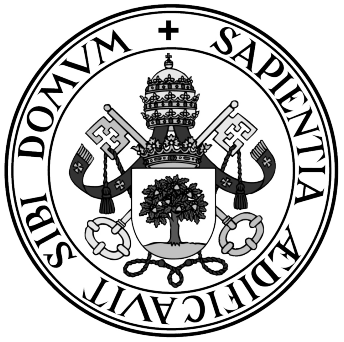


Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA
DE SEGOVIA

Grado en Ingeniería Informática
de Servicios y Aplicaciones

Rosetta: catálogo online para bibliotecas compatible con datos estructurados

---

Alumno: José Miguel Moreno López

Tutor: Juan José Álvarez Sánchez

El código es como un chiste,
si tienes que explicarlo es porque es malo

CORY HOUSE

Resumen

Las bibliotecas actuales están tecnológicamente estancadas en el siglo pasado. La falta de innovación y de acuerdos entre instituciones ha provocado que los estándares y formatos utilizados para almacenar, representar e intercambiar metadatos apenas hayan cambiado desde la década de los noventa, y las dificultades para encontrar recursos bibliográficos en los buscadores web de estos centros siguen siendo las mismas.

Rosetta es un nuevo tipo de software distribuido bajo licencia GPLv3 que plantea una alternativa a estos formatos obsoletos y permite a las instituciones ofrecer un OPAC moderno sin la necesidad de cambiar el sistema informático o el software ILS que estén utilizado.

Palabras clave: bibliotecas, rosetta, ILS, OPAC, software

Abstract

Today's libraries are technologically stuck in the last century. The lack of innovation and agreements between institutions has meant that the standards and formats used to store, represent and exchange metadata have barely changed since the nineties, and the difficulties in finding bibliographic resources in the web search engines of these centers continue to be the same.

Rosetta is a new type of software distributed under the GPLv3 license that proposes an alternative to these obsolete formats and allows institutions to offer a modern OPAC without the need to change their computer systems or ILS software.

Keywords: libraries, rosetta, ILS, OPAC, software

ÍNDICE DE CONTENIDOS

1. Introducción **1**
- 1.1. Motivación . . . 2
  - 1.1.1. Problemática . . . 3
- 1.2. Estado del arte . . . 4
  - 1.2.1. Millennium ILS . . . 5
  - 1.2.2. Sierra LSP . . . 5
  - 1.2.3. Ex Libris Alma . . . 6
  - 1.2.4. SirsiDynix Symphony . . . 7
  - 1.2.5. Koha . . . 8
  - 1.2.6. Comparativa y conclusiones . . . 9
- 1.3. Objetivos y alcance . . . 9
- 1.4. Metodología . . . 10
  - 1.4.1. Herramientas y tecnologías utilizadas . . . 11
  - 1.4.2. Ciclo de trabajo . . . 13
  - 1.4.3. Planificación . . . 14
  - 1.4.4. Presupuesto . . . 18
- 1.5. Contenidos del CD . . . 20

2. Análisis **21**
- 2.1. Características del sistema . . . 21
  - 2.1.1. Árbol de características . . . 22
- 2.2. Actores del sistema . . . 23
- 2.3. Requisitos de usuario . . . 23
  - 2.3.1. Diagrama de casos de uso . . . 24
  - 2.3.2. Especificación de los casos de uso . . . 25
- 2.4. Requisitos de información . . . 29
  - 2.4.1. Diagrama entidad-relación . . . 29
  - 2.4.2. Diccionario de datos . . . 30
- 2.5. Requisitos no funcionales . . . 33

3. Diseño e implementación **35**
3.1. Entidades . . . 37
3.1.1. Jerarquía de entidades . . . 40
3.1.2. Identificadores . . . 41
3.1.3. Relaciones entre entidades . . . 43
3.1.4. Caché de entidades . . . 45
3.2. Motor de búsqueda . . . 51
3.2.1. Consultas de búsqueda . . . 53
3.2.2. Proveedores . . . 57
3.2.3. Agrupación de resultados . . . 59
3.2.4. Datos estructurados con Wikidata . . . 62
3.3. Interfaz gráfica . . . 63
3.3.1. Extensión de Twig . . . 65
3.3.2. Mapas de ubicación . . . 66
3.3.3. Internacionalización . . . 68

4. Pruebas **71**

5. Documentación **83**
5.1. Manual de usuario . . . 83
5.1.1. Iniciar una búsqueda . . . 83
5.1.2. Resultados de búsqueda . . . 84
5.1.3. Página de detalle . . . 85
5.2. Manual de despliegue . . . 86
5.2.1. Despliegue automático . . . 86
5.2.2. Despliegue avanzado . . . 87
5.2.3. Despliegue manual . . . 88
5.3. Manual de personalización . . . 92
5.3.1. Configuración de la aplicación . . . 92
5.3.2. Personalización de la interfaz web . . . 94
5.3.3. Configuración de mapas . . . 96

6. Conclusiones **99**
6.1. Posibles mejoras . . . 100

Índice de figuras **101**

Índice de tablas **103**

Índice de códigos **105**

Referencias **107**

CAPÍTULO

1

INTRODUCCIÓN

Desde sus orígenes hace ya más de cuatro mil años, las bibliotecas han tenido como principal objetivo preservar el conocimiento humano y su historia. El afán de las últimas décadas por conseguir que estos centros sean más abiertos y accesibles se está viendo mermado por una creciente **centralización de la información** que está provocando, a su vez, una privatización de la misma.

El resultado de este Trabajo Fin de Grado (el software "Rosetta") pretende ser una materialización de la primera ley de la bibliotecología de Ranganathan: «los libros están para usarse» [1].

De nada sirve contar con un amplio catálogo de recursos bibliotecarios si estos no pueden ser localizados a través de un buscador. De igual forma, si el espíritu moderno de las bibliotecas consiste en que los usuarios accedan a todos los medios que estas pueden ofrecer, no tiene sentido que los metadatos (y en última instancia las bases de datos) estén centralizados en servidores de empresas privadas que poseen y controlan sistemas informáticos propietarios.

Rosetta es un aplicación web formada por un **OPAC**[1] y un *broker* o **gestor de contenidos**. Este último componente, hasta ahora poco conocido en el ámbito de bibliotecas, es un agente que conecta con múltiples fuentes de datos externas para obtener los resultados de las búsquedas que se realizan desde el OPAC.

[1] *Online Public Access Catalog*

De esta forma se consigue crear una separación entre la información y los programas que la utilizan, permitiendo una **descentralización de las bibliotecas** en la que cada centro es responsable de gestionar su catálogo y el resto de entidades pueden solicitar acceder a él.

Además, como Rosetta incluye un módulo de OPAC, se puede sustituir la plataforma web de búsqueda de una biblioteca **sin tener que cambiar el sistema informático** que se encuentre por debajo y con independencia del protocolo o formato que éste último utilice para conectar con la base de datos.

De esta última característica es de donde Rosetta recibe su nombre[2], pues su núcleo o *broker* es capaz de conectar con casi cualquier sistema de catalogación al poder "traducir" protocolos y formatos de forma nativa, alcanzando una **interoperabilidad** sin precedentes.

1.1. Motivación

Este proyecto surge de la paupérrima interfaz gráfica de Almena, el OPAC de la Universidad de Valladolid a fecha de la realización del trabajo, basado en Millennium ILS®. A raíz de la pobre experiencia de usuario que ofrecía dicho buscador, en 2017 me propuse desarrollar una aplicación web alternativa llamada "Bibliozambrano"[3] que indexara de forma semanal el catálogo de recursos bibliográficos de la Universidad de Valladolid con el fin de obtener mejores resultados de búsqueda.

Figura 1.1: Página de inicio de Bibliozambrano

[2]La palabra "Rosetta" hace referencia a la piedra egipcia del mismo nombre que contiene un decreto del faraón Ptolomeo V escrito en tres lenguas distintas

[3]Véase `https://www.bibliozambrano.com`

A raíz de lo aprendido de su construcción y con el objetivo de democratizar la oferta de software para bibliotecas disponible en ese momento, decidí crear una plataforma libre y gratuita que dotase de las funcionalidades de Bibliozambrano a cualquier centro **sin tener que volcar el catálogo** periódicamente, pudiendo conservar la infraestructura existente.

Es importante mencionar que Rosetta **no pretende ser una alternativa definitiva** al software actual de bibliotecas ya que carezco de los medios, el tiempo o la inversión necesaria para conseguir tal fin. El objetivo es, entonces, **proponer una nueva arquitectura** para dichos sistemas **así como un nuevo formato para representar la información** basado en jerarquía de entidades.

1.1.1. Problemática

Las bibliotecas del siglo XXI se enfrentan a dos problemas socio-políticos que afectan a su crecimiento y desarrollo:

1. Monopolización del software de gestión de bibliotecas

2. Burocracia excesiva y falta de acuerdos entre grandes entidades

El control casi absoluto del mercado de software de bibliotecas por parte de los tres gigantes de la industria (ProQuest, Innovative Interfaces y SirsiDynix) no fomenta la innovación ni favorece la creación de alternativas gratuitas. La única excepción libre es el software Koha, que lamentablemente no supone una alternativa lo suficientemente fuerte como para plantar cara al resto de programas existentes.

El segundo problema es la falta de organizaciones sin ánimo de lucro con suficiente peso en la industria o el seguimiento de estándares modernos para el intercambio de datos. La principal institución en bibliotecas es la ***Library of Congress*** (LOC) de los Estados Unidos, principal impulsora del formato MARC 21, entre otros.

Este formato, creado en 1999 [2], está severamente anticuado y ha crecido de forma descontrolada en las últimas década hasta convertirse en innecesariamente complejo pese a no cubrir todos los casos de uso. Por si fuera poco, la inmensa mayoría de ILS[4] solo implementan las versiones más antiguas de este formato (las de 1999 hasta 2001).

Ante la falta de estándares usables y de su aplicación por parte de los fabricantes de software, surgen disputas entre otras organizaciones y grandes bibliotecas sobre cuál debería ser el sustituto de MARC 21 sin llegar a ningún acuerdo firme y, en consecuencia, siguiendo cada institución el formato que mejor le parece (RDA, OAI-PMH, Bibframe y otra decenas de formatos que casi nadie usa).

[4] *Integrated Library System* (Sistema Integrado de Gestión Bibliotecaria)

A partir del año 1997 la alternativa que más fuerza cobró fue ***Resource Description Framework*** (RDF) [3] propuesta por el W3C[5], aunque una vez más la falta de acuerdos entre bibliotecas llevó al grupo de trabajo de dicho formato a cesar su actividad el 1 de julio de 2014. A día de hoy, RDF sigue siendo una recomendación en vez de un estándar y la mayoría de desarrolladores lo consideran un formato muerto.

Una posible continuación de la filosofía de RDF es la comunidad **schema.org** [4] que, fundada por los principales buscadores de Internet en 2011 y con el apoyo del W3C, pretende establecer una serie de estándares para representar la información de páginas web y otras aplicaciones mediante el marcado de datos, pudiendo estos ser formateados en HTML, JSON-LD o incluso RDFa[6].

Pese a existir una extensión de Schema.org[7] específicamente diseñada para representar registros bibliográficos de forma moderna, las bibliotecas siguen sin usarlo al no haber llegado todavía a un acuerdo entre las instituciones fruto de no querer aceptar que el desarrollo de RDF finalizó hace años.

1.2. Estado del arte

El mercado en el que se mueve Rosetta es el de los sistemas integrados de gestión de bibliotecas o ILS. El software englobado dentro de esta categoría consta de múltiples componentes o módulos que llevan a cabo distintas tareas y suelen ser los siguientes:

- Adquisiciones: pedidos de nuevos ejemplares y facturación
- Catalogación: clasificación de recursos
- Circulación: gestión de préstamos
- Publicaciones periódicas: gestión de revistas, periódicos y colecciones
- OPAC (catálogo online): aplicación web para búsqueda de recursos

Rosetta no es un ILS pues solo consta de dos módulos: OPAC y *broker*, siendo este último algo hasta ahora desconocido en el ámbito de bibliotecas; por lo que no se consideraría un sustituto para estos programas sino más bien un complemento.

Aún así, en este apartado se hace un análisis de competencia de los ILS más populares al ser el software que más se parece al desarrollado en este TFG.

[5] *World Wide Web Consortium*
[6] RDFa permite representar textos de RDF en documentos XML
[7] Véase `https://bib.schema.org/`

1.2.1. Millennium ILS

Millennium fue el primer ILS desarrollado por la empresa estadounidense Innovative Interfaces, Inc. (III) a finales del siglo pasado. Entre sus módulos se encuentra INNOPAC, un OPAC creado por la misma empresa años antes del lanzamiento de Millennium. Actualmente está descontinuado y su sustituto oficial es Sierra.

Está desarrollado en Java y utiliza un motor de base de datos propietario desarrollado por Innovative, aunque también es compatible con Oracle Database Server.

Dado que se lanzó a finales de los años noventa y no ha sufrido grandes cambios en las décadas posteriores, la interfaz gráfica de Millennium es bastante rudimentaria en cuanto a estándares modernos se refiere y no ofrece compatibilidad con los sistemas operativos más recientes.

