python sobre la base del programa que acabamos de construir:

```
source .venv/bin/activate
ipython -i imslpdb_01.py
```

Una vez dentro del intérprete, ejecutamos la función `fetch_page` con el ejemplo concreto de la obra de Tárrega antes citada. Obtendremos lo siguiente:

```
fetch_page("Capricho Árabe (Tárrega, Francisco)")
```

Obtendremos el siguiente resultado:

```
PageRecord(
    page_title='Capricho árabe (Tárrega, Francisco)',
    wikitext=(
        '{{#fte:imslppage\n'
        '\n'
        '| *****AUDIO***** =\n'
        '\n'
        '| *****FILES***** =\n'
        '{{#fte:imslpfile\n'
        '|File Name 1=PMLP54762-Tarrega_-_Capricho_Arabe.pdf\n'
        '|File Description 1=Complete Score\n'
        '|Page Count 1=7\n'
        '|Editor=\n'
        '|Image Type=Normal Scan\n'
        '|Scanner={{SMB}}\n'
        '|Uploader=[[User:Jujimufu|Jujimufu]]\n'
        '|Date Submitted=2009/10/31\n'
        '|Publisher Information=Valencia: Antich y Tena, n.d. Plate: A.y.T. 357\n'
        '|Copyright=Public Domain\n'
        '|Misc. Notes=From the Boije collection\n'
        '}}\n'
        '{{#fte:imslpfile\n'
        '||File Name 1=PMLP54762-Tarrega_-_Capricho_Arabe_Serenata_for_Guitar.pdf\n'
        '|File Description 1=Complete Score\n'
        '|Page Count 1=5\n'
        '|Editor=\n'
        '|Image Type=Normal Scan\n'
        '|Scanner={{DKB}}\n'
        '|Uploader=[[User:Generoso|Generoso]]\n'
```

```
'|Date Submitted=2008/11/9\n'
'|Publisher Information=\n'
'|Reprint=Vienna: [[Schlesinger|Haslinger]], n.d. Plate 2087.\n'
'|Copyright=Public Domain\n'
"|Misc. Notes=From the Rischel & Birket-Smith's Collection of guitar music. \n"
'}}\n'
'{{#fte:imslpfile\n'
'|File Name 1=PMLP54762-RiBSms-233.pdf\n'
'|File Description 1=Complete Score\n'
'|Page Count 1=3\n'
'|Editor=G. Meier (copyist)\n'
'|Image Type=Normal Scan\n'
'|Scanner={{DKB}}\n'
'|Uploader=[[User:Schneidy|Schneidy]]\n'
'|Date Submitted=2013/1/22\n'
'|Publisher Information=Hamburg: Manuscript, 28 August 1914.\n'
'|Copyright=Public Domain\n'
"|Misc. Notes=From the Rischel & Birket-Smith's Collection of guitar music. \n"
'}}\n'
'{{#fte:imslpfile\n'
'|File Name 1=PMLP54762-Tarrega_F-Capricho_Arabe+mid.pdf\n'
'|File Description 1=Complete Score\n'
'|Editor={{LinkEd|Stefan|Apke}}\n'
'|Image Type=Typeset\n'
'|Scanner=editor\n'
'|Uploader=[[User:Stefan Apke|Stefan Apke]]\n'
'|Date Submitted=2016/6/24\n'
'|Publisher Information=Vlotho: Stefan Apke, 2016\n'
'|Copyright=Creative Commons Attribution-ShareAlike 4.0\n'
'|Misc. Notes=\n'
'}}\n'
'\n'
'===Arrangements and Transcriptions===\n'
'=====For Piano solo (Delucchi)=====\n'
'{{#fte:imslpfile\n'
'|File Name 1=PMLP54762-Capricho_Arabe_(Tarrega-Delucchi).pdf\n'
'|File Description 1=Complete Score\n'
'|Page Count 1=5\n'
'|Arranger={{LinkArr|Emanuele|Delucchi}}\n'
'|Image Type=Typeset\n'
'|Scanner=Arranger\n'
'|Uploader=[[User:Sofronio|Sofronio]]\n'
'|Date Submitted=2011/10/12\n'
'|Publisher Information=Emanuele Delucchi\n'
'|Copyright=Creative Commons Attribution 3.0\n'
'|Misc. Notes=\n'
'}}\n'
'=====For Guitar (Olson)=====\n'
'{{#fte:imslpfile\n'
'|File Name 1=PMLP54762-CaprichoArabe.pdf\n'
'|File Description 1=Complete Score\n'
'|Arranger={{LinkArr|J. J.|Olson|1947|}}\n'
'|Editor=\n'
'|Image Type=Typeset\n'
'|Scanner=arranger\n'
'|Uploader=[[User:Charanguero|Charanguero]]\n'
'|Date Submitted=2025/11/23\n'
'|Publisher Information=J. J. Olson, 2025.\n'
'|Copyright=Creative Commons Attribution 4.0\n'
'|Misc. Notes=A melodic reduction for sightreading.\n'
```

```
        '}}\n'
        '\n'
        '| *****WORK INFO*****\n'
        '\n'
        '|Work Title=Capricho árabe\n'
        '|Alternative Title=Serenata\n'
        '|Opus/Catalogue Number=\n'
        '|Number of Movements/Sections=1\n'
        '|Dedication=D. Tomás Breton\n'
        '|Year/Date of Composition=1892\n'
        '|Year of First Publication=\n'
        '|Librettist=\n'
        '|Language=\n'
        '|External Links=[[wikipedia:Capricho_árabe_(Tárrega)|Wikipedia article]]\n'
        '|Piece Style=Romantic\n'
        '|Instrumentation=guitar\n'
        '|Tags=capriccios ; serenades ; gtr\n'
        '\n'
        '| *****COMMENTS***** =\n'
        '\n'
        '\n'
        '\n'
        '| *****END OF TEMPLATE***** }}'
    ),
)
```

Análisis de un registro de página

El resultado de `fetch_page` es un registro de página como el que aparece en el párrafo anterior. Contiene, en efecto, la información que pretendíamos registrar: el título de la página y su contenido (wikitext). Este último, no obstante, es de una apariencia realmente intimidante. Se trata de una larga cadena de texto, donde los datos de información relevante parecen sepultados bajo un amasijo de secuencias textuales de oscuro significado. Aunque este es el verdadero contenido, puro y duro, de la página de IMSLP, lo que vemos cuando abrimos esa página en el navegador es mucho más legible. Que ello sea así es obra del programa que convierte este wikitexto en una página web. Para hacerlo debe analizar (*parse*) el texto bruto y convertirlo en HTML. Nuestra tarea es semejante, la diferencia es que el resultado que queremos no es HTML, sino un tipo de registro más simple y estructurado. Además, nos interesa, por lo que respecta a nuestro propósito actual, sólo una parte del contenido, la relativa a la información de la obra:

```
'| *****WORK INFO*****\n'
'\n'
'|Work Title=Capricho árabe\n'
'|Alternative Title=Serenata\n'
'|Opus/Catalogue Number=\n'
'|Number of Movements/Sections=1\n'
'|Dedication=D. Tomás Breton\n'
'|Year/Date of Composition=1892\n'
'|Year of First Publication=\n'
'|Librettist=\n'
'|Language=\n'
'|External Links=[[wikipedia:Capricho_árabe_(Tárrega)|Wikipedia article]]\n'
'|Piece Style=Romantic\n'
'|Instrumentation=guitar\n'
'|Tags=capriccios ; serenades ; gtr\n'
'\n'
'| *****COMMENTS***** =\n'
'\n'
```

```
'\n'
'\n'
'| *****END OF TEMPLATE***** }}'
```

Una forma legible que encajaría con nuestro objetivo sería parecida a la siguiente, donde el el wikitexto se ha transformado en un registro en formato JSON, que es uno de los formato más interesantes si queremos registrar información del modo más simple y claro posible:

```
{
  "page_title": "Capricho árabe (Tárrega, Francisco)",
  "work_title": "Capricho árabe",
  "alternative_title": "Serenata",
  "opus_catalogue_number": "",
  "number_of_movements_sections": "1",
  "dedication": "D. Tomás Breton",
  "year_date_of_composition": "1892",
  "year_of_first_publication": "",
  "librettist": "",
  "language": "",
  "external_links": "[[wikipedia:Capricho_árabe_(Tárrega)|Wikipedia article]]",
  "piece_style": "Romantic",
  "instrumentation": "guitar",
  "tags": "capriccios ; serenades ; gtr",
}
```

En esta sección veremos cómo transformar el registro de página y, en especial, su wikitexto en un tipo de dato que llamaremos *registro de composición* (`CompositionRecord`), una estructura de datos muy semejante y fácilmente convertible al formato JSON que acabamos de mostrar.

Empecemos esta vez por definir un `CompositionRecord`, dado que ya sabemos por donde empezar a hacerlo, después de haber definido `PageRecord`:

```
class CompositionRecord(BaseModel):
    page_title: str
    ...
```

Puesto que las instancias de `CompositionRecord` van a ser el elemento nuclear del programa, lo primero que interesa es consignar también en esta clase de registros el título de la página que contiene la información.