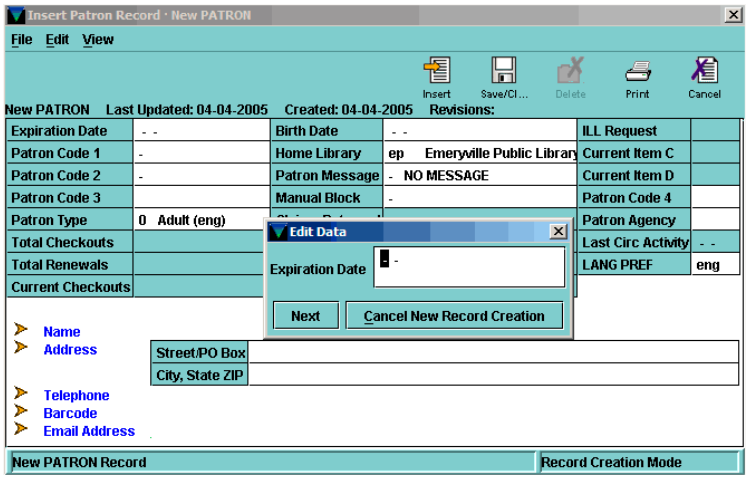
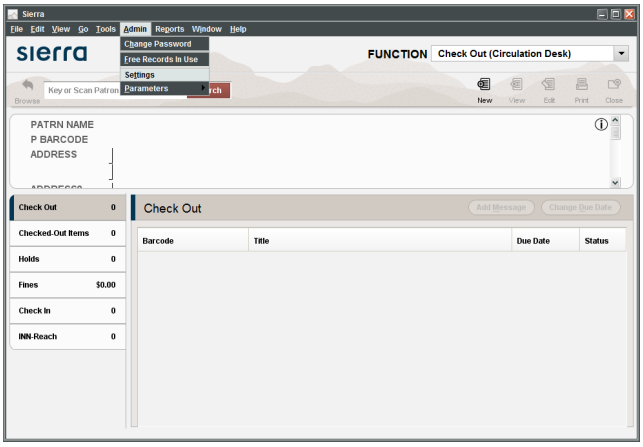


Figura 1.2: Ventana de creación de usuarios de Millennium ILS

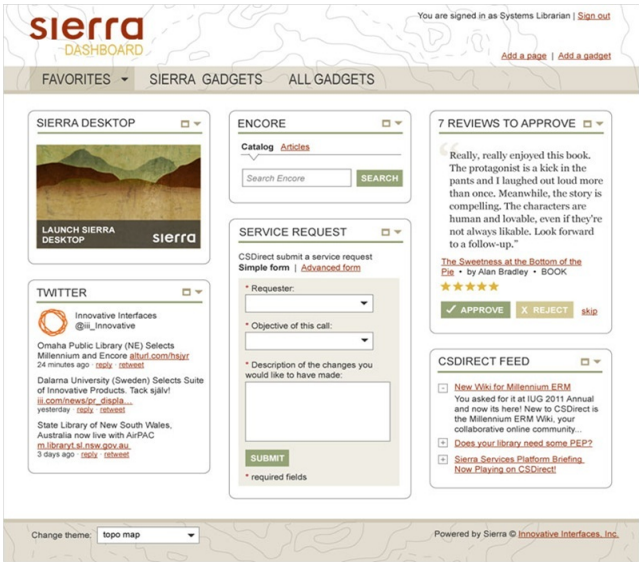
1.2.2. Sierra LSP

Inicialmente conocido como Sierra ILS, se ha convertido en un LSP (*Library System Platform*), lo que significa que ya no se distribuye como software instalable por las instituciones sino un **servicio** por el que se paga periódicamente.

En sus inicios Sierra ILS era una aplicación Java de código abierto (pero no software libre) que en los últimos años se ha convertido en SaaS (*Software as a Service*) como el resto de alternativas dominantes del mercado. Al igual que Millennium, también está programado en Java pero utiliza PosgreSQL como motor de base de datos.



(a) Pantalla de inicio de Sierra ILS



(b) Página de inicio de Sierra LSP

Figura 1.3: Interfaces gráficas de Sierra

1.2.3. Ex Libris Alma

Quizá el producto que más fuerza está cobrando últimamente, Alma es un SaaS desarollado por el grupo Ex Libris, de adquisición reciente por la compañía ProQuest.

Su primera versión se lanzó de forma oficial en enero de 2011 con acuerdos de colaboración con el Boston College o la biblioteca de la universidad de Princeton [6] y sustituye a Voyager, el ILS de Endeavor Information Systems Inc., al ser adquirida esta empresa por Ex Libris en diciembre de 2006.

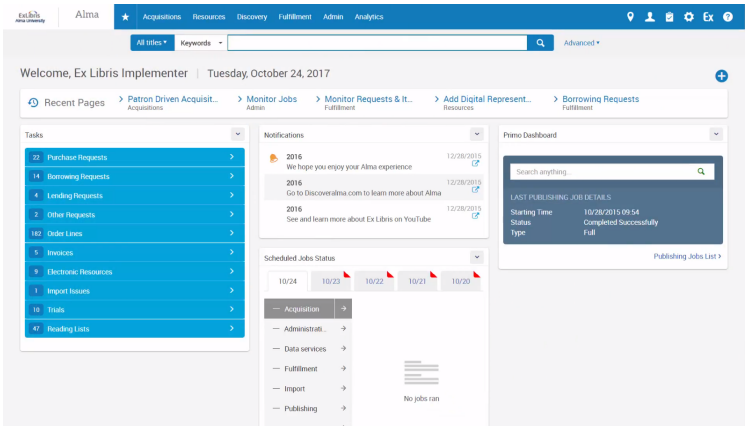


Figura 1.4: Página de inicio de Alma

A diferencia de Sierra LSP, Alma se diseñó como una aplicación cien por cien en la nube desde sus inicios y todos los catálogos de las bibliotecas se gestionan y alojan en los servidores de Ex Libris y ProQuest.

1.2.4. SirsiDynix Symphony

El sucesor de Unicorn (otro ILS de la misma compañía), Symphony es el software con menos prestaciones de los recogidos en este apartado. Aún así y debido a la gran acogida de su predecesor, sigue siendo usado por una considerable cantidad de bibliotecas, sobre todo en el continente americano.

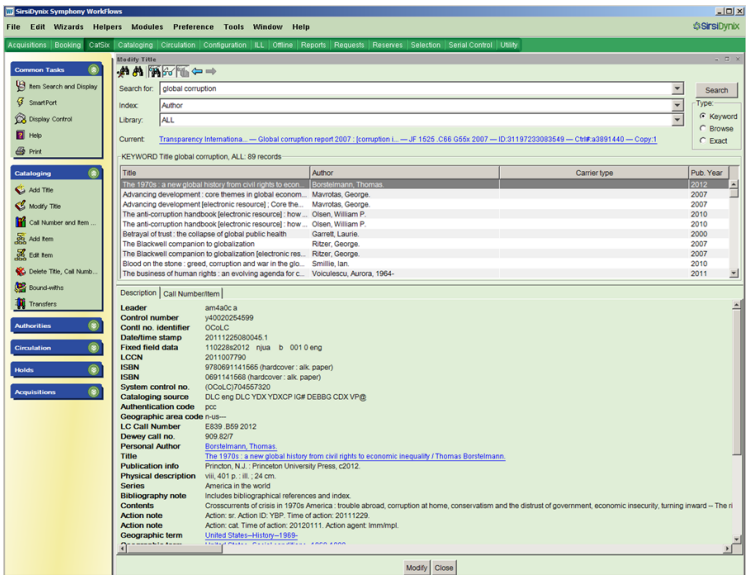


Figura 1.5: Edición de entradas de SirsiDynix Symphony

Como el resto de competidores, también está sufriendo una transformación de aplicación de escritorio a servicio web (éste último llamado SymphonyWeb, sucesor de BLUEcloud), aunque sin tanto éxito o popularidad como Ex Libris y su plataforma Alma.

1.2.5. Koha

Koha es una solución ILS libre y de código abierto surgida en enero del año 2000 bajo licencia GPL (GNU Public License). Es una solución ILS en formato de aplicación web que, a diferencia del resto de productos basados en la nube, es auto-alojada (*self-hosted*). Es decir, cada biblioteca instala Koha en un servidor central al que se conectan los bibliotecarios y resto de clientes.

Gracias a esta arquitectura (que es la misma utilizada por Rosetta) los datos siguen siendo gestionados por la institución a la que pertenecen y no pasan a ser propiedad de terceros, como en el caso de Ex Libris Alma.

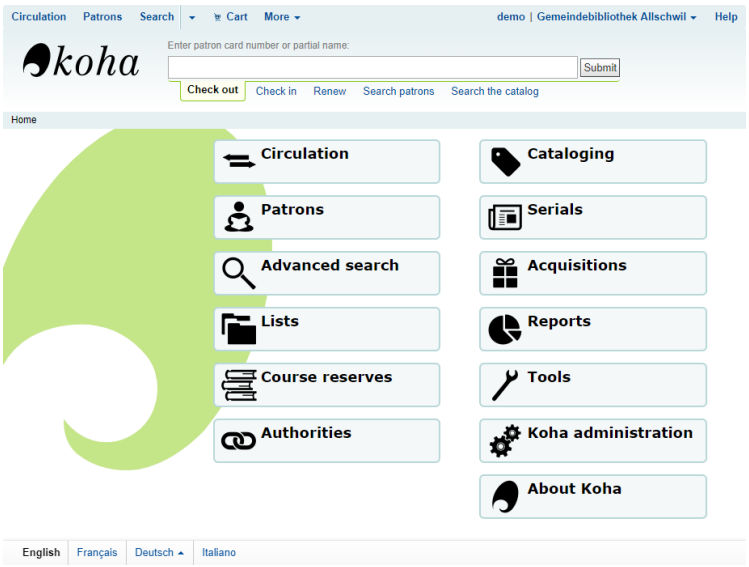


Figura 1.6: Página de inicio de administración de Koha

1.2.6. Comparativa y conclusiones

	Millennium	**Sierra**	**Alma**	**Symphony**	**Koha**
Desarrollador	Innovative Interfaces	Innovative Interfaces	Ex Libris Group (ProQuest)	SirsiDynix	Koha Community
Licencia	Propietario	Propietario	Propietario	Propietario	GPL 3.0
Precio	Pago único por licencia*	Coste fijo anual*	Coste por uso*	Coste fijo anual*	Gratuito
Lanzamiento	1997	2013	2011	2006	2000
Tipo	ILS	ILS/LSP	LSP	ILS/LSP	*Self-hosted*
En desarrollo	-	✓	✓	-	✓
Adquisiciones	✓	✓	✓	✓	✓
Catalogación	✓	✓	✓	✓	✓
Circulación	✓	✓	✓	✓	✓
P. periódicas	✓	✓	✓	✓	✓
OPAC	✓	✓	✓	✓	✓
GUI moderna	-	✓	✓	-	-
Fácil de usar	-	✓	✓	-	-
API préstamos	✓	✓	✓	✓	✓
API recursos	-	✓	✓	-	✓

Tabla 1.1: Comparativa de software para bibliotecas

El asterisco (*) mencionado en la tabla anterior se refiere a la imposibilidad de conocer el rango de precios del producto al variar con cada contrato y a la existencia de acuerdos de confidencialidad que impiden revelar dicha información.

1.3. Objetivos y alcance

El objetivo de este trabajo es el desarrollo de un OPAC para bibliotecas que siente las bases para una nueva generación de sistemas informáticos bibliotecarios.

Para lograr tal objetivo, se acota el alcance del proyecto dentro de los siguientes enunciados:

- Crear un software retrocompatible con los ILS más populares del mercado y adaptable a cualquier formato existente e imaginable
- Estructurar la información de forma que se puedan crear relaciones entre los datos (datos estructurados)
- Ofrecer una interfaz web para buscar información en el catálogo de la aplicación
- Permitir a los administradores de las plataformas que usen este software poder configurar las fuentes de datos y personalizar la interfaz web fácilmente

1.4. Metodología

Para la realización de este proyecto se ha empleado una **metodología incremental** en la que se ha ido construyendo el producto final progresivamente al añadir funcionalidades en intervalos de tiempo periódicos denominados "incrementos".

Todas estas iteraciones o incrementos constan de cuatro fases (análisis, diseño, implementación y pruebas) y siempre terminan en un entregable del producto para su evaluación (*pre-releases* o *nightlies*).

Gracias al uso de un sistema de integración continua como Travis CI, la fase de pruebas y la creación y despliegue al servidor de *staging* de las versiones *alpha* está prácticamente automatizada y apenas requiere de intervención humana, agilizando los procesos.

Antes de esta fase principal de incrementos y debido a la naturaleza novedosa del proyecto, se utilizó un **modelo de prototipos** para determinar la mejor arquitectura para crear el producto final. Al ser un proceso de desarrollo evolutivo, se crearon múltiples programas sencillos, mínimamente funcionales, en muy poco tiempo para probar formas de lograr el objetivo deseado.

Gracias a los conocimientos adquiridos durante la fase de prototipado, se pudo diseñar y desarrollar un sistema que cumpliera con los requisitos de nuestro proyecto en la siguiente etapa.

Los incrementos realizados durante la fase incremental del proyecto fueron los siguientes:

- **Primer incremento (v0.1.0-alpha)**: crear código base del proyecto utilizando el framework Symfony. Incluye la configuración de Webpack para compilar los recursos estáticos y de Travis CI para la integración continua, así como

el desarrollo de las entidades mínimas y el proveedor de Z39.50 para poder empezar a probar la aplicación.

- **Segundo incremento (v0.2.0-alpha)**: añadir compatibilidad con Docker, configurado despliegue automático hacia el servidor de *staging*, crear resto de proveedores y entidades, añadir algoritmo de agrupación de resultados e implementar las fuentes externas en el motor de búsqueda.

- **Tercer incremento (v0.3.0-alpha)**: crear métodos de combinación de entidades tras haber implementado el algoritmo de agrupación, añadir soporte para portadas o vistas previas de entidades, y crear interfaces gráficas de inicio y resultados de búsqueda.

- **Cuarto incremento (v0.4.0-alpha)**: crear normalizador, añadir soporte para Wikidata, mapeado de entidades para Doctrine ORM y servicio de caché.

- **Quinto incremento (v0.5.0-alpha)**: añadir página de detalles de una entidad con soporte para plantillas jerárquicas, crear iconos de resultados sin vista previa e implementar soporte para internacionalización (principalmente traducciones).

- **Sexto incremento (v1.0.0-beta)**: último incremento previsto, con duración hasta el final del proyecto, pensado para añadir las últimas funcionalidades y otras no previstas. Incluye crear servicio de mapas.

Todos estos incrementos incluyen sus respectivas fases de análisis y diseño en base a los resultados de la fase anterior, y de pruebas con respecto a la fase de implementación de la iteración actual, así como arreglar fallos.

1.4.1. Herramientas y tecnologías utilizadas

Dentro de este subapartado podemos distinguir entre herramientas utilizadas para poder lograr el objetivo final del proyecto (como IDEs, VCS[8], editores de texto, etc.) y tecnologías que hacen funcionar el producto final (*frameworks*, lenguajes y librerías).

Las herramientas utilizadas son las siguientes:

- **PhpStorm**: IDE desarrollado por la compañía JetBrains especializado en proyectos web que utilicen PHP, también compatible con Node.js y otras tecnologías similares. Es software propietario aunque ofrece licencias gratuitas a estudiantes universitarios.

[8] *Version Control Systems*

- **GitHub**: plataforma online para el control de versiones de proyectos de software mediante Git. También ofrece otras funcionalidades como gestión de tareas o pizarras Kanban.

- **Travis CI**: plataforma de integración continua en la nube compatible con GitHub que permite automatizar procesos de *testing* y despliegue, entre otros. Es gratuito para proyectos de código abierto y estudiantes universitarios.

- **Codefactor**: herramienta online de revisión de código automatizada para controlar la calidad del código de un producto de software y detectar vulnerabilidades y errores, como anti-patrones. Cada vez que hay cambios en el repositorio central del proyecto en GitHub analiza los ficheros y otorga una puntuación que representa la calidad del producto.

- **Dependabot**: complemento de GitHub que, al añadirlo a un proyecto y configurarlo, crear *pull-requests* automáticos cuando existan actualizaciones de dependencias para evitar paquetes obsoletos o fallos de seguridad.

- **Scaleway**: proveedor de servicios en la nube utilizado para alojar una instancia virtual de una máquina con Debian 9 que sirva de servidor de *staging*.

- **Texmaker**: editor de texto para documentos de TeX con vista previa integrada y gestión de proyectos distribuidos en múltiples ficheros. Utilizado para escribir la memoria del TFG.

- **Draw.io**: editor de diagramas gratuito con versión en la nube y de escritorio que permite exportar las figuras finales a múltiples formatos, entre ellos PDF, facilitando la inserción de imágenes vectoriales en documentos TeX.

Las tecnologías empleadas son las siguientes:

- **PHP 7**: lenguaje de programación multi-paradigma pensado para el desarrollo de aplicaciones web.

- **JavaScript (ES6)**: lenguaje de programación interpretado utilizado para el desarrollo web tanto en el lado del cliente como en el lado del servidor. ECMAScript 2016 (ES6) es su versión más reciente.

- **SASS**: lenguaje de hoja de estilos basado en CSS que añade nuevas funcionalidades a este último, como variables que resuelve en tiempo de compilación o anidamiento de reglas.

- **Node.js**: entorno de ejecución basado en ECMAScript habitualmente utilizado como *backend* de aplicaciones web. Para este proyecto se emplea para compilar Rosetta usando Webpack.

- **Symfony 4**: *framework* de PHP para el desarrollo de aplicaciones web basado en un conjunto de componentes reutilizables denominados *bundles* que buscan obtener el mayor rendimiento posible incluso en máquinas con prestaciones bajas.

- **Webpack 4**: Librería para Node.js basada en módulos que permite compilar recursos estáticos de una aplicación y minificarlos (comprimirlos).

- **Bootstrap 4**: kit de herramientas (*toolkit*) para el diseño de interfaces gráficas de aplicaciones web programado en SASS, JavaScript y HTML. Su repositorio oficial dispone de un paquete de Node.js y es fácilmente integrable con Webpack.

- **YAZ**: extensión de PHP para la conexión a servidores de Z39.50/SRW/SRU basado en el YAZ toolkit de la empresa Index Data. Utilizado para conectar con fuentes de datos usando el protocolo Z39.50.

- **Docker**: plataforma de virtualización de software que permite la creación y despliegue de paquetes denominados "contenedores" facilitando la portabilidad y la creación de entornos de ejecución de aplicaciones.

- **nginx**: servidor web recomendado para ejecutar Rosetta y el utilizado por defecto en la imagen de Docker Compose del proyecto. Ofrece un alto rendimiento sin necesidad de un consumo de recursos excesivo.

1.4.2. Ciclo de trabajo

La metodología empleada durante el desarrollo del producto de este trabajo se basa en una versión simplificada del *gitflow workflow*, consistente en crear *commits* de Git en una rama de desarrollo llamada "develop" y en hacer un *merge* de esa rama hacia "master" con cada nueva versión o *release*. Gracias a este flujo, se pueden automatizar procesos con relativa facilidad, como la puesta en pre-producción (*staging*) o el control de calidad.

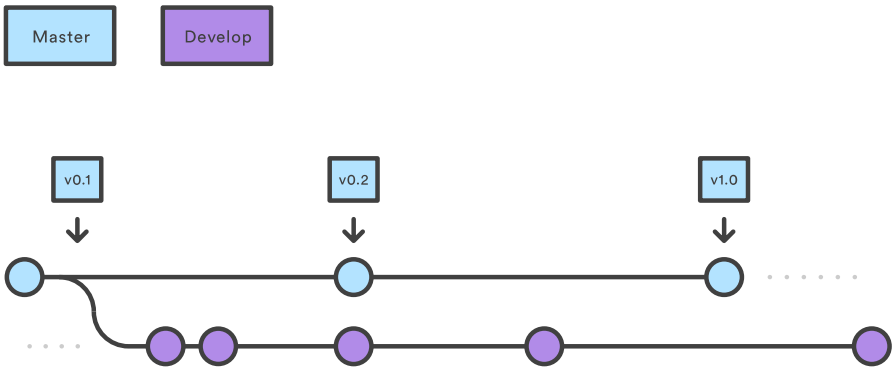


Figura 1.7: Esquema de ramas de *gitflow workflow* [5]

En el caso de este trabajo, el control de calidad es realizado por la plataforma Codefactor[9], que cada vez que se envían *commits* a una rama analiza los cambios y otorga al repositorio una puntuación sobre diez que determina la calidad del código (a mayor puntuación, mayor calidad).

Esta herramienta se puede configurar para detectar posibles fallos de seguridad, métodos complejos, anti-patrones y *code-smells*. A fecha de entrega del producto final, la puntuación otorgada por Codefactor es de 9,9 sobre 10 (grado A).

En cuanto a las pruebas y el despliegue automático, se ha utilizado Travis CI[10] para ejecutar una serie de comandos cada vez que se detectaran cambios en el repositorio central del proyecto. Entre las tareas realizadas por la herramienta se encuentran validación de dependencias, ficheros de configuración, plantillas y mapeos de Doctrine ORM, búsqueda de vulnerabilidades, verificación de traducciones, y ejecución de tests unitarios. Adicionalmente, al detectar cambios en la rama "master" Travis CI realiza un despliegue hacia el servidor de *staging*.

Es relevante mencionar que para poder probar la aplicación con el catálogo de Almena de la Universidad de Valladolid fue necesario establecer un servidor VPN en las instalaciones de la Escuela de Ingeniería Informática de Segovia (previa autorización de Secretaría General), ya que el servidor Z39.50 de Almena autentica mediante dirección IP en vez de par usuario-contraseña.

1.4.3. Planificación

El proyecto se inicia en febrero de 2019 y se estima finalice a principios de mayo (misma duración que el cuatrimestre en la Escuela de Ingeniería Informática de Segovia), contando con una sola persona. No se estima una duración superior a cuatro meses pese a lo innovador del proyecto debido a la experiencia adquirida con Bibliozambrano en los años anteriores y al conocimiento de antemano de las librerías y herramientas a usar.

Se espera trabajar a jornada completa (8 horas diarias) de lunes a viernes, con posibilidad de recuperar horas o trabajo los fines de semana. El diagrama de Gantt estimado del proyecto es el mostrado en la figura 1.8, en el que se aprecia una fecha inicial del 4 de febrero de 2019 con un final previsto el 6 de mayo de 2019.

Las tareas de planificación se dividen, principalmente, en tres grandes grupos: **formalización**, **prototipado** y **fase incremental**. Del último grupo de tareas, es destacable mencionar la sexta iteración que posee una duración estimada de aproximadamente un mes (cuatro semanas), en la que se pretende acomodar nuevas funcionalidades no previstas, afinar determinados detalles del producto final y completar la memoria de este trabajo.

[9] Véase `https://www.codefactor.io`
[10] Véase `https://travis-ci.com`

Si comparamos la evolución real del proyecto (figura 1.9) con el diagrama de Gantt estimado, se aprecian retrasos en su formalización a la hora de obtener permiso de la Secretaría General de la Universidad de Valladolid para acceder al servidor Z39.50 de Almena y poder probar Rosetta en un entorno *pseudo*-real. Por suerte, esta incidencia no supone un aumento en la duración total del proyecto.

Lo mismo sucede en la fase de prototipado en lo relativo a la obtención de manuales de uso y de desarrollo de Millennium ILS®. Lamentablemente, no es posible obtener acceso a estos últimos pero sí a los primeros, lo que retrasa dicha tarea en más de una semana sin afectar a la duración de la fase.

Por último, los dos retrasos que sí provocan un aumento de la duración total del proyecto son el segundo y el cuarto incremento de la fase incremental. En el caso del segundo incremento se debe a la falta de documentación y formación previa para el despliegue de la aplicación como contenedor de Docker.

En lo relativo al cuarto incremento, se alarga una semana más de lo previsto debido a la necesidad de implementar un tipo de relación especial para el caché de entidades que Doctrine ORM no proporciona de forma nativa y que no estaba contemplado desde un principio (véase página 48 para más información sobre la implementación final).

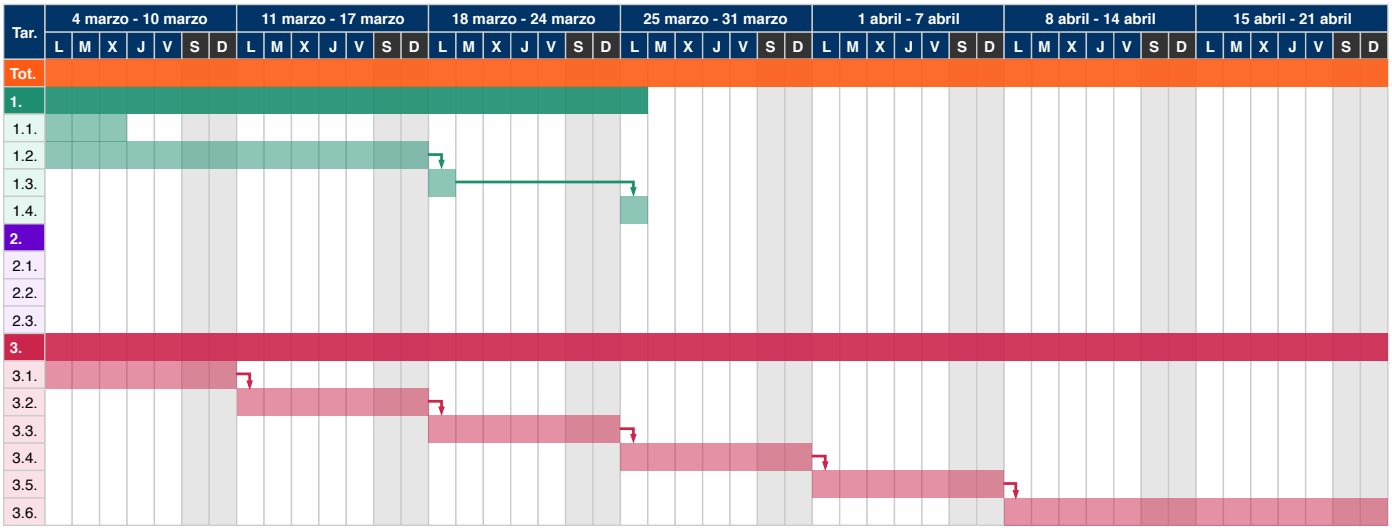
Estimación de horas

Tomando como referencia el diagrama de Gantt estimado, se obtiene un total de 13 semanas que, a razón de 5'5 días/semana, equivalen a 71'5 días o **572 horas**, una cifra más que acertada teniendo en cuenta que un Trabajo Fin de Grado se valora en 24 ECTS[11], o lo que es lo mismo, 600 horas.