Ahora bien, la cuestión inmediata que se plantea es qué otros campos interesa añadir al registro. Es evidente que el nombre del compositor debería aparecer, algo que, curiosamente, no consta en el wikitexto relacionado con la información de la obra. Dentro de un momento estudiaremos cómo obtener ese dato. Por lo que respecta al resto de campos es natural estar tentado por la idea de registrarlos todos, esto es, crear un campo en `CompositionRecord` por cada uno de los campos de información del wikitexto: `Work Title`, `Alternative Title`, `Opus Catalogue Number`, etc. El problema de esta estrategia reside en el hecho de que no todas las páginas de IMSLP sobre composiciones contienen el mismo conjunto de campos. Hay, ciertamente, una serie de campos que aparecen en todas las páginas —al menos, en todas las que he visitado—; pero algunos campos sólo aparecen en algunas páginas. Como consecuencia, la estrategia más segura sería consignar los campos que aparecen en cada página, lo cual suena perfectamente razonable, pero con el evidente inconveniente de que no podemos saber por anticipado qué campos serán esos. ¿Hay alguna manera de dejar abierta la definición de `ComposerRecord` para que en el momento de creación de una instancia de esa clase se generen los campos extra que se encuentren en cada caso? Existe una manera y es la siguiente:

```
class CompositionRecord(BaseModel):
    page_title: str
    model_config = ConfigDict(extra="allow")
```

que requiere de `ConfigDict`, una primitiva de `pydantic`, que, en en cuanto tal, debemos importar:

```
from pydantic import (
    BaseModel,
    ConfigDict
}
```

Queda aún pendiente el campo del compositor. Éste puede computarse fácilmente a partir del título de la página correspondiente. En efecto, todas las páginas de composiciones de IMSLP siguen el mismo formato:

`título_de_la_obra (compositor)`

Podemos, pues, extraer ese nombre del título de la página. La función `composer` entresaca del título de la página la secuencia entre paréntesis y elimina esos paréntesis, produciendo como resultado el nombre del compositor. Esta función puede incluirse como *campo computado* dentro de la definición de la clase, que quedaría finalmente como sigue:

```
class CompositionRecord(BaseModel):
    page_title: str

    @computed_field
    def composer(self) -> str:
        m = re.search(r"\((.*?)\)$", self.page_title)
        if not m:
            return ""
        composer_label = m.group()
        return composer_label.removeprefix("(").removesuffix(")")

    model_config = ConfigDict(extra="allow")
```

Puesto que `computed_field` es un recurso de `pydantic` es necesario importarlo. También hay que importar el módulo `re` (*regular expressions*), dado que la función `composer` recurre a `re.search`.

```
import re

from pydantic import (
    BaseModel,
    ConfigDict,
    computed_field
}
```

Una vez que disponemos de un tipo adecuado de datos `CompositionRecord` para registrar el resultado de un análisis de un registro de página `PageRecord`, podemos definir la función que realice ese análisis (`parse_page_record`):

```
def parse_page_record(page_record: PageRecord) -> CompositionRecord:
    """Parse the given page_record as a CompositionRecord."""
    page_title = page_record.page_title
    work_info = parse_wikitext(page_record.wikitext)
    return CompositionRecord(
        page_title = page_title,
        **work_info
    )
```

Esta función toma como argumento un registro de página (`page_record`) y produce un registro de composición (`CompositionRecord`) que consta de un campo para el título de la página (`page_title`), de un campo que consigna el compositor, el cual —como hemos visto— se computa automáticamente a partir de ese título de página, y de tantos otros campos como el análisis del wikitexto sobre la información de la obra revele.

Por su parte, el análisis de la información de la obra que aparece en el wikitexto del registro de página es el propósito de la función de ayuda `parse_wikitext`, que se encarga de la parte más compleja del

trabajo, desenmarañar el wikitexto y producir un objeto estructurado muy semejante al formato JSON mostrado anteriormente, que en terminología de Python es un objeto de tipo `dict`:

```
def parse_wikitext(wikitext: str) -> dict:
    """Parse the given wikitext."""
    def normalize_parameter(p):
        return p.replace(" ", "_").replace("/", "_").lower()

    code = mwparserfromhell.parse(wikitext)
    templates = code.filter_templates()
    work_info: dict = {}
    for template in templates:
        if template.name.contains("imslppage"):
            parameters = template.params
            for parameter in parameters:
                if not parameter.lstrip().startswith("*"):
                    k = normalize_parameter(str(parameter.name))
                    v = str(parameter.value).strip()
                    work_info[k] = v
    return work_info
```

Esta función recurre para ejecutar su propósito a la función `parse` de la biblioteca externa `mwparserfromhell`, una herramienta especializada en el análisis de los wikitextos de las páginas de sitios basados en MediaWiki. La importamos pues:

```
import mwparserfromhell
```

La función `parse_wikitext` aplica, en primer lugar, la función `parse` de `mwparserfromhell`, la cual analiza apropiadamente el wikitexto completo; en segundo lugar, extrae la parte correspondiente a la información de la obra (que aparece como el contenido de una plantilla `imslppage` dentro del wikitexto); finalmente, selecciona de ahí las líneas que contienen campos de información y formatea el nombre de cada campo para que tenga un aspecto común *normalizado*: guiones en lugar de espacios, minúsculas, y sin espacios finales pendientes.

Si ejecutamos, desde un intérprete de Python, esta función sobre el registro de página de la obra de Tárrega del ejemplo anterior

```
capricho_arabe_page = fetch_page("Capricho Árabe (Tárrega, Francisco)")
parse_record(capricho_arabe_page)
```

obtenemos este resultado:

```
CompositionRecord(
    page_title='Capricho árabe (Tárrega, Francisco)',
    work_title='Capricho árabe',
    alternative_title='Serenata',
    opus_catalogue_number='',
    number_of_movements_sections='1',
    dedication='D. Tomás Breton',
    year_date_of_composition='1892',
    year_of_first_publication='',
    librettist='',
    language='',
    external_links='[[wikipedia:Capricho_árabe_(Tárrega)|Wikipedia article]]',
    piece_style='Romantic',
    instrumentation='guitar',
    tags='capriccios ; serenades ; gtr',
    composer='Tárrega, Francisco',
)
```

Colecciones de registros

Haber diseñado la función que obtiene una página de IMSLP (`fetch_page`) y la que analiza los datos brutos recuperados en una forma relevante y estructurada (`parse_page_record`) es el paso decisivo sobre el cual el resto del programa se construye. Este resto es ingeniería en el sentido más amplio, es decir, es la aplicación de principios generales de construcción de software al caso específico del problema que tratamos de resolver.

El primer principio es el de la *generalización*. Se trata de extender el diseño del programa para que sea capaz de manipular conjuntos o colecciones de objetos de los tipos modelados en las secciones anteriores. Estas colecciones son también clases de objetos: la colección de registros de partituras y la colección de registros de composiciones, `PageRecordCollection` y `CompositionRecordCollection`, respectivamente. Para definirlas como modelos en nuestro programa haremos uso de un tipo primitivo de `pydantic` específicamente diseñado para representar colecciones de objetos, `RootModel`:

```
from pydantic import (
    BaseModel,
    RootModel,
    ConfigDict,
    computed_field
)
```

En un `RootModel` la colección de objetos propiamente dicha es el valor del campo `root`. Pero en la definición de una clase basada en `RootModel`, este campo es implícito y no es necesario declararlo si especificamos como parámetro el tipo de datos de la colección que queremos modelar:

```
class PageRecordCollection(RootModel[list[PageRecord]]):
    pass
```

```
class CompositionRecordCollection(RootModel[list[CompositionRecord]]):
    pass
```

Así pues, un colección de registros de páginas queda definida sencillamente como una lista de registros de páginas y una colección de registros de composiciones como una lista de registros de composiciones[3].

Podemos ahora definir funciones que procesen y produzcan las colecciones de registros que acabamos de modelar.

El caso más sencillo de implementar, dadas las características del programa, de una función de este tipo es la función `parse_page_record_collection`, que toma una colección de registros de partituras y produce una colección de registros de composiciones, resultante de analizar cada uno de los registros de partituras de la colección dada:

```
def parse_page_record_collection(
    page_record_collection: PageRecordCollection
) -> CompositionRecordCollection:
    """Parse records in the given collection."""
    composition_records = []
    for page_record in page_record_collection.root:
        composition_record = parse_page_record(page_record)
        composition_records.append(composition_record)
    return CompositionRecordCollection(composition_records)
```

Nótese que para acceder a cada registro de la colección de registros de páginas suministrada como argumento de la función, hemos de hacerlo a través del campo `root` de la colección, que, como se comentó anteriormente, tiene como valor la lista de registros de páginas propiamente dicha.

Para los puristas en Python, una versión más sucinta e idiomática, que usa a *list comprehension* sería:

```
def parse_page_record_collection(
    page_record_collection: PageRecordCollection
```

[3] El enunciado `pass` es obligatorio, pero no tiene ningún significado, más allá de completar sintácticamente la definición.

```
) -> CompositionRecordCollection:
    """Parse records in the given collection."""
    return CompositionRecordCollection(
        [parse_page_record(pr) for pr in page_record_collection.root]
    )
```

Antes de poner a prueba esta función, pasemos a la función `fetch_pages`. La implementación obvia es equivalente *mutatis mutandis* a la que acabamos de implementar. El único añadido necesario es que aquí hemos de asegurarnos de no incluir registros de página nulos, esos que se producen si algún fallo ocurre en la comunicación con el servidor[4]:

```
def fetch_pages(page_titles: list[str]) -> PageRecordCollection:
    """Fetch the IMSLP pages with given titles."""
    page_records = []
    for page_title in page_titles:
        page_record = fetch_page(page_title)
        if page_record:
            page_records.append(page_record)
    return PageRecordCollection(page_records)
```

Ahora bien, esta versión se enfrenta a un problema de educación. Sí, tal cual está escrita, es maleducada, hasta el punto de que podemos acabar siendo bloqueados por el servidor de IMSLP a causa de nuestra absoluta falta de tacto, incluso de respeto.