Fijándonos en la distribución real del trabajo, se han empleado 15 semanas para terminar el proyecto que, siguiendo los mismos factores de conversión del caso anterior (mismos días por semana a jornada completa) obtenemos un total de **660 horas** reales (11 días más de lo previsto).

[11] *European Credit Transfer and Accumulation System*, equivale a 25 horas de trabajo

Nombre de la tarea	Fecha de inicio	Fecha de fin
Completar la ejecución del proyecto	**04/02/2019**	**05/05/2019**
1. Formalización y preparación del proyecto	**04/03/2019**	**25/03/2019**
1.1. Despliegue servidor de *staging*	04/03/2019	06/03/2019
1.2. Obtener autorización de Secretaría General	04/03/2019	17/03/2019
1.3. Envío autorización de acceso a biblioteca UVa	18/03/2019	18/03/2019
1.4. Despliegue servidor VPN para acceso Almena	25/03/2019	25/03/2019
2. Fase de prototipado	**04/02/2019**	**03/03/2019**
2.1. Obtención de manuales de Millennium ILS	04/02/2019	17/02/2019
2.2. Estudiar y recabar información de manuales	11/02/2019	03/03/2019
2.3. Diseño de prototipos	18/02/2019	03/03/2019
3. Fase incremental	**04/03/2019**	**05/05/2019**
3.1. Primer incremento (v0.1.0-alpha)	04/03/2019	10/03/2019
3.2. Segundo incremento (v0.2.0-alpha)	11/03/2019	17/03/2019
3.3. Tercer incremento (v0.3.0-alpha)	18/03/2019	24/03/2019
3.4. Cuarto incremento (v0.4.0-alpha)	25/03/2019	31/03/2019
3.5. Quinto incremento (v0.5.0-alpha)	01/04/2019	07/04/2019
3.6. Sexto incremento (v1.0.0-beta)	08/04/2019	05/05/2019



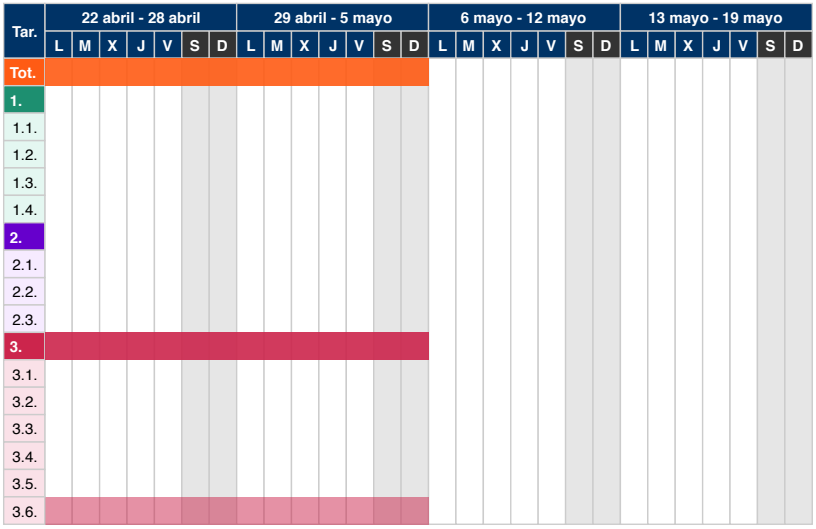
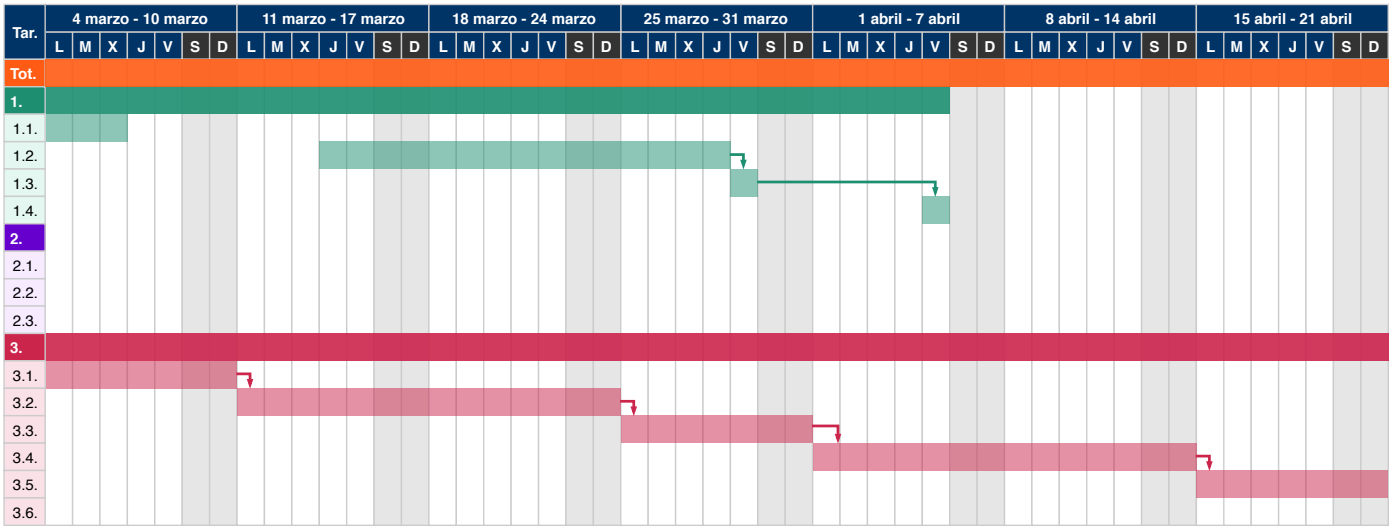


Figura 1.8: Diagrama de Gantt estimado

Nombre de la tarea	Fecha de inicio	Fecha de fin
Completar la ejecución del proyecto	**04/02/2019**	**19/05/2019**
1. Formalización y preparación del proyecto	**04/03/2019**	**05/04/2019**
1.1. Despliegue servidor de *staging*	04/03/2019	06/03/2019
1.2. Obtener autorización de Secretaría General	14/03/2019	28/03/2019
1.3. Envío autorización de acceso a biblioteca UVa	29/03/2019	29/03/2019
1.4. Despliegue servidor VPN para acceso Almena	05/04/2019	05/04/2019
2. Fase de prototipado	**04/02/2019**	**03/03/2019**
2.1. Obtención de manuales de Millennium ILS	04/02/2019	27/02/2019
2.2. Estudiar y recabar información de manuales	11/02/2019	03/03/2019
2.3. Diseño de prototipos	18/02/2019	03/03/2019
3. Fase incremental	**04/03/2019**	**19/05/2019**
3.1. Primer incremento (v0.1.0-alpha)	04/03/2019	10/03/2019
3.2. Segundo incremento (v0.2.0-alpha)	11/03/2019	24/03/2019
3.3. Tercer incremento (v0.3.0-alpha)	25/03/2019	31/03/2019
3.4. Cuarto incremento (v0.4.0-alpha)	01/04/2019	14/04/2019
3.5. Quinto incremento (v0.5.0-alpha)	15/04/2019	21/04/2019
3.6. Sexto incremento (v1.0.0-beta)	22/04/2019	19/05/2019



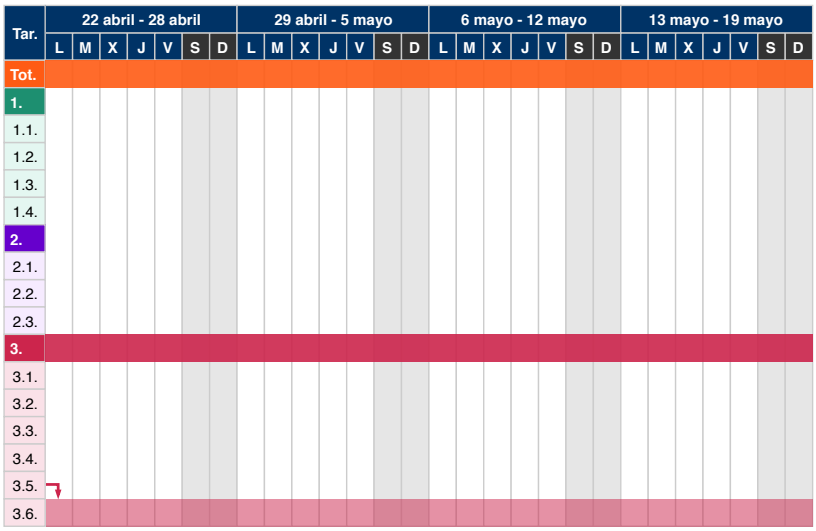


Figura 1.9: Diagrama de Gantt real

1.4.4. Presupuesto

A partir de las estimaciones realizadas en el apartado anterior se calcula el coste total del proyecto, o mejor dicho, el **valor monetario del proyecto**, pues en la práctica no se ha obtenido financiación real para su ejecución.

Rosetta es un proyecto de código abierto, software libre y sin ánimo de lucro realizado por una sola persona. De tener que poner un precio al trabajo realizado se podrían desglosar los costes en tres grupos: hardware, servicios, licencias de software y nóminas (recursos humanos).

Elemento	Coste (€)	Uso (%)	Total (€)
Ordenador de sobremesa	1100	5	55
Ordenador portátil	200	3	6

Tabla 1.2: Costes de hardware

Elemento	Coste (€/mes)	Uso (%)	Duración (meses)	Total (€)
Conexión a Internet	80	10	3	24
Servidor de *staging*	3	100	2,07	6,21
GitHub	0	100	3	0
Dependabot	0	100	3	0
Travis CI	0	100	3	0
Codefactor	0	100	3	0

Tabla 1.3: Costes de servicios

Elemento	Licencia	Coste (€)	Uso (%)	Total (€)
PhpStorm	Student License	0	100	0
Texmaker	GPLv2	0	100	0
draw.io Desktop	Apache 2.0	0	100	0

Tabla 1.4: Costes de licencias de software

Para el cálculo de las nóminas se establece un sueldo medio bruto, sin contar con las cotizaciones a la Seguridad Social, de 20.000 €/año para el desarrollador y 25.000 €/año para el analista con 4 horas diarias para cada rol durante 71'5 días (13 semanas).

Rol	Coste (€/hora)	Duración (horas)	Total (€)
Analista	13,02	286	3.723,72
Desarrollador	10,42	286	2.980,12

Tabla 1.5: Costes estimados de nóminas

El coste total estimado del proyecto sería, por tanto, de **6.795,05 €** brutos:

	% del total	Total (€)
Hardware	0,90	61
Servicios	0,44	30,21
Licencias de software	0,00	0
Nóminas	98,66	6.703,84
	100,00	6.795,05

Tabla 1.6: Desglose total de costes estimados

Teniendo en cuenta la variación del proyecto que aumenta en dos semanas su duración real, los costes de nóminas se ven incrementos hasta alcanzar las cifras que aparecen en las siguientes tablas (se ignora la variación de costes de servicios al ser insignificante):

Rol	Coste (€/hora)	Duración (horas)	Total (€)
Analista	13,02	330	4.296,60
Desarrollador	10,42	330	3.438,60

Tabla 1.7: Costes reales de nóminas

En base a estos datos, el sobrecoste final del proyecto es de **1.031,36 €**:

	% del total	Variación (€)	Total (€)
Hardware	0,78	0	61
Servicios	0,39	0	30,21
Licencias de software	0,00	0	0
Nóminas	98,83	1.031,36	7.735,20
	100,00	1.031,36	7.826,41

Tabla 1.8: Desglose total de costes reales

1.5. Contenidos del CD

En la raíz del CD correspondiente a este Trabajo Fin de Grado se encuentran los siguientes ficheros:

- `memoria.pdf`: versión en PDF de este documento
- `memoria.zip`: archivo comprimido con el proyecto de LaTeX para la compilación del documento PDF de la memoria
- `rosetta.zip`: archivo comprimido con el código fuente completo de la última versión de Rosetta a fecha de grabar el disco
- `sha1sum`: hashes en SHA-1 de los ficheros del CD
- `sha512sum`: hashes en SHA-512 de los ficheros del CD

CAPÍTULO

2

ANÁLISIS

2.1. Características del sistema

Las características de un sistema son las funcionalidades básicas y principales con las que éste debe contar para poder cumplir con los objetivos del proyecto.

- **CAR-01.1 - Obtener resultados primarios**: devuelve el listado de entidades o resultados de una búsqueda para las fuentes de datos primarias.
- **CAR-01.2 - Obtener resultados secundarios**: igual que la CAR-01.1 pero con las fuentes de datos secundarias, buscando a partir de los resultados obtenidos anteriormente.
- **CAR-01.3 - Agrupar resultados**: cataloga las entidades obtenidas en una búsqueda en varios grupos a partir de identificadores comunes.
- **CAR-01.4 - Combinar resultados**: fusiona las entidades de los grupos obtenidos en la CAR-01.3 en una sola entidad por grupo.
- **CAR-02.1 - Leer configuración**: obtiene la configuración de la aplicación de los ficheros almacenados en disco.
- **CAR-02.2 - Obtener versión**: obtiene la versión actual de la aplicación.

- **CAR-03.1 - Buscar entidades**: consulta Wikidata para obtener las entidades correspondientes a una determinada consulta.
- **CAR-03.2 - Rellenar entidades**: completa los atributos de las entidades de la aplicación con la información obtenida de las entidades de Wikidata.
- **CAR-03.3 - Parsear atributos**: interpreta los atributos devueltos por la API de Wikidata que se encuentran codificados en el formato propio de este servicio.
- **CAR-04.1 - Persistir entidades**: guarda los resultados de una búsqueda en caché, discerniendo entre entidades existente a modificar y nuevas entidades que se deben crear.
- **CAR-05.1 - Generar índice**: crea un archivo con la relación de códigos UDC y su correspondiente mapa por motivos de rendimiento.
- **CAR-05.2 - Obtener mapa de ejemplar**: devuelve el nombre del mapa correspondiente a un ejemplar.
- **CAR-05.3 - Renderizar mapa**: genera una imagen del mapa solicitado.

2.1.1. Árbol de características

A continuación se incluye un diagrama con todas las características de la aplicación mencionadas en el apartado anterior para visualizar el alcance del proyecto con mayor facilidad:

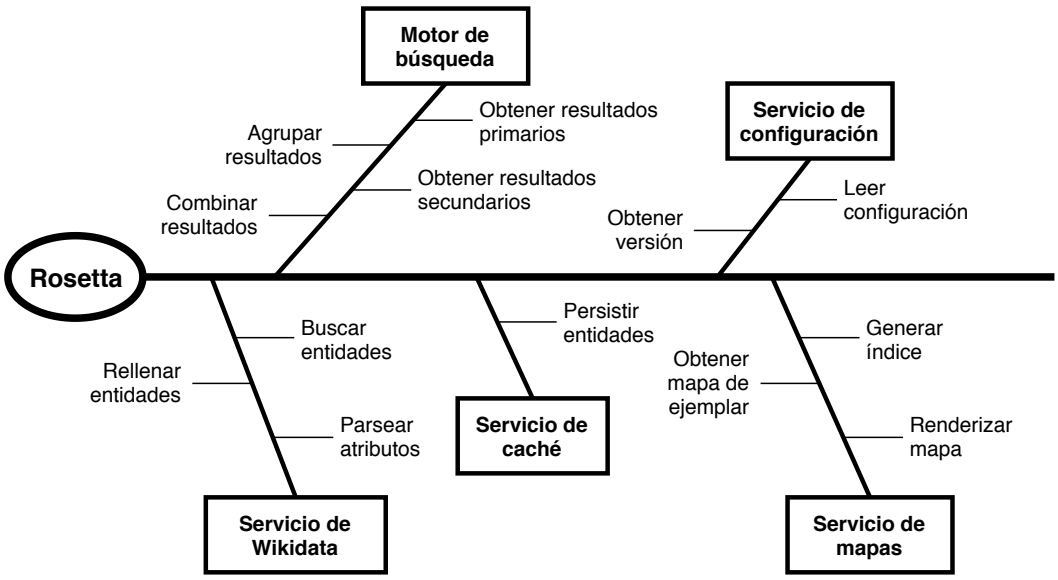


Figura 2.1: Árbol de características

2.2. Actores del sistema

Se consideran actores a aquellos agentes externos al sistema, tanto personas como otras máquinas, que interactúan con él.

Partiendo de esta definición, los actores del software desarrollado para este TFG son los siguientes:

- **Usuario**: cliente que se conecta al OPAC para realizar búsquedas o consultar la información de una entidad.
- **Fuente de datos**: servicio de terceros al que se conecta Rosetta a través de un proveedor para obtener los resultados de una búsqueda. Puede haber (y es lo habitual) múltiples actores de este tipo, pues cada servidor externo con el que se conecta la aplicación se representa de esta forma.
- **Wikidata**: plataforma libre y gratuita ofrecida por la Wikimedia Foundation que permite buscar y obtener metadatos y relaciones de un concepto o entidad.

Nótese que no existe ninguna jerarquía entre estos actores: son todos independientes y ninguno se comunica directamente con el otro.

2.3. Requisitos de usuario

Todos los requisitos de usuario del proyecto se corresponden con los del cliente que se conecta a través de la interfaz web (las fuentes de datos y Wikidata no definen enunciados que puedan recogerse en este apartado).

- **UR-01**: un usuario podrá realizar búsquedas desde el OPAC
- **UR-02**: un usuario podrá ver los detalles de una entidad
- **UR-03**: un usuario podrá ver la ubicación de un ejemplar
- **UR-04**: un usuario podrá acceder a los datos de una entidad en otra página externa

2.3.1. Diagrama de casos de uso

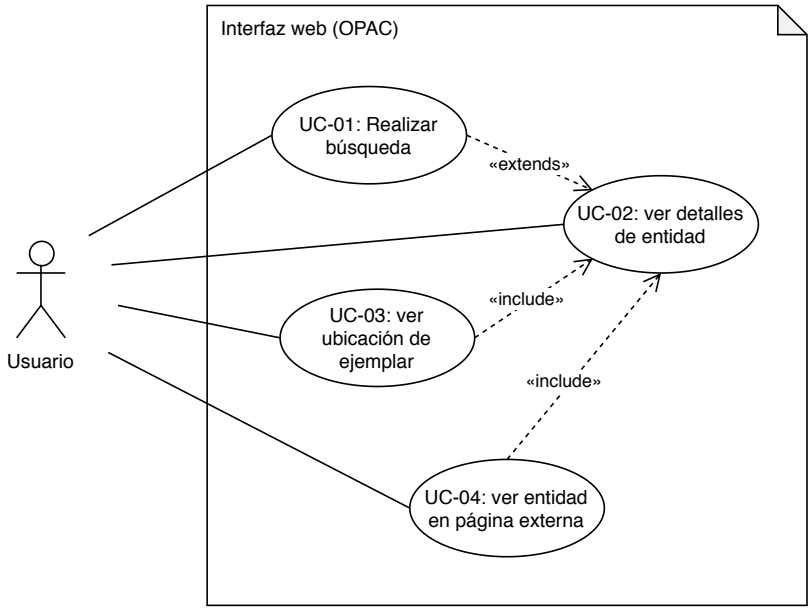


Figura 2.2: Diagrama de casos de uso de usuario

Como se ve en la figura 2.2, el usuario del OPAC puede realizar una búsqueda en la plataforma (UC-01). Habitualmente, una vez completada la búsqueda, el usuario elegirá un resultado para ver más información (UC-02) aunque también es posible acceder a la página de detalles de una entidad desde su dirección URL.

Desde la página de detalles, el usuario puede hacer click sobre un ejemplar para obtener una vista previa del mapa con su ubicación (UC-03) o pulsar en uno de los enlaces externos para ver la información sobre la entidad en otra fuente (UC-04).

2.3.2. Especificación de los casos de uso

CU-01	**Realizar búsqueda**
Versión	1.0
Autor	José Miguel Moreno López
Requisitos asociados	UR-01: un usuario podrá realizar búsquedas desde el OPAC
Actor	Usuario
Descripción	Un usuario puede iniciar una nueva búsqueda en la plataforma web a partir de una consulta y de unos filtros, estos últimos opcionales.
Precondiciones	N/A
Flujo normal	1. El usuario accede a la página de inicio o a la página de resultados de una búsqueda. 2. El usuario escribe la consulta de búsqueda en la barra de búsqueda. 3. Opcionalmente, el usuario filtra por base de datos del desplegable de la barra de búsqueda. 4. El usuario hace click en "Buscar" o pulsa INTRO en el teclado. 5. El sistema muestra los resultados de la búsqueda.
Postcondiciones	N/A
Excepciones	No hay resultados de búsqueda: se mostrará un mensaje de error animando al usuario a utilizar otros términos de búsqueda.
Frecuencia	Alta: la interfaz web tiene como principal objetivo permitir al usuario realizar búsquedas en el catálogo
Importancia	Alta
Prioridad	Alta
Observaciones	N/A

Tabla 2.1: Caso de uso "realizar búsqueda" (CU-01)

CU-02	**Ver detalles de entidad**
Versión	1.0
Autor	José Miguel Moreno López
Requisitos asociados	UR-02: un usuario podrá ver los detalles de una entidad
Actor	Usuario
Descripción	El usuario podrá consultar los detalles de una entidad, como sus ejemplares, relaciones con otras entidades, etc.
Precondiciones	N/A
Flujo normal	Existen dos posibles flujos para este caso de uso: 1a. El usuario hace click sobre uno de los resultados de una búsqueda. 1b. El usuario abre un enlace ya conocido de una entidad. 2. El sistema muestra la página de detalles de la entidad solicitada.
Postcondiciones	N/A
Excepciones	No existe la entidad o ha sido eliminada: se mostrará un mensaje de error animando al usuario a utilizar el buscador.
Frecuencia	Alta: la consecuencia de una búsqueda suele ser ver la página de detalles
Importancia	Alta
Prioridad	Alta
Observaciones	N/A

Tabla 2.2: Caso de uso “ver detalles de entidad” (CU-02)

CU-03	**Ver ubicación de ejemplar**
Versión	1.0
Autor	José Miguel Moreno López
Requisitos asociados	UR-03: un usuario podrá ver la ubicación de un ejemplar
Actor	Usuario
Descripción	Para aquellos ejemplares cuya ubicación sea conocida, la aplicación mostrará un mapa con su ubicación física.
Precondiciones	El usuario debe estar en la página de detalles de una entidad. La entidad debe ser prestable o consultable.
Flujo normal	1. El usuario hace click sobre el ejemplar del listado de ejemplares que quiere consultar. 2. El sistema muestra en la columna de la derecha de la pantalla el mapa con las estanterías en las que se encuentra el ejemplar resaltadas.
Postcondiciones	N/A
Excepciones	No hay ejemplares con ubicación conocida: en este caso no se renderizará ningún mapa.
Frecuencia	Media: no todas las entidades tiene ejemplares ni todos los ejemplares tienen ubicación conocida
Importancia	Alta
Prioridad	Media
Observaciones	El sistema mostrará un cursor especial (una mano) para dar feedback al usuario sobre cuándo es posible hacer click en un ejemplar para renderizar su mapa.

Tabla 2.3: Caso de uso “ver ubicación de ejemplar” (CU-03)

CU-04	**Ver entidad en página externa**
Versión	1.0
Autor	José Miguel Moreno López
Requisitos asociados	UR-04: un usuario podrá acceder a los datos de una entidad en otra página externa
Actor	Usuario
Descripción	Un usuario puede acceder a las fuentes externas que utilizó la aplicación para generar la página de detalles de una entidad.
Precondiciones	El usuario debe estar en la página de detalles de una entidad.
Flujo normal	1. El usuario hace click sobre el nombre del proveedor externo de la lista de enlaces externos al que quiera acceder. 2. El sistema abre en una nueva pestaña/ventana el enlace externo.
Postcondiciones	N/A
Excepciones	No hay enlaces externos: en este caso no se renderizará la sección de enlaces externos.
Frecuencia	Baja: no es habitual que un usuario quiera ver la misma información desde otra fuente en este contexto
Importancia	Baja
Prioridad	Media
Observaciones	N/A

Tabla 2.4: Caso de uso “ver entidad en página externa” (CU-04)

2.4. Requisitos de información

Los requisitos de información son aquellos conjuntos y especificaciones de datos necesarias para que el sistema puedan funcionar o para la correcta finalización del proyecto.