Entender el porqué y el cómo ser buenos ciudadanos en nuestras interacciones en la red es de una importancia capital para cualquier programador cuyas aplicaciones hacen uso extensivo de los recursos de servidores remotos.

Estos servidores, que habitualmente proporcionan toda clase de recursos a una gran cantidad de usuarios simultáneamente, requieren para su correcto funcionamiento que sus usuarios se comporten cívicamente. Y esto, en primer lugar, significa no saturar al servidor con un bombardeo continuo de solicitudes. Un usuario humano no puede en principio saturar el servidor, su velocidad de realizar solicitudes es limitada, pero un robot, esto es, otro ordenador, puede realizar una solicitud tras otra a intervalos de milisegundos. De ahí que estos servidores suelan distribuir el fichero `robots.txt`, donde se especifican algunas recomendaciones básicas para máquinas civilizadas. El `robots.txt` de IMSLP reza como sigue:

```
User-agent: MyriadBot
Disallow:

User-agent: *
Disallow: /index.php
Disallow: /wiki/Special:
Disallow: /images/
Disallow: /imglnks/
Disallow: /wiki/File:
Disallow: /wiki/Image:
Disallow: /instruments
Disallow: /works
Disallow: /work/
Disallow: /instrument/
Disallow: /library/
```

[4] Una sintaxis alternativa a las más prolija presentada en el texto principal de `fetch_pages`, una que, a buen seguro, sería del agrado de puristas en Python moderno, usa también *list comprehension* con *variable assignment* (*operador Walrus*):

```
def fetch_pages(page_titles: list[str]) -> PageRecordCollection:
    """Fetch the IMSLP pages with given titles."""
    return PageRecordCollection(
        [pr for pt in page_titles if (pr := fetch_page(pt))]
    )
```

```
Sitemap: https://imslp.org/sitemap/sitemap-index-imslp_wiki.xml
Crawl-delay: 2
```

La línea que ahora nos interesa es la última. Significa que se debería esperar dos segundos al menos entre una solicitud y la siguiente. De no hacerlo, no sólo nos presentamos como bárbaros egoístas sino que nuestra IP puede llegar a ser bloqueada por el sitio web.

El módulo `time` contiene utilidades que pueden aplicarse en el caso de nuestra función `fetch_pages` con el fin de limitar la velocidad de las solicitudes de páginas a IMSLP. Es más, podemos idear un procedimiento para que ese tiempo de espera entre una solicitud y otra sea aleatoriamente elegido entre un número mínimo y máximo de segundos, lo cual hace más probable que nuestras comunicaciones pasen inadvertidas —otro signo de educación es no dar la nota—:

```
import random
import time

MIN_DELAY = 2  # seconds
MAX_DELAY = 10 # seconds

def wait_for_fetching(min_delay: int, max_delay: int) -> float:
    """Produce a random number of seconds between min and max delay."""
    delay = round(random.uniform(min_delay, max_delay), 2)
    print(f"Waiting {delay} seconds ...")
    time.sleep(delay)
    return delay
```

Provistos de esta herramienta, modificamos la función `fetch_pages` para que entre una solicitud y otra transcurra ese retraso aleatoriamente computado. De paso, hacemos que la propia función imprima mensajes sobre el proceso y su duración.

```
def fetch_pages(page_titles: list[str]) -> PageRecordCollection:
    """Fetch the IMSLP pages with given titles."""
    page_records = []

    time_start = time.perf_counter()
    for page_title in page_titles:
        wait_for_fetching(MIN_DELAY, MAX_DELAY)
        page_record = fetch_page(page_title)
        if page_record:
            print(f"'{page_title}' fetched")
            page_records.append(page_record)
    time_end = time.perf_counter()

    time_elapsed = time_end - time_start
    print(f"{len(page_records)} fetched in {time_elapsed:.2f} seconds")
    return PageRecordCollection(composition_records)
```

Educadamente moderada ahora nuestra ansiedad por recuperar las páginas de IMSLP, aún queda en la implementación de esta función un importante inconveniente, a saber, nos fuerza a conocer de antemano el título exacto de todas las páginas en las que estamos interesados para poder solicitarlas.

La mejor manera de evitar esta grave contrariedad es obtener todas las páginas que pertenecen a un determinado grupo de páginas de nuestro interés. De esta forma solo necesitaríamos conocer por anticipado el título de ese conjunto y no de cada uno de sus elementos. Los sitios basados en MediaWiki se organizan en torno a *categorías*, que no son otra cosa que páginas especiales que contienen a su vez distintas páginas singulares. En IMSLP, hay categorías por compositor, estilo, instrumentación, etc. Así pues, `fetch_pages` puede mejor renombrarse como `fetch_pages_by_category` y tomar como argumento, en lugar de una lista de títulos de página, un título de categoría:

```
IMSLP_CATEGORIES = IMSLP_SITE.categories
```

```python
def fetch_pages_by_category(category_title: str) -> PageRecordCollection:
    """Fetch IMSLP pages of the given category_title.
    Assume: the corresponding category exists in IMSLP."""
    category = IMSLP_CATEGORIES[category_title]
    page_records: list[PageRecord] = []

    time_start = time.perf_counter()
    for page in category:
        page_title = page.page_title
        wait_for_fetching(MIN_DELAY, MAX_DELAY)
        page_record = fetch_page(page_title)
        if page_record:
            print(f"'{page_title}' fetched")
            page_records.append(page_record)
    time_end = time.perf_counter()

    time_elapsed = time_end - time_start
    print(f"{len(page_records)} fetched in {time_elapsed:.2f} seconds")
    return PageRecordCollection(page_records)
```

`mwclient` posee una interfaz a las categorías de un sitio web. Por eso definimos `IMSLP_CATEGORIES` que puede recibir el título de una categoría, de la misma manera que `IMSLP_PAGES` lo hace con el título de una página. De ahí se obtiene luego mediante un bucle cada una de las páginas miembros de la categoría en cuestión.

Provistos de ambas funciones, `fetch_pages_by_category` —que produce una colección de registros de las páginas pertenecientes a un título de categoría dado— y `parse_page_record_collection` —que produce una colección de registros de composiciones a partir de una colección de registros de página y que, por tanto, puede analizar el resultado de `fetch_pages_by_category`—, podemos ver cómo funcionan. Digamos, por ejemplo, que queremos obtener un conjunto de registros de las composiciones de Emilia Giuliani-Guglielmi. Podemos hacerlo recuperando en primer lugar las páginas en IMSLP que corresponden a la categoría de esa compositora y pasando el resultado a nuestro analizador de registros de páginas:

```python
emilia_giuliani_pages = fetch_pages_by_category("Giuliani-Guglielmi, Emilia")
parse_page_record_collection(emilia_giuliani_pages)
```

La ejecución de la primera línea produce, además, de recuperar las páginas requeridas, los siguientes mensajes[5]

```
Waiting 9.61 seconds ...
'6 Preludi, Op.46 (Giuliani-Guglielmi, Emilia)' fetched
Waiting 3.22 seconds ...
'Variations on 'Ah perchè non posso odiarti', Op.3 (Giuliani-Guglielmi, Emilia)' fetched
2 fetched in 13.83 seconds
```

La ejecución de la segunda línea produce el siguiente resultado, que es el que justamente esperamos, una colección de registros de composiciones:

```
CompositionRecordCollection(
    root=[
        CompositionRecord(
            page_title='6 Preludi, Op.46 (Giuliani-Guglielmi, Emilia)',
            work_title='6 preludi',
            alternative_title='Sei preludi per chitarra',
            opus_catalogue_number='Op.46',
            key='',
            movements_header='6 pieces',
            number_of_movements_sections='6 pieces',
            average_duration='',
```

[5] las líneas de texto muy largas se abrevian de aquí en adelante para que quepan bien en la página del documento, aunque aparezcan como una sola línea en la salida del programa:

```
            dedication='{{LinkDed|Luigi|Moretti}}',
            first_performance='',
            year_date_of_composition='',
            year_of_first_publication='',
            librettist='',
            language='',
            piece_style='Romantic',
            instrumentation='guitar',
            tags='preludes ; gtr',
            composer='Giuliani-Guglielmi, Emilia',
        ),
        CompositionRecord(
            page_title="Variations ..., Op.3 (Giuliani-Guglielmi, Emilia)",
            work_title="Varizioni ... del Mo. Bellini",
            alternative_title='',
            opus_catalogue_number='Op.3',
            key='{{key|C}}',
            movements_header='Introduction, theme, 5 variations',
            number_of_movements_sections='Introduction, theme, 5 variations',
            average_duration='',
            dedication='',
            first_performance='',
            year_date_of_composition='',
            year_of_first_publication='',
            librettist='',
            language='',
            piece_style='Romantic',
            related_works='{{OpDerivs|Lasonnambula}}',
            instrumentation='guitar',
            tags='variations ; gtr',
            composer='Giuliani-Guglielmi, Emilia',
        ),
    ],
)
```

Ficheros JSON y registros genéricos

De poco servirían estas colecciones de registros si no somos capaces de guardarlas en el disco duro y de cargarlas cuando las necesitemos.

En cualquier programa que manipula datos la cuestión de asegurar una *persistencia* de esos datos entre las diferentes sesiones de ejecución del programa es de vital importancia.

En el caso que nos ocupa lo primero que hay que decidir es qué conjunto de datos nos interesa preservar. Es obvio que el único conjunto de datos de interés es el que contiene los registros de composiciones. La colección de registros de páginas es, según parece a todas luces, un objeto intermedio que sirve para generar los registros de composiciones. Aunque, sin duda, esto es así, conviene preservar también el conjunto de registros de páginas. La razón es la siguiente. Los registros de composiciones contienen la información estructurada sobre la obra que consta en el wikitexto de los registros de páginas correspondientes. Sin embargo, no es esta toda la información que suministran tales registros. En particular, el wikitexto de los registros de páginas recoge información sobre los ficheros de partituras o audios relacionados con la obra en cuestión. Si en el futuro queremos recuperar esa información y no disponemos de un fichero donde se haya preservado, nos veremos obligados a solicitarla de nuevo al servidor de IMSLP, lo cual es, naturalmente, un gasto innecesario de recursos, de tiempo y un abuso del servidor.

Decidido qué preservar, queda determinar cómo preservarlo. En concreto, qué clase de formato elegir como medio de preservación. El formato JSON, del que ya hablamos previamente y del que se mostró un ejemplo, es especialmente indicado. Un fichero JSON de registros es, propiamente hablando, una

base de datos. Así pues, acabaríamos con dos bases o conjuntos de datos, una de registros de páginas de composiciones y otra de registros de composiciones:

```
IMSLP_COMPOSITION_PAGE_RECORDS_JSON = "data/imslp_composition_page_records.json"
IMSLP_COMPOSITION_RECORDS_JSON = "data/imslp_composition_records.json"
```

El siguiente paso es diseñar funciones que carguen los ficheros JSON como objetos de colecciones de registros de páginas o composiciones y que guarden esas colecciones como ficheros JSON.

Es decir deberíamos tener estas cuatro funciones [se expone sólo la firma (*signature*) de cada función]:

```
def load_page_record_collection(filename: str) -> PageRecordCollection:

def load_composition_record_collection(
    filename: str
) -> CompositionRecordCollection:

def save_page_record_collection(
    page_record_collection: PageRecordCollection,
    filename: str
) -> None:

def save_composition_record_collection(
    composition_record_collection: CompositionRecordCollection,
    filename: str
) -> None:
```

Las dos primeras funciones cargan los datos contenidos en el fichero JSON cuyo nombre se proporciona como argumento y producen, respectivamente, una colección de registros de páginas y una colección de registros de composiciones. Las dos últimas funciones guardan en disco como fichero JSON las colecciones de registros, de páginas y composiciones, que se pasen como argumento. Nótese que estas funciones `save` no producen un resultado, o, dicho de otro modo, devuelven `None` y sólo tienen sentido por su efecto (*side effect*): escribir un fichero en el disco.

A la luz de esta breve descripción es de esperar que la implementación de las funciones que implican una clase de registros puede ser prácticamente la misma o idéntica a la que implica la otra clase de registros. Es pues recomendable que en lugar de duplicar código en cada pareja de funciones, consideremos la posibilidad de diseñar un modelo de objeto para un colección genérica de registros.

Este punto es crucial y forma parte de los principios básicos de construcción de software: la *abstracción* a partir de casos semejantes. En el caso actual la creación de un modelo *genérico* de registro, que llamaremos `ImslpRecord` cuyas especies particulares (subclases) serían `PageRecord` y `CompositionRecord`. Asimismo, definimos una colección genérica de registros `RecordCollection` que utiliza una variable de tipo `T`, la cual nos permite representar en las clases concretas colecciones de registros de distinta clase:

```
class ImslpRecord(BaseModel):
    page_title: str

class PageRecord(ImslpRecord):
    wikitext: str

class CompositionRecord(ImslpRecord):
    @computed_field
    def composer(self) -> str:
        m = re.search(r"\((.*?)\)$", self.page_title)
        if not m:
            return ""
        composer_label = m.group()
        return composer_label.removeprefix("(").removesuffix(")")

    model_config = ConfigDict(extra="allow")
```

```python
class RecordCollection[T: BaseModel](RootModel[list[T]]):
    pass

class PageRecordCollection(RecordCollection[PageRecord]):
    pass

class CompositionRecordCollection(RecordCollection[CompositionRecord]):
    pass
```

Con estos modelos de representación a nuestra disposición ajustamos las definiciones de las funciones `load` and `save` propuestas antes, que ahora hacen uso de los nuevos tipos. Añadimos, para empezar, una función abstracta `load_record_collection` que nos permitirá simplificar la implementación de las funciones `load_page_record_collection` y `load_composition_record_collection`. En segundo lugar, creamos una función `save_record_collection` que vale tanto para guardar registros de páginas como registros de composiciones:

```python
def load_record_collection(filename: str, cls: type[RecordCollection]):

def load_page_record_collection(filename: str) -> PageRecordCollection:

def load_composition_record_collection(
    filename: str
) -> CompositionRecordCollection:

def save_record_collection(
    record_collection: RecordCollection,
    filename: str
) -> None:
```

La implementación es relativamente sencilla. Recurre a los métodos de modelos `pydantic`: `model_validate_json`, que produce un modelo a partir de su representación JSON, y `model_dump_json`, que produce una representación JSON del modelo. De hecho, es la disponibilidad de estas funciones una de las razones fundamentales de utilizar `pydantic` en la representación de las clases de objetos de nuestro programa:

```python
from pathlib import Path

def load_record_collection(filename: str, cls: type[RecordCollection]):
    """Load the given JSON filename as a RecordCollection of the given cls."""
    try:
        json_string = Path(filename).read_text()
    except FileNotFoundError:
        print(f"'{filename}' does not exist. Starting from scratch...")
        json_string = "[]"
    return cls.model_validate_json(json_string)

def load_page_record_collection(filename: str) -> PageRecordCollection:
    """Load the given JSON filename as a PageRecordCollection."""
    return load_record_collection(filename, PageRecordCollection)

def load_composition_record_collection(
    filename: str
) -> CompositionRecordCollection:
    return load_record_collection(filename, CompositionRecordCollection)

def save_record_collection(
    record_collection: RecordCollection,
    filename: str
) -> None:
    """Save the given record collection as a JSON filename."""
```

```
    def backup(src: Path) -> None:
        """Backup path"""
        dest = src.with_suffix(".json.bak")
        dest.write_text(src.read_text())
        print(f"'{src}' backed up as '{dest}'")

    json_string = record_collection.model_dump_json()
    json_path = Path(filename)
    if json_path.exists():
        backup(json_path)
    json_path.write_text(json_string)
    print(f"Records saved in '{json_path}'")
```

La función `load_record_collection` lee un fichero JSON de registros y produce una colección de registros, ya sea `PageRecordCollection` o `CompositionRecordCollection` a partir de ese fichero. En caso de que el fichero no exista produce una colección vacía de registros. Las funciones `load_page_record_collection` y `load_composition_record_collection` sirven para concretar qué clase de colección de registros se produce. Por su parte, la función `save_record_collection` convierte la colección de registros dada en un cadena JSON y la escribe en el disco, no sin antes hacer una copia de seguridad del fichero JSON que contiene esa misma colección. Ambas funciones, la función que carga los registros y la que los escribe en el disco hacen uso de las funciones `read_text` y `write_text` de `Path`, una clase proporcionada por el módulo `pathlib`.

Registros nuevos y antiguos

El siguiente paso fundamental en la construcción del programa tiene que ver con la *eficiencia*. De poco serviría tener almacenados los registros de páginas y de composiciones si cada vez que ejecutamos la aplicación se tienen que reconstruir enteramente, en lugar de añadirse a las colecciones de registros únicamente aquellos que aún no han sido registrados. En el caso de la obtención de las páginas de IMSLP la situación es aún más grave, puesto que, sin la eficiencia necesaria, cada vez que ejecutásemos la aplicación volveríamos a realizar las mismas solicitudes al servidor de IMSLP una y otra vez, de nuevo mala educación, pero esta vez elevada al cuadrado.

Es obligatorio, pues, refinar, las funciones que se encargan de obtener los registros de página y de composiciones para que tengan en cuenta los datos que ya existen en nuestros ficheros.

En términos generales la estrategia para lograr este objetivo es la siguiente:

1. Se pasan como argumento de la función que refinar la colección de datos preexistentes que convenga en cada caso.
2. En cada llamada a una función de procesamiento de un dato simple se inspecciona si ese dato ya existe en la colección dada. En caso de que exista, no se hace nada; en el caso contrario, el dato se procesa y el resultado del procesamiento se añade a una lista de nuevos registros.
3. Finalizado el procesamiento de todos los datos simples, se actualiza la colección preexistente para que incluya los nuevos datos.

Los pasos primero y último son triviales, como apreciaremos en un momento. El segundo paso, sin embargo, requiere una reflexión más profunda. ¿Cómo determinamos si el dato que estamos procesando está contenido ya en la colección actual? Puesto que cada registro consta de un identificador único, el título de la página registrada, determinar si un dato está ya registrado equivale a determinar si su título de página figura entre los títulos de las páginas de la colección actual.

Afortunadamente, y gracias al proceso de abstracción acometido en la sección anterior, podemos refinar la definición de `RecordCollection` para que proporcione una lista de los títulos de página de sus registros, independientemente de si se trata de registros de páginas o de composiciones. Podemos además añadir a su definición una función que determine si, dado un título de página, el registro que corresponde a ese título está contenido o no en la colección.

```python
class RecordCollection[T: ImslpRecord](RootModel[list[T]]):
    @computed_field
    @property
    def page_titles(self) -> list[str]:
        return [record.page_title for record in self.root]

    def contains(self, page_title: str) -> bool:
        return page_title in self.page_titles
```

En esta nueva definición de `RecordCollection`, `page_titles` se convierte en un campo computado cuyo valor es la lista de todos los títulos de la colección. Por su parte, la función `contains`, técnicamente hablando un *método* de `RecordCollection`, determina si, dado un título de página, éste forma parte de la colección.