- **IR-01**: el sistema almacenará en caché las entidades que sean resultado de las búsquedas
- **IR-02**: el sistema permitirá guardar un registro de error y de eventos
- **IR-03**: el sistema almacenará la configuración de la aplicación
- **IR-04**: el sistema guardará versiones compiladas de las vistas en un directorio de almacenamiento temporal

2.4.1. Diagrama entidad-relación

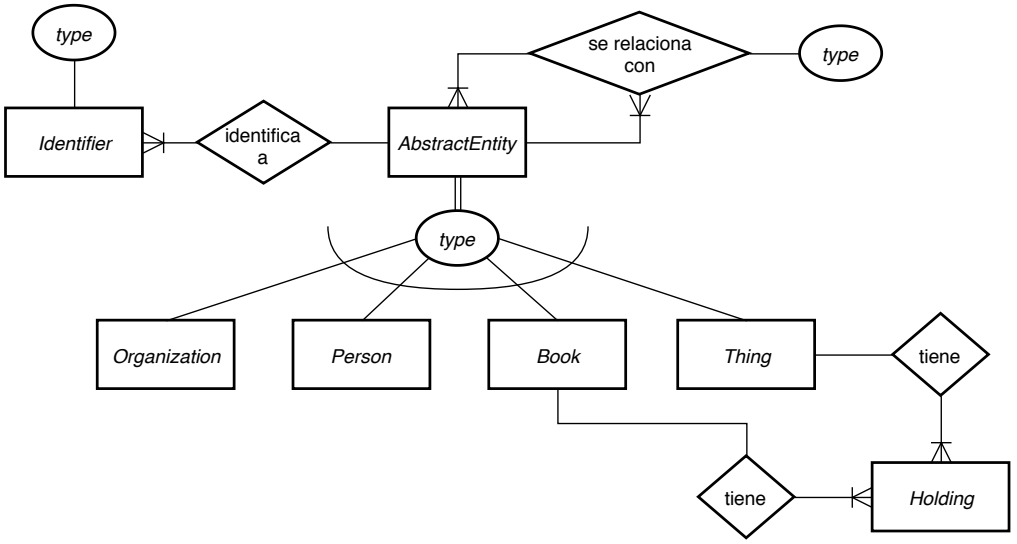


Figura 2.3: Diagrama entidad-relación

Pese a la complejidad de la aplicación, el diagrama E/R de su base de datos es relativamente sencillo. La principal entidad es "AbstractEntity", de la que parten casi todas las demás ("Organization", " Person", "Book" y "Thing"). Esta entidad tiene un discriminante que determina el tipo de hijo, por tanto, en la práctica solo se utilizará una tabla para almacenar esta información.

Tanto "Book" como "Thing" puede tener varios ejemplares (se relacionan con la entidad "Holding"), que a efectos de lógica de aplicación se implementa con un *trait*. Cualquier hijo de "AbstractEntity" se indexa a través de sus relaciones con la entidad "Identifier" y puede relacionarse con otros hijos de la misma entidad.

2.4.2. Diccionario de datos

El producto final de este trabajo no necesita de un análisis previo de los requisitos de información relativos al diccionario de datos. El motivo es que este tipo de análisis está obsoleto a día de hoy pues solo tiene sentido cuando se trabaja con bases de datos relacionales.

Aunque Rosetta utiliza por defecto el motor MariaDB (basado en MySQL) nunca llegar a tocar lógica a tan bajo nivel, sino que existe un agente intermedio (Doctrine ORM) que traduce una jerarquía de clases (orientado a objetos) a sentencias SQL y viceversa. Rosetta es, por tanto, *database agnostic*: desconoce la base de datos que se encuentra por debajo y le es irrelevante.

Aún así, en este apartado se incluye la definición del diccionario de datos que Doctrine ORM genera por defecto al utilizar una base de datos relacional, no sin antes volver a hacer hincapié en que esta puede cambiar en función de la arquitectura final elegida por el administrador del sistema.

De las tablas que se muestran a continuación, cada una de ellas se corresponde con una tabla de la base de datos e incluye la siguiente información:

- **Atributo**: nombre de la columna en la tabla
- **Tipo**: tipo primario de dato en el que está almacenado el valor de la columna
- **Tamaño**: longitud reservada para almacenar los valores de la columna, pueden ser bytes o dígitos y tratarse de una longitud fija o máxima dependiendo del tipo de dato
- **Flags**: booleanos que especifican propiedades de la columna
    - **PK**: es clave primaria
    - **FK**: es clave foránea
    - **NL**: es *nullable* (puede ser nulo)
    - **UQ**: es *unique* (único)
    - **AI**: es *AUTO_INCREMENT*
    - **US**: es *unsigned* (no admite números negativos)

Atributo	Tipo	Tamaño	PK	FK	NL	UQ	AI	US
id	int	10	✓	-	-	✓	✓	✓
creation_date	datetime	-	-	-	-	-	-	-
modification_date	datetime	-	-	-	-	-	-	-
slug	varchar	300	-	-	-	-	-	-
image_url	varchar	2083	-	-	✓	-	-	-
entity_type	varchar	5	-	-	-	-	-	-
title	varchar	1024	-	-	✓	-	-	-
legal_deposits	longtext	-	-	-	✓	-	-	-
pub_year	smallint	5	-	-	✓	-	-	✓
pub_month	smallint	5	-	-	✓	-	-	✓
pub_day	smallint	5	-	-	✓	-	-	✓
languages	longtext	-	-	-	✓	-	-	-
num_of_pages	smallint	5	-	-	✓	-	-	✓
num_of_volumes	smallint	5	-	-	✓	-	-	✓
name	varchar	255	-	-	✓	-	-	-
foundation_date	date	-	-	-	✓	-	-	-
website	varchar	300	-	-	✓	-	-	-
firstname	varchar	255	-	-	✓	-	-	-
lastname	varchar	255	-	-	✓	-	-	-
description	varchar	256	-	-	✓	-	-	-
birth_date	date	-	-	-	✓	-	-	-
death_date	date	-	-	-	✓	-	-	-
subtitle	varchar	3072	-	-	✓	-	-	-
signature_url	varchar	2083	-	-	✓	-	-	-
birth_place	varchar	256	-	-	✓	-	-	-
death_place	varchar	256	-	-	✓	-	-	-

Tabla 2.5: Diccionario de datos de “entity”

Atributo	Tipo	Tamaño	PK	FK	NL	UQ	AI	US
id	int	10	✓	-	-	✓	✓	✓
entity_id	int	10	-	✓	✓	-	-	✓
call_number	varchar	64	-	-	✓	-	-	-
loanable	tinyint	1	-	-	-	-	-	-
available	tinyint	1	-	-	-	-	-	-
location_name	varchar	128	-	-	✓	-	-	-
online_url	varchar	2083	-	-	✓	-	-	-
source_id	varchar	16	-	-	✓	-	-	-

Tabla 2.6: Diccionario de datos de “holding”

Atributo	Tipo	Tamaño	PK	FK	NL	UQ	AI	US
id	varchar	54	✓	-	-	✓	-	-
entity_id	int	10	-	✓	✓	-	-	✓
type	smallint	5	-	-	-	-	-	✓
value	varchar	50	-	-	-	-	-	-

Tabla 2.7: Diccionario de datos de “identifier”

Atributo	Tipo	Tamaño	PK	FK	NL	UQ	AI	US
type	smallint	5	✓	-	-	✓	-	✓
from_id	int	10	✓	-	-	✓	-	✓
to_id	int	10	✓	-	-	✓	-	✓

Tabla 2.8: Diccionario de datos de “relation”

2.5. Requisitos no funcionales

Non-Functional Requirements (NFR) son aquellas restricciones del sistema que afectan a su desarrollo o comportamiento. Dentro de este grupo las más habituales son los atributos de calidad (QA).

Usabilidad

- **QA-01**: la aplicación web deberá ser fácil de usar
- **QA-02**: la aplicación web cumplirá con los principios de *responsive design*
- **QA-03**: la aplicación informará al usuario de errores de forma clara y sin proporcionar información innecesaria
- **QA-04**: la aplicación web se localizará a inglés de los Estados Unidos (en-US) y español de España (es-ES)

Rendimiento

- **QA-05**: las consultas hacia fuentes de datos externas se paralelizarán en la medida de lo posible
- **QA-06**: los resultados de una búsqueda se guardarán en caché para reducir el número de llamadas externas

Disponibilidad

- **QA-07**: el servidor de *staging* tendrá un SLA de tres nueves (99,9 % de disponibilidad)

Seguridad

- **QA-08**: el sistema sanetizará todas las entradas proporcionadas por el usuario
- **QA-09**: los datos sensibles de configuración se guardarán como variables de entorno
- **QA-10**: las consultas a la base de datos se escribirán en DQL (Doctrine Query Language) para evitar ataques de inyección

Escalabilidad

- **QA-11**: el producto final se dividirá en dos módulos (núcleo y OPAC)
- **QA-12**: el núcleo de la aplicación se dividirá en componentes denominados "servicios"
- **QA-13**: la lógica de la aplicación será *stateless* para poder escalarla con mayor facilidad
- **QA-14**: la aplicación será desplegable con Docker y Kubernetes para facilitar su escalabilidad

Restricciones de arquitectura y proyecto

- **NFR-01**: el sistema se desarrollará utilizando el *framework* Symfony 4 como base
- **NFR-02**: la aplicación solo podrá usar dependencias de software libre bajo licencia GPL, MIT o similar[1]
- **NFR-03**: los ficheros de traducción de la aplicación se guardarán por defecto en formato JSON
- **NFR-04**: los ficheros de configuración de la aplicación se guardarán por defecto en formato YAML

Interfaces externas

- **ERQ-01**: el sistema deberá poder comunicarse con servidores que utilicen el protocolo Z39.50 y el formato MARC 21

[1] Esta restricción se debe a que Rosetta se distribuye bajo licencia GPLv3, que exige que todas las piezas de código que use un programa también sean software libre

CAPÍTULO

3

DISEÑO E IMPLEMENTACIÓN

Nota del autor

Dada la complejidad lógica de Rosetta he decido unir los capítulos de diseño e implementación en uno solo para poder explicar el funcionamiento de la aplicación **por componentes** en vez de hacer una división arbitraria que no ayuda a su comprensión.

Tanto la arquitectura lógica de este software como la física son extremadamente flexibles y pueden organizarse de múltiples formas: con un servidor web y un servidor de aplicaciones, o con el servidor HTTP integrado de Symfony, o utilizando solo el *broker* y prescindiendo por completo de la interfaz gráfica, etc.

Consulte el manual de despliegue del capítulo de documentación si necesita más información.

Para facilitar la legibilidad del código, la corrección de fallos y adición de nuevas funcionalidades, la lógica de negocio de Rosetta se estructura siguiendo el *Libro de Buenas Prácticas de Symfony* [7], framework en el que se encuentra programado el proyecto. De esta forma se pretende reducir la curva de aprendizaje para nuevos desarrolladores que quieran desplegar la aplicación o hacer cambios sobre la misma.

Principalmente, la división más clara en la que se organiza el código fuente de la aplicación es *Front-End* y *Back-End*. El núcleo de Rosetta que se encarga de gestionar las búsquedas, conectar con proveedores externos y almacenar y recuperar los resultados en caché (parte de *Back-End*) se recoge en el *bundle*[1] llamado `RosettaBundle`.

Las plantillas, hojas de estilos, controladores y demás recursos necesarios para renderizar las distintas interfaces gráficas (parte de Front-End) se encuentran fueran de este *bundle* en los directorios de la ruta `/src`.

De esta forma, el núcleo de Rosetta queda separado de la interfaz gráfica y otros componentes no esenciales de la plataforma, pudiendo reutilizar `RosettaBundle` en otros proyectos sin tener que copiar todo el repositorio. Es decir, lo que en este documento se conoce como Rosetta son en realidad dos proyectos distintos (`RosettaBundle` y la interfaz gráfica) que cooperan entre ellos para crear la aplicación final.

Paralelamente, el código fuente se agrupa según la funcionalidad que provee en **servicios**, clases *Singleton*[2] que son gestionadas e inyectadas, a su vez, en otros servicios a través del inyector de dependencias de Symfony.

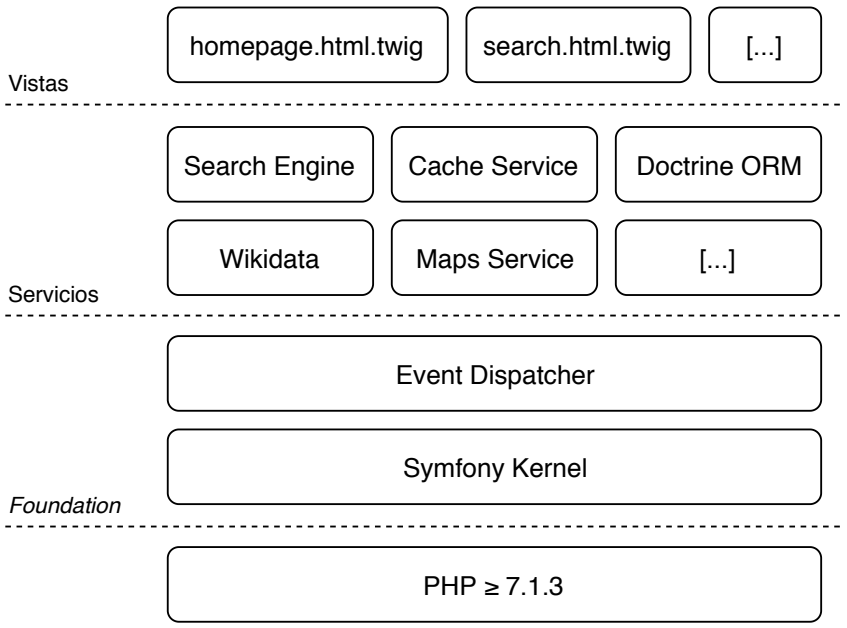


Figura 3.1: *Stack* de componentes de Rosetta

[1] Un *bundle* es una estructura independiente y reutilizable de Symfony que puede ser migrada fácilmente de un proyecto a otro

[2] Un *Singleton* es un patrón de diseño que restringe la creación de objetos de una clase a una única instancia

Según esta definición, un servicio es cualquier componente que pone una serie de métodos públicos a disposición de otros componentes de la aplicación, dotándolos de funcionalidad adicional. Por tanto, en Symfony (y consecuentemente en Rosetta) **casi todas las clases son servicios**.