Provistos de esta remozada definición de `RecordCollection` podemos modificar las funciones `fetch_pages_by_category` y `parse_page_record_collection` para que tengan en cuenta las colecciones actuales de registros de páginas y composiciones, respectivamente:

```python
def fetch_pages_by_category(
    category_title: str,
    page_record_collection: PageRecordCollection
) -> PageRecordCollection:
    """Fetch IMSLP pages of the given category_title
    if not already contained in page_record_collection.
    Assume: the corresponding category exists in IMSLP.
    """
    category = IMSLP_CATEGORIES[category_title]
    new_page_records = []

    time_start = time.perf_counter()
    for page in category:
        page_title = page.page_title
        if not page_record_collection.contains(page_title):
            wait_for_fetching(MIN_DELAY, MAX_DELAY)
            page_record = fetch_page(page_title)
            if page_record:
                print(f"'{page_title}' fetched")
                new_page_records.append(page_record)
        else:
            print(f"'{page_title}' is already registered.")
    time_end = time.perf_counter()

    time_elapsed = time_end - time_start
    print(f"{len(new_page_records)} fetched in {time_elapsed:.2f} seconds")
    updated_records = page_record_collection.root + new_page_records
    return PageRecordCollection(updated_records)
```

`fetch_page_by_category` toma ahora, aparte del título de la categoría, una colección de registros de página, la colección preexistente. Por cada página de la categoría dada, recupera el título y comprueba que ese título no consta entre los de la colección dada de registros de página. En caso de que no exista, genera una registro de página de ese título y lo añade a lista de nuevos registros de página; en caso contrario, informa de que la página ya existe. Una vez han sido procesadas todas las páginas de la categoría, actualiza la lista de registros dada añadiéndole la lista de nuevos registros. El resultado, la lista de registros actualizada, se devuelve como colección de registros de página.

```python
def parse_page_record_collection(
    page_record_collection: PageRecordCollection,
    composition_record_collection: CompositionRecordCollection
) -> CompositionRecordCollection:
    """Parse records in the given collection."""
```

```
    new_composition_records = []
    for page_record in page_record_collection.root:
        page_title = page_record.page_title
        if not composition_record_collection.contains(page_title):
            composition_record = parse_page_record(page_record)
            new_composition_records.append(composition_record)
    updated_records = (
        composition_record_collection.root + new_composition_records
    )
    return CompositionRecordCollection(updated_records)
```

`parse_page_record_collection` sigue la misma estrategia que `fetch_pages_by_category`. En este caso, además de la colección de registros de página que hay que analizar, se pasa una colección de registros de composiciones, la colección preexistente y se añaden a ella los registros de composición de las páginas no analizadas previamente. En este caso, la función actúa silenciosamente; esto es, no notifica si un registro de composición existe ya o no. En el uso normal de esta aplicación, que cuenta con que `fetch_pages_by_category` será ejecutada inmediatamente antes, una doble notificación resultaría redundante.

Dataframes, ficheros CSV y Excel

En la ciencia de los datos (*Data Science*), tan prominente en los últimos tiempos, la representación en forma de tabla bidimensional con columnas y filas a las que se asocian nombres u etiquetas se ha convertido en un estándar *de facto*. Una de estas estructuras, universalmente soportada por muchas de las bibliotecas especializadas en el análisis de datos dentro de los distintos lenguajes de programación es el *dataframe*. Más conocidos para el usuario no profesional son los ficheros CSV y las tablas de Excel.

El objetivo de esta sección es definir funciones que permitan *convertir* nuestras representaciones de registros en *dataframe*s, y de ahí en ficheros CSV y Excel.

Una de las bibliotecas de Python más populares últimamente en la ciencia de datos es *Polars* que, por supuesto, tiene al *dataframe* como clase central. La función `convert_to_dataframe` convierte la colección de registros de composiciones en un *dataframe* de *Polars*:

```
import polars as pl

IMSLP_COMPOSITION_URL_SCHEMA = "https://imslp.org/wiki/<page_title>"

def convert_to_dataframe(
    composition_record_collection: CompositionRecordCollection,
    cols: list[str] | None = None,
    with_anchors: bool = False
) -> pl.DataFrame:
    """Convert the given composition_record_collection to a Polars dataframe.
    If cols are given, produces a dataframe with those cols.
    If with_anchors, wrap the URL of the IMSLP page in an anchor HTML tag
    """
    def page_title_to_url(page_title):
        """Produce the URL in IMSLP of the given page_title."""
        return re.sub("<page_title>", page_title, IMSLP_COMPOSITION_URL_SCHEMA)

    def url_to_anchor(url):
        """Produce an HTML anchor for the given url."""
        return f'<a href="{url}" target="_blank">IMSLP page</a>'

    # Handle empty collection case
    if not composition_record_collection.root:
        if cols:
            return pl.DataFrame({col: pl.Series([], dtype=pl.Object) for col in cols})
```

```
        else:
            return pl.DataFrame()

    df_raw = pl.from_dicts(composition_record_collection.model_dump())
    df = df_raw.with_columns(
        imslp_page=pl.col("page_title").map_elements(
            lambda x: page_title_to_url(x)
        )
    )

    if with_anchors:
        df = df.with_columns(
        imslp_page=pl.col("imslp_page").map_elements(
            lambda x: url_to_anchor(x)
        )
    )

    if cols:
        df = df.select(cols)
    return df
```

La función toma como argumento una colección de registros de composiciones y produce como resultado un *dataframe*.

En primer lugar, trata el caso de que la colección de registros esté vacía. Cuando es así, genera un *dataframe* vacío con las columnas indicadas en el argumento opcional `cols`, o uno sin columnas, si `cols` no ha sido especificado.

En el caso de que la colección contega registros, genera un *dataframe* donde los valores de la columna de título `page_title` se transforman —por medio de la función interna `page_title_to_url`— en una dirección URL que sigue este esquema:

```
IMSLP_COMPOSITION_URL_SCHEMA = "https://imslp.org/wiki/<page_title>"
```

Por ejemplo, la página IMSLP con título `Capricho_Árabe (Tárrega, Francisco)` se convierte en:

`https://imslp.org/wiki/Capricho Árabe (Tárrega, Francisco)`

que es la dirección de la página en IMSLP relativa a esa obra de Tárrega.

Si el argumento opcional `with_anchors`, que puede recibir los valores `True` or `False` (por defecto, `False`), se pone a `True` entonces esas URLs se convierten en etiquetas HTML de enlaces que se pueden seguir pinchando sobre ellos.

Asi, la URL del ejemplo anterior quedaría convertida en el siguiente elemento HTML:

```
<a href="https://imslp.org/wiki/Capricho Árabe (Tárrega, Francisco)"
   target="_blank">IMSLP page</a>
```

Finalmente, si al argumento opcional `cols` se le da el valor de una lista de columnas —por defecto, no tiene ningún valor—, entonces se genera un subconjunto del *dataframe* que consta únicamente de las columnas indicadas. De este modo, si establezco las columnas siguientes como valor de `cols`,

```
BASIC_COLS = [
    "composer",
    "work_title",
    "year_of_first_publication",
    "piece_style",
    "instrumentation",
    "imslp_page",
    "tags"
]
```

obtendré un *dataframe* que contiene únicamente esas columnas.

Veamos un ejemplo, tomando como fuente de datos el fichero examples/imslp_composition_page_records.json:

```
compositions = (
    load_composition_record_collection("examples/imslp_composition_records.json")
)
df = convert_to_dataframe(compositions, cols=BASIC_COLS, with_anchors=True)
df.show(20, ascii_tables=True, tbl_cols=3)
```

```
+---------------------------+-----------------------------------+-----+------------------+
| composer                  | work_title                        | ... | tags             |
| ---                       | ---                               |     | ---              |
| str                       | str                               |     | str              |
+===========================================================================================+
| Giuliani-Guglielmi, Emilia | 6 preludi                        | ... | preludes ; gtr   |
| Giuliani-Guglielmi, Emilia | Varizioni sul tema 'Ah perchè ... | ... | variations ; gtr |
| Paulian, Athénaïs          | Airs et Variations               | ... | variations ; gtr |
| Sor, Carlos                | Minuet pour la Guitare           | ... | minuets ; gtr    |
| Hayden, C. V.              | Cambridge Gavotte                | ... | gavottes ; gtr   |
| Hayden, C. V.              | Constellation March              | ... | marches ; gtr    |
| Hayden, C. V.              | The Darkies' Dance               | ... | dances ; gtr     |
| Hayden, C. V.              | Delight Polka                    | ... | polkas ; gtr     |
| Hayden, C. V.              | Elfin Gavotte                    | ... | gavottes ; gtr   |
| Hayden, C. V.              | Elsie Mazurka                    | ... | mazurkas ; gtr   |
| Hayden, C. V.              | Gavotte "My Love"                | ... | gavottes ; gtr   |
| Hayden, C. V.              | Happy Heart Polka                | ... | polkas ; gtr     |
| Hayden, C. V.              | Happy Memories Waltz             | ... | waltzes ; gtr    |
| Hayden, C. V.              | In Dreamland Waltzes             | ... | waltzes ; gtr    |
| Hayden, C. V.              | Maidella Waltz                   | ... | waltzes ; gtr    |
| Hayden, C. V.              | March Brillant                   | ... | marches ; gtr    |
| Hayden, C. V.              | Reverie                          | ... | pieces ; gtr     |
| Hayden, C. V.              | Romance of Venice                | ... | waltzes ; gtr    |
| Hayden, C. V.              | That is Love                     | ... | ecossaises ; gtr |
| Hayden, C. V.              | Unique Gavotte                   | ... | gavottes ; gtr   |
+---------------------------+-----------------------------------+-----+------------------+
```

Las funciones para convertir este *dataframe* es un fichero CSV o Excel son muy fáciles de implementar —cortesía de *Polars*—:

```
def save_as_csv(dataframe: pl.DataFrame, filename: str) -> None:
    """Save the given dataframe as a CSV filename."""
    dataframe.write_csv(filename, quote_style="always")
    print(f"CSV file '{filename}' created")

def save_as_excel(dataframe: pl.DataFrame, filename: str) -> None:
    """Save the given dataframe as an Excel filename."""
    dataframe.write_excel(filename)
    print(f"Excel file '{filename}' created")
```

Registros de páginas de compositores

Tras las secciones anteriores el núcleo del programa queda completado. Tenemos, en efecto, un flujo de trabajo bien definido.