La inyección de dependencias de Symfony funciona a través de un procedimiento conocido como *autowiring* que consiste en pasar la instancia de los servicios requeridos por otro servicio usando el constructor de este último.

```php
namespace App\RosettaBundle\Service;

use Psr\Log\LoggerInterface;

class SearchEngine {
    private $logger;
    private $config;

    public function __construct(LoggerInterface $logger,
                                ConfigEngine $config) {
        $this->logger = $logger;
        $this->config = $config;
    }

    // [...]
}
```

Código 1: Ejemplo de inyección de dependencias en un servicio

En este ejemplo se ve cómo el servicio `SearchEngine`, que depende a su vez de `LoggerInterface` y de `ConfigEngine` solicita al inyector de dependencias las referencias a las instancias de estos dos servicios, que le son proporcionadas a través del constructor en el momento de su instanciación.

Dado que se hace uso del *type hinting* de PHP 7 (se indica el tipo del argumento en la definición del método), Symfony es capaz de determinar a qué clase nos referimos sin tener que especificarlo explícitamente.

3.1. Entidades

Hasta la fecha, la inmensa mayoría de los sistemas de gestión bibliotecaria han representado la información que almacenaban y gestionaban con una estructura similar a esta:

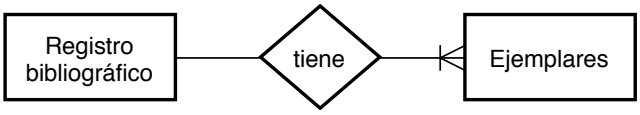


Figura 3.2: Arquitectura simplificada de Millennium® ILS

Un inconveniente de este diseño es que, a efectos prácticos, todos los recursos de los que dispone una autoridad (biblioteca) son libros **no relacionables**. Es decir, a nivel interno, un programa como Innovative Millennium®ILS no conoce otra cosa que no sean libros de texto, de tal forma que para su base de datos una película con un nombre y un director **no es más que un libro con un título y un autor** que tiene la particularidad de ser del tipo "película".

Esto se debe a que un registro bibliográfico no es más que un documento de texto, generalmente estructurado siguiendo el formato MARC 21[3], que contiene los datos de un recurso. Aunque MARC 21 soporta diferentes atributos para los distintos tipos de elementos que puede almacenar (libros, recursos audiovisuales, mapas, partituras, etc.) lo cierto es que son muy limitados a la par que obsoletos para el año en el que vivimos y apenas ningún software bibliotecario los implementa.

Aquí nos encontramos con la primera consecuencia de esta arquitectura: una película es una película, y por mucho que intentemos *mapear* sus atributos a los de un libro lo seguirá siendo. Necesitamos, por tanto, **entidades distintas para representar recursos distintos**.

Otro importante defecto del diagrama entidad-relación de la figura 3.2 es que, si solo existen registros bibliográficos, no disponemos de entidades auxiliares para representar autores, editoriales, ilustradores, etc. y, por tanto, **no podemos relacionar recursos**.

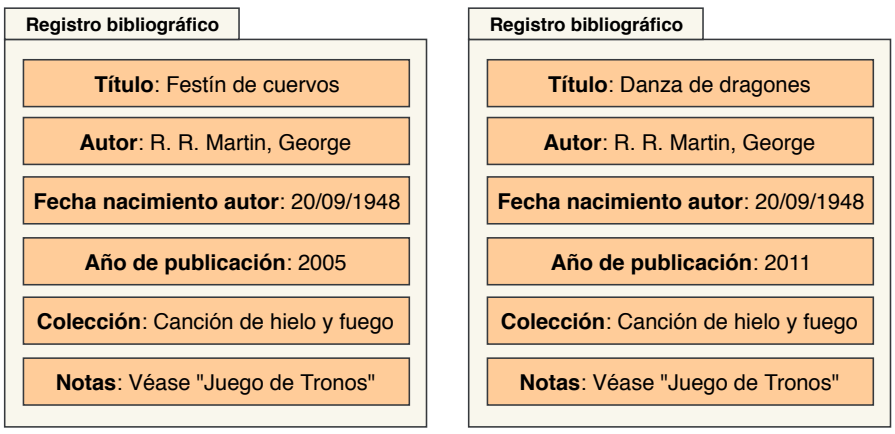


Figura 3.3: Representación habitual de registros

[3] Véase `https://www.loc.gov/marc/holdings/`

Si, por ejemplo, se definiera una entidad "persona" y un conjunto de relaciones con el enunciado "es autor de" en vez de utilizar un campo para almacenar el nombre del autor en cada registro bibliográfico se podrían establecer vínculos entre obras:

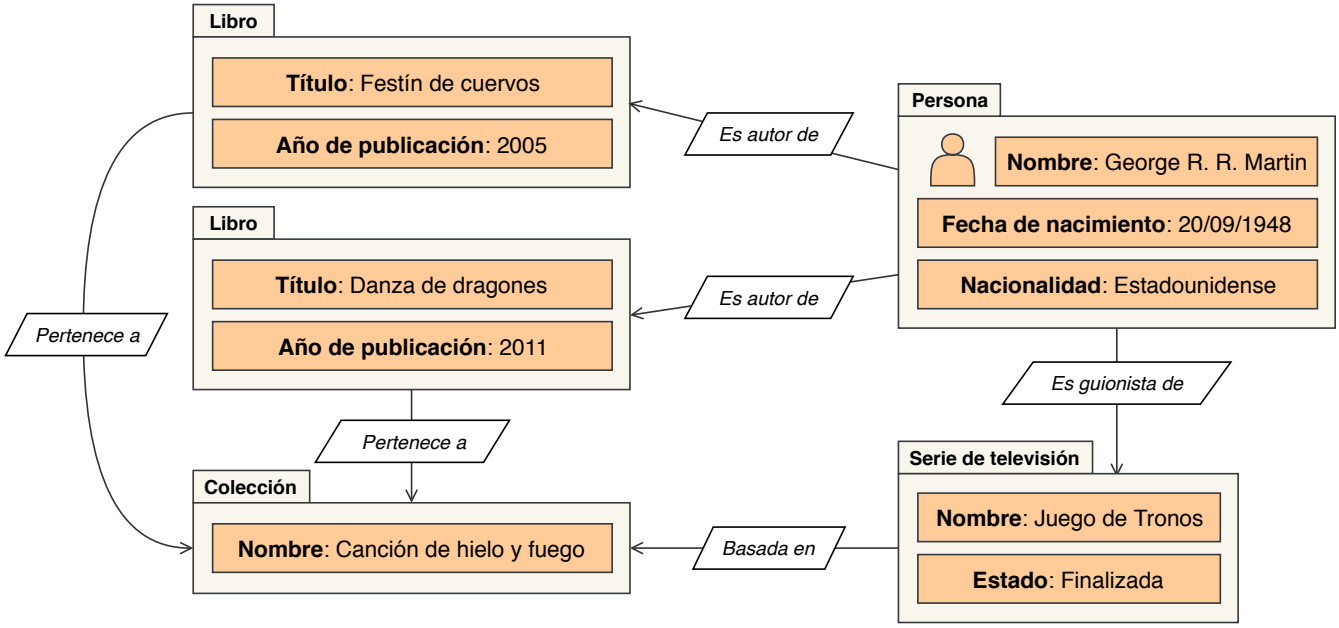


Figura 3.4: Representación relacional de registros

Con Rosetta se propone una estructura jerarquizada de **entidades** que representan todo el conocimiento con capacidad de ser catalogado. Más concretamente, una entidad es cualquier elemento **representable en una base de datos**, que pueda ser mostrado en pantalla y accedido a través de un buscador.

Al establecerse una relación de parentesco entre las entidades, se parte desde el conocimiento más genérico hasta lo concreto (por ejemplo, un libro es un tipo de obra que a su vez es un tipo de entidad).

Además, las entidades se caracterizan por ser relacionables entre sí, lo que permite evitar información duplicada en la base de datos y enriquecer entidades comunes u otras sin tener que modificar todos los elementos implicados.

Con una estructura como la planteada en la figura 3.4 se abre el camino para que las bibliotecas puedan catalogar nuevos tipos de obras y representarlas fielmente sin tener que hacer compromisos debidos a las limitaciones del software gestor, posibilitando conectar con fuentes de datos externas de otros ámbitos de la cultura (como TMDb[4] o Wikidata[5]), enriqueciendo la información que se muestra al usuario final.

[4]The Movie Database es una base de datos de películas y series disponible en `www.tmdb.org`
[5]Wikidata es una base de conocimiento libre creada por la Fundación Wikimedia

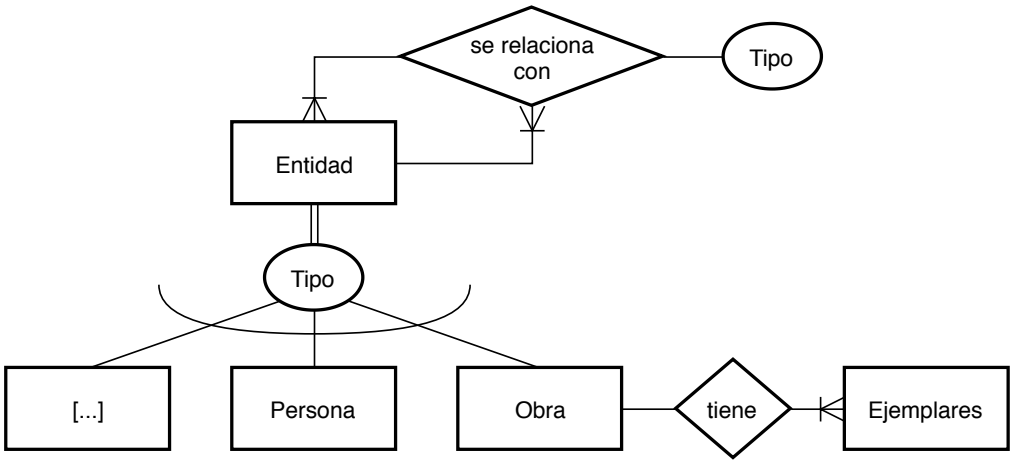


Figura 3.5: Arquitectura simplificada de Rosetta

3.1.1. Jerarquía de entidades

En Rosetta, todas las entidades parten de la clase `AbstractEntity` que, como indica su nombre, es abstracta y por tanto no puede ser instanciada. Un `AbstractEntity` se caracteriza por tener un identificador numérico entero (utilizado con fines de caché) un *slug*[6] y fechas de creación y última modificación.

Las clases que heredan `AbstractEntity` se caracterizan por ser **relacionables** e **identificables** como se verá más adelante en el subapartado 3.1.3. Esta última cualidad juega un papel clave dentro de la aplicación a la hora de buscar información duplicada y relaciones entre entidades.

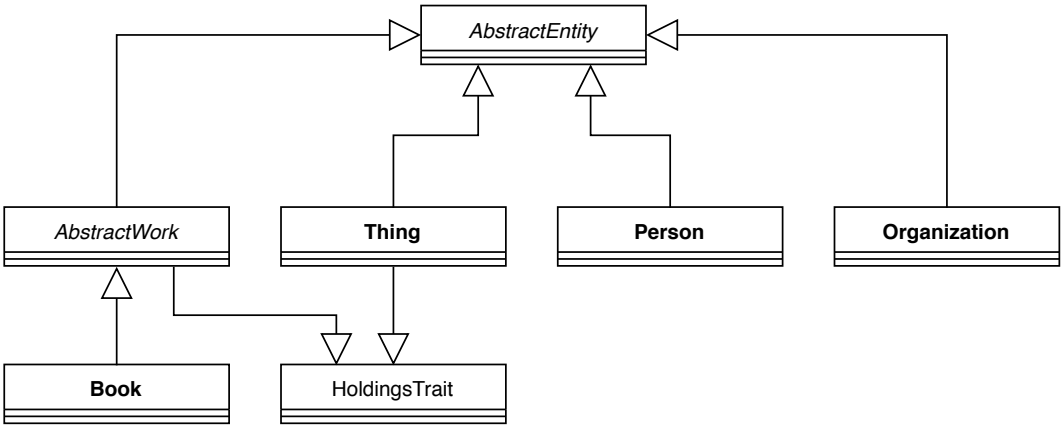


Figura 3.6: Diagrama de clases de entidades

[6]Un *slug* es la parte de una dirección URL que se refiere una página o post específico

Por debajo de `AbstractEntity` encontramos las clases finales `Organization`, `Person` y `Thing`, así como la clase abstracta `AbstractWork`. Una instancia de `Organization` representa a un colectivo o grupo de personas (generalmente en el contexto de bibliotecas hablamos una editorial), mientras que `Person` sirve para guardar la información de un individuo concreto (un autor, ilustrador, fundador de una organización, etc.).

`AbstractWork` es un tipo más concreto de `AbstractEntity` utilizado para representar obras, entendiendo por obra a **cualquier creación artística que puede ser consultada o prestada**. Por defecto, la única entidad final que implementa `AbstractWork` es `Book`, aunque otros tipos de obras que extenderían dicha clase abstracta serían películas, álbumes de canciones y sencillos, artículos científicos, prensa y revistas, etc.