`fetch_pages_by_category` obtiene los registros de páginas de IMSLP a partir de un título de categoría. Estos registros son luego analizados por `parse_page_record_collection`, que genera los registros de composiciones, los cuales se convierten en un *dataframe* por medio de `convert_to_dataframe`. Opcionalmente, el *dataframe* se transforma a su vez en un fichero CSV o Excel. Tras la finalización del proceso acabamos con varios ficheros que almacenan la base de datos de composiciones: como mínimo el fichero JSON y, opcionalmente, los ficheros CSV y Excel:

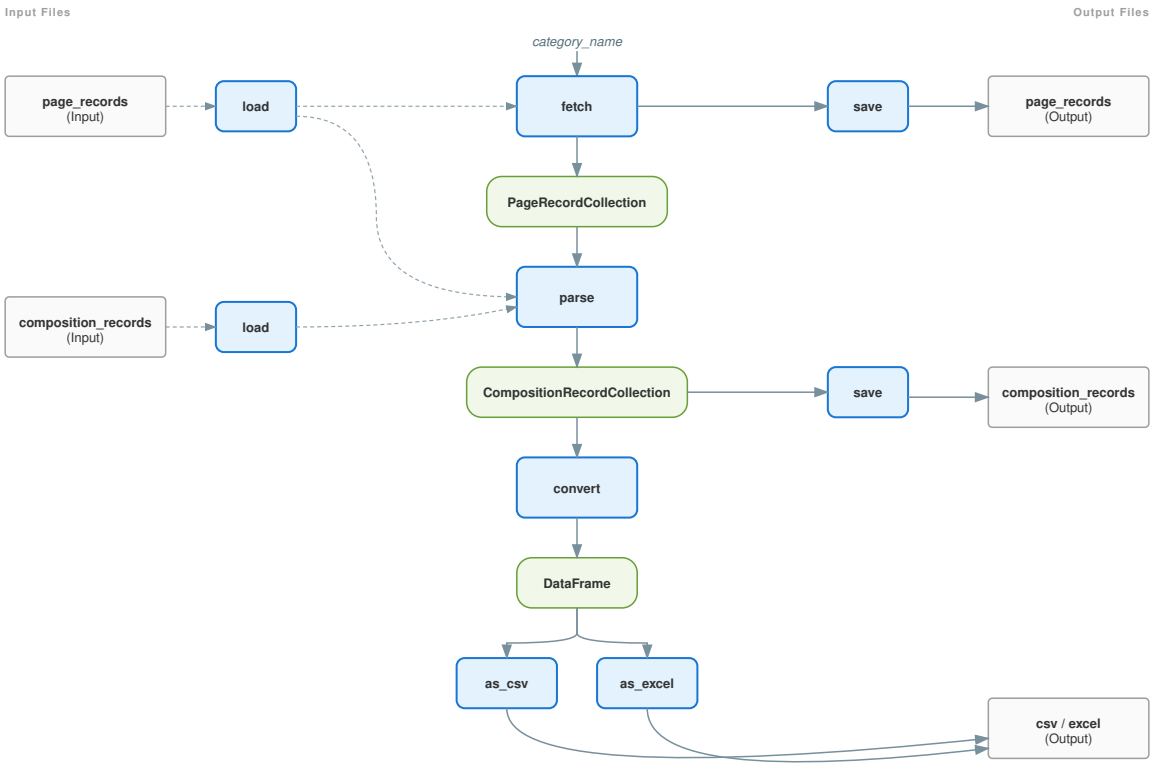

Figura 3: Diagrama del flujo del programa

El único eslabón débil en esta cadena está al principio, en el argumento de la función `fetch_pages_by_category`. El usuario debe conocer por anticipado no sólo que la categoría existe, sino también su título exacto. Así, por ejemplo, si se pasa como argumento a dicha función la categoría `Giuliani, Emilia` no se obtendrá lo deseado puesto que la categoría en IMSLP relacionada con la compositora romántica es `Giuliani-Guglielmi, Emilia`.

Existen diversas estrategias de resolución de este problema. Una de ellas es diseñar un función de consulta que produzca las categorías en IMSLP que encajan con un término de búsqueda como `Emilia Giuliani`, de entre las cuales el usuario seleccionaría una, que se pasaría como argumento a `fetch_pages_by_category`. Una segunda opción es presentar al usuario todas las posibles categorías de un ámbito dado que existan en IMSLP, por ejemplo, todas las categorías de compositores, y diseñar una interfaz que facilite la selección de una de esas categorías.

Puesto que en la parte final de este proyecto se creará una interfaz como la aludida, la alternativa más fácil de acometer es la descrita en segundo lugar.

El objetivo inmediato es, pues, obtener el nombre de todos los compositores de los que existe una página en IMSLP. La lista de esos compositores nutrirá la interfaz que permita al usuario elegir el compositor cuyas obras queremos añadir a la base de datos.

Ahora bien, aunque primariamente nos interesa el nombre del compositor en cuanto tal, que es el valor que se ha de pasar como argumento a `fetch_pages_by_category`, conviene contar también con el título completo de la página referida a ese compositor.

Consideremos un ejemplo concreto para clarificar este punto. La página en IMSLP relativa a Francisco Tárrega tiene como título interno —el que aparece en la URL de la página— la siguiente cadena de texto:

`Category:Tárrega, Francisco`

Este título, plenamente cualificado, es necesario si, por ejemplo, queremos obtener dicha página, lo cual debería hacerse por medio de `IMSLP_PAGES`:

```
IMSLP_PAGES["Category:Tárrega, Francisco"]
```

Si, por el contrario, lo que interesa es obtener las páginas de las composiciones de Francisco Tárrega, debemos usar el nombre del compositor en cuanto tal, esto es, sin cualificar con el prefijo `Category:`, y pasarlo como índice de `IMSLP_CATEGORIES`:

```
IMSLP_CATEGORIES["Tárrega, Francisco"]
```

Puesto que conviene tener a la mano ambos títulos, creamos una clase de colección especializada `ComposerPageRecordCollection` basada en `PageRecordCollection` a la que añadimos un campo `composers`. Así, una instancia de esta clase constará de dos campos: `page_titles` —heredado de `PageRecordCollection`—, que contendrá los nombres, plenamente cualificados de las páginas, y `composers`, que contendrá los nombres sin cualificar, esto es, los nombres propiamente dichos, de los compositores[6].

```
class ComposerPageRecordCollection(PageRecordCollection):
    @computed_field
    @property
    def composers(self) -> list[str]:
        return [
            record.page_title.removeprefix("Category:")
            for record in self.root
        ]
```

[6] Es probable que algún lector se pregunte por qué no diseñar un modelo para un registro de página de compositor, `ComposerPageRecord`, y hacer que la colección de registros de página `ComposerPageRecordCollection` quede integrada por instancias de `ComposerPageRecord`. En términos de arquitectura esa probablemente fuese una mejor decisión. El problema es que ello implica un diseño más complejo y abstracto que casa peor con el carácter introductorio de esta exposición. En realidad, aunque fue esa opción más abstracta la que implementé en primer lugar, llegué a la conclusión, tras una sesión de lluvia de ideas con la IA, de que era más amigable el diseño actual: menos preciso, sí, pero más comprensible para el no experto.

Sobre la base de esta definición diseñamos la función `fetch_composer_pages`, que obtiene las páginas de categorías de compositores en IMSLP, las cuales están organizadas como subcategorías de la supercategoría `Composers`.

Puesto que estas páginas de compositores son a día de hoy en torno a 30000, añadimos un argumento `only_titles` a `fetch_pages_by_category` con la idea de obtener sólo los títulos de páginas —y, en consecuencia, de compositores—, sin incluir el wikitexto de cada página, lo cual sobrecargaría al servidor de IMSLP.

Además, la implementación de `fetch_pages_by_category` diferencia entre el valor de `page_title` que se registra dependiendo de sí la página solicitada es una categoría o una página normal. En el caso de que se trate de una categoría se registra su nombre completamente cualificado, esto es, el que incluye el prefijo `Category:`

Con estos cambios, la nueva verión de `fetch_pages_by_category` sería la siguiente:

```
from mwclient.listing import Category

def fetch_pages_by_category(
    category_title: str,
    page_record_collection: PageRecordCollection,
    only_titles: bool = False
) -> PageRecordCollection:
    """Fetch IMSLP pages of the given category_title
    if not already contained in page_record_collection.
    If only_titles is True, fetch only page titles. Default: False
    Assume: the corresponding category exists in IMSLP.
    """
    category = IMSLP_CATEGORIES[category_title]
    new_page_records = []

    time_start = time.perf_counter()
    for page in category:
        page_title = (
            page.name if isinstance(page, Category)
            else page.page_title
        )

        if not page_record_collection.contains(page_title):
            match only_titles:
                case True:
                    page_record = fetch_page(page_title, only_title=True)
                case False:
                    wait_for_fetching(MIN_DELAY, MAX_DELAY)
                    page_record = fetch_page(page_title)
            if page_record:
                print(f"'{page_title}' fetched")
                new_page_records.append(page_record)
        else:
            print(f"'{page_title}' is already registered.")
    time_end = time.perf_counter()

    time_elapsed = time_end - time_start
    print(f"{len(new_page_records)} fetched in {time_elapsed:.2f} seconds")
    updated_records = page_record_collection.root + new_page_records
    return PageRecordCollection(updated_records)
```

Naturalmente, ello implica —como se ve— reimplementar `fetch_page` con el argumento parejo `only_title`, que, en caso de que sea `True`, el valor del campo `wikitexto` del registro se deja vacío.

```
def fetch_page(
    page_title: str,
    only_title: bool = False
) -> PageRecord | None:
    """Fetch the given IMSLP page title and produce its PageRecord.
    Returns None if retrieval fails.
    If only_title is True, set wikitext to empty string. Default: False
    Assume: page_title exists in IMSLP
    """
    try:
        imslp_page = IMSLP_PAGES[page_title]
    except (APIError, MaximumRetriesExceeded):
        print(f"{page_title} cannot be retrieved")
        return None
    return PageRecord(
        page_title = page_title,
        wikitext = "" if only_title else imslp_page.text()
    )
```

Esta versión refinada de `fetch_pages_by_category` es la que hace posible implementar `fetch_composer_pages`, cuyo propósito es crear una colección de registros de páginas de todos los compositores en IMSLP [7]:

```
IMSLP_COMPOSERS_CATEGORY_TITLE = 'Composers'

def fetch_composer_pages(
    composer_page_record_collection: ComposerPageRecordCollection
) -> ComposerPageRecordCollection:
    """Fetch IMSLP composer pages if not already contained in
    composer_page_record_collection.
    If only_titles is True (default), fetch only page titles.
    """
    updated_records = fetch_pages_by_category(
        IMSLP_COMPOSERS_CATEGORY_TITLE,
        composer_page_record_collection,
        only_titles=True
    )
    return ComposerPageRecordCollection(updated_records.root)
```

Nótese también que el valor devuelto por `fetch_composer_pages`, que debe ser una colección de registros de páginas de compositores, debe reconstruirse a partir de la colección de registros de páginas que produce `fetch_pages_by_category`. Es decir, reconstruimos el tipo específico de registros de páginas de compositores a partir de su tipo genérico, la clase padre, de registros de páginas en general.

El resultado de estos añadidos nos permite resolver el problema inicial. Ahora el usuario podrá elegir —como se verá más tarde— el nombre del compositor a partir del conjunto de todos los compositores que existen en los registros de páginas de compositores.

El flujo del programa quedaría, pues, extendido, como se muestra en el siguiente diagrama:

Funciones envoltorio de lectura y escritura

Antes de continuar con la siguiente sección y en previsión de lo que se verá en ella, refinemos un poco más la parte del código relativa a las funciones de lectura y escritura de los ficheros JSON mediante la aplicación de una técnica gracias a la cual se crean nuevas funciones fáciles de usar y recordar que *envuelven* la función principal de lectura o escritura, a la, además, se proporciona un fichero de datos por defecto. Se trata, pues, de funciones no esenciales, pero que simplifican y hacen más fácil de entender el

[7] Una colección de tales registros está disponible en el subdirectorio `data` de este proyecto, habida cuenta de que la obtención de esos en torno a 30000 registros puede suponer con la configuración actual del programa varias horas de descarga.

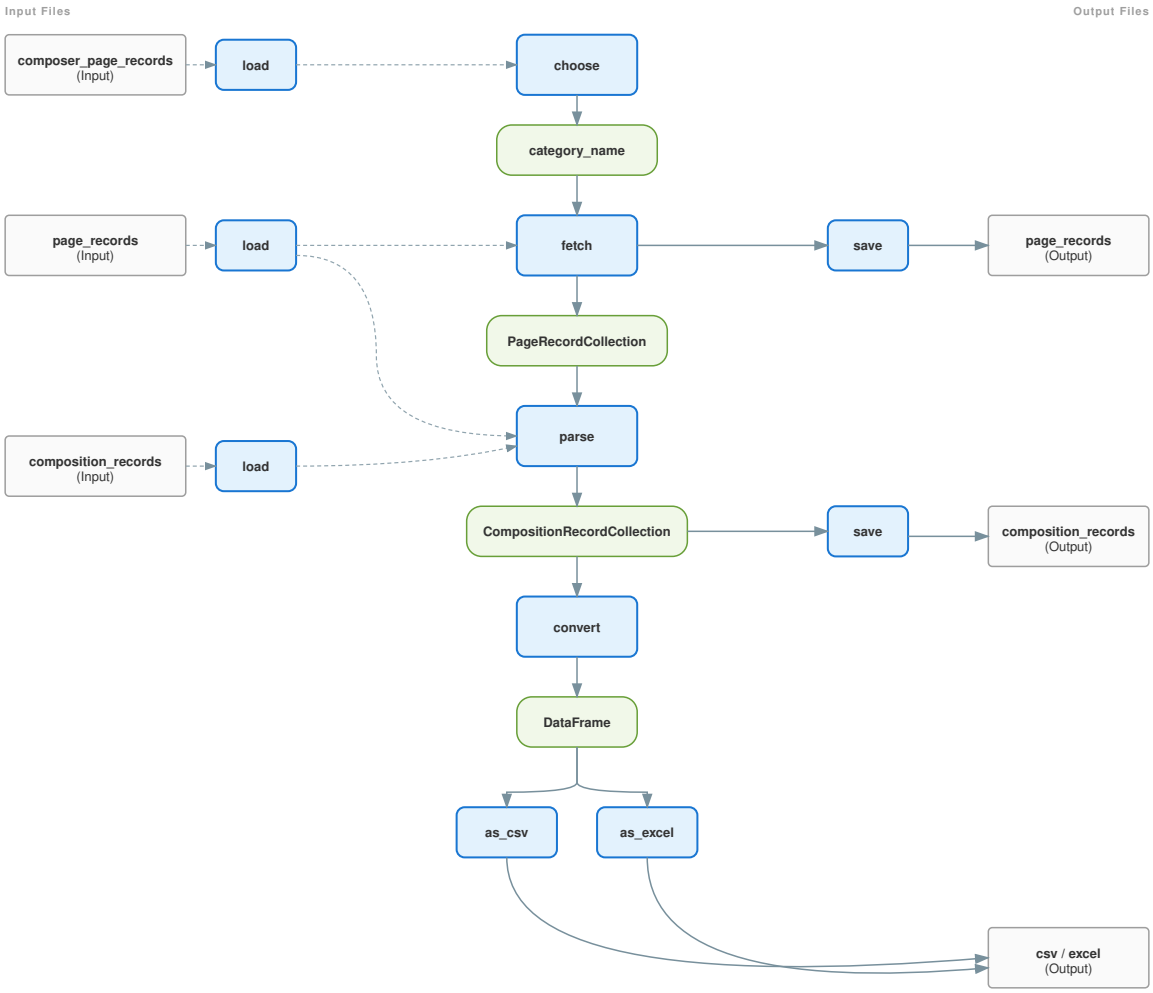

Figura 4: Diagrama del flujo del programa extendido

código al presentar en un *envoltorio* más amigable las funciones primarias, de ahí su nombre técnico de *wrappers*.

De hecho, de entre las funciones de lectura y escritura anteriormente descritas, algunas de ellas se pueden considerar ya funciones envoltorio. En sentido estricto, las únicas funciones primarias allí son `load_page_record_collection` y `save_record_collection`.

Diseñamos, pues, alguna más y damos a todas esas funciones de conveniencia un valor por defecto, quedando como sigue el conjunto completo de las funciones de lectura y escritura:

```
# *** Basic functions
def load_record_collection(filename: str, cls: type[RecordCollection]):
    """Load the given JSON filename as a RecordCollection of the given cls."""
    try:
        json_string = Path(filename).read_text()
    except FileNotFoundError:
        json_string = "[]"
        print(f"'{filename}' does not exist. Starting from scratch...")
    return cls.model_validate_json(json_string)

def save_record_collection(
    record_collection: RecordCollection,
    filename: str
) -> None:
    """Save the given record collection as a JSON filename."""
    def backup(src: Path) -> None:
        """Backup path"""
        dest = src.with_suffix(".json.bak")
        dest.write_text(src.read_text())
        print(f"'{src}' backed up as '{dest}'")

    json_string = record_collection.model_dump_json()
    json_path = Path(filename)
    if json_path.exists():
        backup(json_path)
    json_path.write_text(json_string)
    print(f"Records saved in '{json_path}'")

# *** Wrappers
def load_page_record_collection(filename: str) -> PageRecordCollection:
    """Load the given JSON filename as a PageRecordCollection."""
    return load_record_collection(filename, PageRecordCollection)

# *** Wrappers with default values
def load_composer_page_record_collection(
    filename: str = IMSLP_COMPOSER_PAGE_RECORDS_JSON
) -> ComposerPageRecordCollection:
    """Load the given JSON filename (IMSLP_COMPOSER_PAGE_RECORDS_JSON by
    default) as a ComposerPageRecordCollection."""
    return load_record_collection(filename, ComposerPageRecordCollection)

def load_composition_page_record_collection(
    filename: str = IMSLP_COMPOSITION_PAGE_RECORDS_JSON
) -> PageRecordCollection:
    """Load the given JSON filename (IMSLP_COMPOSITION_PAGE_RECORDS_JSON by
    default) as a PageRecordCollection."""
    return load_page_record_collection(filename)

def load_composition_record_collection(
    filename: str = IMSLP_COMPOSITION_RECORDS_JSON
) -> CompositionRecordCollection:
```

```
    """Load the given JSON filename (IMSLP_COMPOSITION_RECORDS_JSON by
    default) as a CompositionRecordCollection."""
    return load_record_collection(filename, CompositionRecordCollection)

def save_composer_page_record_collection(
    record_collection: ComposerPageRecordCollection,
    filename: str = IMSLP_COMPOSER_PAGE_RECORDS_JSON
) -> None:
    """Save the given record collection as filename,
    IMSLP_COMPOSER_PAGE_RECORDS_JSON by default."""
    return save_record_collection(record_collection, filename)

def save_composition_page_record_collection(
    record_collection: PageRecordCollection,
    filename: str = IMSLP_COMPOSITION_PAGE_RECORDS_JSON
) -> None:
    """Save the given record collection as filename,
    IMSLP_COMPOSITION_PAGE_RECORDS_JSON by default."""
    return save_record_collection(record_collection, filename)

def save_composition_record_collection(
    record_collection: CompositionRecordCollection,
    filename: str = IMSLP_COMPOSITION_RECORDS_JSON
) -> None:
    """Save the given record collection as filename,
    IMSLP_COMPOSITION_RECORDS_JSON by default."""
    return save_record_collection(record_collection, filename)
```

Ensamblaje de las piezas

A lo largo de las secciones precedentes hemos construido todos los componentes necesarios del programa. El siguiente paso es crear funciones que combinen estas piezas y realicen el trabajo conjunto de todas ellas.

Con la vista puesta en el diagrama de flujo del programa podemos describir el proceso completo punto por punto y exponerlo en forma de receta, donde cada paso cuenta con el resultado producido por el anterior.

El único ingrediente inicial de la receta —que el usuario tendrá que introducir a través de la interfaz que se desarrollará en la sección seguiente de este documento— es el nombre del compositor de cuyas composiciones se quiere obtener información registrable. El resultado visible de aplicar la receta es un *dataframe* integrado por todos los registros de composiciones ya guardados más aquellos que corresponden al compositor de interés. Las pasos de la receta son los siguientes:

1. Carga los ficheros de datos, los que contienen los registros de páginas de compositores, de páginas de composiciones y de composiciones.
2. En caso de que el compositor introducido no exista entre los disponibles termina el programa generando un *dataframe* de las composiciones actuales[8].
3. Obtén los páginas de composiciones del compositor introducido.
4. Analiza los registros de páginas procedentes del paso anterior.
5. En el caso de que los resultados de los dos últimos pasos posean nuevos registros, actualiza los ficheros que contienen nuestros conjuntos de datos.
6. Convierte la colección de composiciones producida en un *dataframe*, que será el resultado final de la receta.

[8] En buena lógica, el compositor introducido existirá y este paso no sería necesario. Sin embargo, por otra parte, conviene *defenderse* contra la posibilidad de un abuso de la función, por ejemplo, que alguien pueda usar la función sin pasar por la interfaz de usuario e introducir un compositor que no exista o simplemente un cadena vacía.

O, si se quiere, expresado resumidamente y haciendo alusión a la funciones implicadas:

1. Carga los conjuntos de datos por defecto. Funciones `load`.
2. Si el compositor dado no existe, muestra las composiciones actuales. Función `convert_to_dataframe`.
3. Obten las páginas de composiciones del compositor dado. Función `fetch_pages_by_category`.
4. Analiza las páginas de composiciones: Función `parse_page_record_collection`.
5. Actualiza los conjuntos de datos en disco. Funciones `save`.
6. Muestra las composiciones actualizadas. Función `convert_to_dataframe`.

Llamemos a esta receta —otra función, por supuesto— `update_records`. El código que refleja los pasos indicados es el siguiente:

```
def update_records(composer: str) -> pl.DataFrame:
    """Update the default datasets of composition_pages and compositions
    with works by the given composer.
    Return dataframe of the compositions set with new compositions added.
    """
    # Load default datasets
    composer_pages = load_composer_page_record_collection()
    composition_pages = load_composition_page_record_collection()
    compositions = load_composition_record_collection()

    # If composer does not exist, show current compositions
    composers = composer_pages.composers
    if composer not in composers:
        return show_compositions(compositions)

    # Fetch composition pages by composer
    updated_composition_pages = fetch_pages_by_category(
        composer,
        composition_pages
    )

    # Parse composition pages
    updated_compositions = parse_page_record_collection(
        updated_composition_pages,
        compositions
    )

    # Save datasets if records changed
    if not updated_composition_pages == composition_pages:
        save_composition_page_record_collection(updated_composition_pages)
    if not updated_compositions == compositions:
        save_composition_record_collection(updated_compositions)

    # Show updated compositions
    return show_compositions(updated_compositions)

def show_compositions(
    compositions: CompositionRecordCollection
) -> pl.DataFrame:
    """Produce a dataframe with BASIC_COLS and anchors of compositions."""
    return convert_to_dataframe(
        compositions,
        cols=BASIC_COLS,
        with_anchors=True
    )
```

La operación final, generar el *dataframe* a partir de los registros de composiciones, se ha envuelto en la función `show_compositions` para evitar repetir el mismo código en la producción final del *dataframe*.

Interesa, por otro lado, disponer de una función que realice la tarea de producir el *dataframe* de las composiciones actuales independientemente de la función de actualización de registros, es decir, una función que combine la carga de los datos de composiciones actuales con la generación del *dataframe*:

```
def show_current_compositions(
    filename: str = IMSLP_COMPOSITION_RECORDS_JSON
) -> pl.DataFrame:
    """Produce a dataframe of the composition records in the given filename."""
    current_compositions = load_composition_record_collection()
    return show_compositions(current_compositions)
```

La interfaz de usuario

El programa diseñado hasta aquí es perfectamente funcional, pero faltan mecanismos para que el usuario pueda utilizarlo con facilidad. Tal es el papel que desempeña la interfaz de usuario. A la hora de crear una interfaz amigable una opción fácil de implementar es construir una aplicación que se ejecute en el navegador.

En esta sección describimos brevemente el diseño de una interfaz basada en la biblioteca `Panel`. El código de la interfaz está en `imslpdb_ui.py`, y ha sido producido, salvo en unos pocos lugares, por una IA de acuerdo con una clara especificación (*prompt*) de las partes que componen la interfaz y de su funcionamiento:

La aplicación consta de los siguientes componentes:

- Una tabla (`tabulator`) que muestra el contenido del *dataframe* de composiciones actuales.
- Una caja de texto (`composer_input`) en la que usuario va escribiendo el nombre del compositor que busca y donde, a medida que lo va haciendo, se reduce el nombre de posibles candidatos, los cuales proporciona el campo `composer` de la colección de registros de páginas de compositores.
- Un botón (`add_button`) que, cuando se pulsa, ejecuta la función `update_composers` y, en consecuencia, actualiza el conjunto de las composiciones añadiendo aquellas no registradas que pertenecen al compositor introducido.
- Un indicador del progreso (`progress_bar`) de descarga de las páginas de IMSLP correspondientes al compositor cuyas obras se están añadiendo.
- Una caja de texto (`search_input`) que permite filtrar filas de la tabla de composiciones.
- Un selector (`row_count_select`) que permite determinar el número de filas que mostrar en cada pantalla del navegador.
- Una pantalla de terminal (`terminal`) que reproduce los mensajes que la función `update_composers` emite mientras se ejecuta.

La aplicación es lo que técnicamente se denomina *reactiva*. Es decir, la tabla de composiciones se actualiza automáticamente dependiendo del estado de la aplicación en cada momento.

La aplicación se ejecuta mediante la siguiente orden, que abrirá la aplicación en el navegador por defecto:

```
panel serve imslpdb_ui.py --show
```

El lanzador de la aplicación

El usuario no tendría por qué conocer la instrucción necesaria para abrir en el navegador la aplicación anteriormente expuesta. Mantenerlo en la ignorancia de los detalles engorrosos es una máxima en las interfaces amigables. Creamos, pues, otro fichero, que llamaremos `main.py`, cuyo único propósito es lanzar la aplicación a través de la función `start_ui`:

```
from panel.io.server import serve
import imslpdb_ui

def start_ui():
    serve(imslpdb_ui.app, title="ImslpDB UI")

if __name__ == "__main__":
    print("\nYou can shutdown the server by pressing Ctrl+C\n")
    start_ui()
```

Además, creamos un script de bash con el nombre **imslpdb**, que inicia el entorno de Python en el directorio actual y llama a **imslpdb_main** desde el intérprete de Python:

```
# Activate the Python environment and call the module imslpdb_main
#!/usr/bin/env bash
source .venv/bin/activate
python imslpdb_main
```

A este fichero hay que otorgarle permisos de ejecución:

```
chmod u+x imslpdb
```

El usuario puede ahora ejecutar la aplicación con la instrucción[9]:

```
./imslpdb
```

[9] Una forma alternativa y más cómoda es instalar este proyecto, en cuyo caso bastará la orden **imslpdb** para lanzar la aplicación. La información sobre el proceso de instalación se encuentra en el README.