La clase `Thing` es un tanto especial ya que comparte algunas cualidades con `AbstractWork` pese a ser una entidad completamente distinta. Para Rosetta, una "cosa" o `Thing` es un **objeto físico que puede ser consultado o prestado** como, por ejemplo, un portátil o una pizarra que se prestan por horas.

Tanto los hijos de `AbstractWork` como la entidad `Thing` tienen asociados una serie de ejemplares que son los objetos realmente consultables o prestables. Para implementar esta multi-herencia se hace uso del *trait* `HoldingsTrait`, que otorga a una entidad la capacidad de contener ejemplares.

3.1.2. Identificadores

Tal y como se ha mencionado antes, las clases que implementan `AbstractEntity` son identificables. Esto, a afectos de implementación, supone que una entidad abstracta se asocia a una serie de identificadores. Estos identificadores se representan en la clase `Identifier` y almacenan un par tipo-valor: el tipo de identificador y su correspondiente valor.

Por ejemplo, para un libro que se identifica por un código ISBN, la entidad tendría asociado un identificador del tipo "ISBN" con el valor correspondiente al número estándar del libro.

Habitualmente, una entidad posee un identificador para cada tipo o, en algunos casos, varios (una misma edición de un libro puede tener varios ISBN si ésta se encuentra recogida en varios tomos).

Para poder trabajar con identificadores, los cuatro métodos principales que proporciona `AbstractEntity` son:

- `AbstractEntity::getIdentifiers()`: devuelve las instancias de identificadores asociados a la entidad
- `AbstractEntity::addIdentifier($identifier)`: asocia una instancia de identificador a la entidad
- `AbstractEntity::getIdsOfType($type)`: devuelve los valores de los identificadores de la entidad que sean del tipo indicado
- `AbstractEntity::getFirstIdOfType($type)`: igual que el método anterior, devolviendo solo el primer valor

```php
public function getIdentifiers() {
    return $this->identifiers;
}

public function addIdentifier(Identifier $identifier): self {
    $key = (string) $identifier;
    $this->identifiers->set($key, $identifier);
    return $this;
}

public function getIdsOfType(int $type): array {
    $res = [];
    foreach ($this->identifiers as $identifier) {
        if ($identifier->getType() == $type) {
            $res[] = $identifier->getValue();
        }
    }
    return $res;
}

public function getFirstIdOfType(int $type): ?string {
    foreach ($this->identifiers as $identifier) {
        if ($identifier->getType() == $type) {
            return $identifier->getValue();
        }
    }
    return null;
}
```

Código 2: Métodos de identificadores de `AbstractEntity`

Además de estos métodos, determinadas entidades disponen de atajos que solo tienen sentido en el contexto en el que están siendo usados, como en el caso de la entidad `Book` desde la que se puede llamar a `Book::getIsbns()` o `Book::addIsbn(string $isbn)`, entre otros.

Los identificadores y otras estructuras de lista se almacenan en Rosetta utilizando colecciones de Doctrine (`ArrayCollection`) en vez de `arrays`, pues facilita su serialización e instanciación desde y hacia el caché (véase subapartado 3.1.4).

Por defecto, Rosetta provee de los siguientes tipos de identificadores:

- `INTERNAL`: para referirse a la numeración privada de un proveedor (véase apartado 3.2.2)
- `ISBN10`: código ISBN de 10 dígitos
- `ISBN13`: código ISBN de 13 dígitos
- `OCLC`: identificador dentro de la base de datos de WorldCat[7]
- `GBOOKS`: identificador dentro de la base de datos de Google Books
- `WIKIDATA`: identificador del elemento dentro la base de datos de Wikidata

A nivel de caché, los identificadores sirven de índices para buscar a otras entidades o determinar si una entidad dada ya existe en la base de datos (véase subapartado 3.1.4).

3.1.3. Relaciones entre entidades

Todas las entidades son relacionables entre sí. Para cumplir con este enunciado, Rosetta debe ser capaz de representar en su estructura de datos interna frases como "Cervantes es el autor del Quijote" o "el CSIC es la editorial del Archivo Español de Arqueología".

Al abstraernos de la semántica, se puede apreciar que estas frases poseen una estructura del tipo **«sujeto» «acción» sobre «objeto»**, por lo que una relación, almacenada en la clase `Relation`, deberá tener tres partes: una entidad de origen, una entidad de destino y el tipo de relación.

[7] WorldCat es un catálogo en línea de recursos bibliotecarios gestionado por el OCLC (Online Computer Library Center)

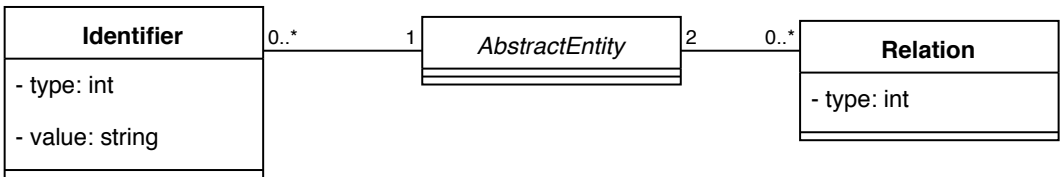


Figura 3.7: Diagrama de clases de `AbstractEntity`

Los tipos de relaciones existentes en la versión Beta de Rosetta son:

- `IS_AUTHOR_OF`: Sujeto es autor de objeto
- `IS_EDITOR_OF`: Sujeto es editor de objeto
- `IS_ILLUSTRATOR_OF`: Sujeto es ilustrador de objeto
- `IS_PUBLISHER_OF`: Sujeto es editorial de objeto
- `IS_FOUNDER_OF`: Sujeto es fundador de objeto

Para facilitar al desarrollador la búsqueda y recorrido de relaciones de una entidad, la aplicación provee de una serie de métodos que reducen las líneas de código necesarias para lograr tal objetivo, siendo el más relevante `Relation::getOther($subject)`:

```
public function getOther($subject) {
    return ($subject === $this->getTo()) ?
        $this->getFrom() :
        $this->getTo();
}
```

Código 3: Método `Relation::getOther($subject)`

Dado que en una relación intervienen dos partes, hay ocasiones en las que nos interesa conocer la otra parte de la misma cuando encontramos en una de ellas. Gracias a este método se generan otros auxiliares en la clase `AbstractEntity`: `getRelatedOfType` y `getFirstRelatedOfType`.

Igual que con los identificadores, a partir de estos dos métodos se crean otros más específicos en las distintas entidades de Rosetta, como en `Book`, que ofrece el atajo `Book::getPublisher()` como sustituto de `AbstractEntity::getFirstRelatedOfType(Relation::IS_PUBLISHER_OF)`, entre otros.

```php
public function getRelatedOfType(int $type): array {
    $res = [];
    foreach ($this->getRelations() as $relation) {
        if ($relation->getType() == $type) {
            $res[] = $relation->getOther($this);
        }
    }
    return $res;
}

public function getFirstRelatedOfType(int $type) {
    foreach ($this->getRelations() as $relation) {
        if ($relation->getType() == $type) {
            return $relation->getOther($this);
        }
    }
    return null;
}
```

Código 4: Métodos de relaciones de `AbstractEntity`

3.1.4. Caché de entidades

Con el objetivo de reducir la transferencia de datos entre Rosetta y los servidores de las bases de datos de los proveedores, la aplicación guarda un caché local de las entidades que han sido buscadas con anterioridad para poder ser recuperado más tarde.

Este proceso se realiza mediante la colaboración de dos servicios, que son `CacheService` (nativo de Rosetta) y `EntityManagerInterface` (perteneciente al proyecto Doctrine ORM).

Mapping de entidades

Antes de explorar cómo consigue la clase `CacheService` crear nuevas entidades en el caché, sobrescribir las existentes y detectar duplicados debemos entender **cómo se almacenan las instancias que heredan de `AbstractEntity`** en la base de datos.

Rosetta utiliza por defecto una base de datos relacional estructurada por tablas para almacenar estas entidades al considerarse la opción más extendida y fácil de implementar, aunque se puede configurar para emplear cualquier otro sistema (por ejemplo, una base de datos no relacional como MongoDB, u otros motores orientados a columnas como ClickHouse si buscamos un alto rendimiento).

Entre las instancias de `AbstractEntity` y la base de datos existe un componente intermedio conocido como *Object-Relational Mapper* (ORM) que se encarga de traducir entre objetos de PHP y sentencias SQL o el lenguaje que usemos para comunicarnos con el gestor de la base de datos.

Más concretamente, el ORM utilizado por Rosetta es **Doctrine ORM** al ofrecer una muy buena integración con Symfony. Cuando Doctrine ORM recibe una entidad para guardar en la base de datos, serializa los atributos para los que ha sido configurado y generar la sentencia SQL correspondiente, que luego ejecuta, y realiza el proceso inverso para recuperar del caché.

La forma en la que el ORM debe guardar los objetos se especifica a través de **anotaciones**[8] en las clases de las entidades. Estas anotaciones son comentarios de PHP que empiezan por el carácter «@» y son leídas por Doctrine ORM en tiempo de compilación.

[8] `https://www.doctrine-project.org/projects/doctrine-annotations/en/latest/index.html`

```php
/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 * @ORM\InheritanceType("SINGLE_TABLE")
 * @ORM\DiscriminatorColumn(name="entity_type")
 * @ORM\DiscriminatorMap({
 *     "thing": "App\RosettaBundle\Entity\Thing",
 *     "work":  "App\RosettaBundle\Entity\Work\AbstractWork",
 *     "book":  "App\RosettaBundle\Entity\Work\Book",
 *     "org":   "App\RosettaBundle\Entity\Organization",
 *     "pers":  "App\RosettaBundle\Entity\Person"
 * })
 */
abstract class AbstractEntity {
    /**
     * @ORM\Id
     * @ORM\Column(type="integer", options={"unsigned":true})
     * @ORM\GeneratedValue
     */
    protected $id;

    /** @ORM\Column(type="datetime") */
    protected $creationDate = null;

    /** @ORM\Column(length=300) */
    protected $slug = null;

    // [...]
}
```

Código 5: Anotaciones ORM de `AbstractEntity`

El comentario asociado a la clase del código 5 define a `AbstractEntity` como una entidad que se guardará en la tabla "entity" junto a todas las propiedades de sus clases hijas, utilizando la columna "entity_type" como un discriminador que indica el tipo de entidad. Los comentarios que se encuentra encima de cada atributo define cómo se guardará el valor de dicho atributo en la base de datos.

Una clase de Doctrine ORM también puede tener relaciones con otras clases, que igualmente se configuran utilizando anotaciones:

```php
abstract class AbstractEntity {
    // [...]

    /**
     * @ORM\OneToMany(
     *     targetEntity="Other\Identifier",
     *     indexBy="id",
     *     mappedBy="entity"
     * )
     */
    protected $identifiers;

    public function __construct() {
        $this->identifiers = new ArrayCollection();
        // [...]
    }

    // [...]
}
```

Código 6: Ejemplo de relación ORM en `AbstractEntity`

Para el ejemplo anterior, se define una relación uno a muchos (*One to many*) entre `AbstractEntity` y la clase `Identifier`, donde el atributo "id" de `AbstractEntity` es el *inverse side* y el atributo "entity" de `Identifier` se corresponde con el *owning side*. Del mismo modo, habrá que definir una relación muchos a uno en la clase `Identifier` para completar la unión.

Lamentablemente, los tipos de relaciones permitidos en Doctrine ORM son limitados. Es por ello que la relación entre `AbstractEntity` y `Relation`, que es un tanto peculiar (véase apartado 3.1.3), **son en realidad dos relaciones** pues no se puede distinguir un solo par *owning* e *inverse* como nos exige Doctrine ORM.

```php
abstract class AbstractEntity {
  // [...]

  /**
   * @ORM\OneToMany(targetEntity="Relation",
   *                mappedBy="to")
   */
  protected $relationsTo;

  /**
   * @ORM\OneToMany(targetEntity="Relation",
   *                mappedBy="from")
   */
  protected $relationsFrom;

  // [...]

  public function getRelations() {
      return new ArrayCollection(array_merge(
          $this->relationsTo->toArray(),
          $this->relationsFrom->toArray()
      ));
  }

  public function addRelation(Relation $relation): self {
      if ($relation->getFrom() === $this) {
          $this->relationsFrom->add($relation);
      } else {
          $this->relationsTo->add($relation);
      }
      return $this;
  }

  // [...]
}
```

Código 7: Implementación de relaciones en `AbstractEntity`

Por último en relación con el *mapping* de entidades, una clase puede tener escuchadores de eventos, que provee Doctrine ORM. Un ejemplo de uso de esta funcionalidad está en la clase `AbstractEntity`, que guarda una referencia a sí misma en todos sus identificadores asociados para definir el *owning side* de esa relación:

```php
abstract class AbstractEntity {
    // [...]

    /**
     * @ORM\PrePersist
     * @ORM\PreUpdate
     */
    public function linkIdentifiers(): self {
        foreach ($this->identifiers as $identifier) {
            $identifier->setEntity($this);
        }
        return $this;
    }

    // [...]
}
```

Código 8: Ejemplo de escuchador ORM en `AbstractEntity`

El servicio `CacheService`

Después de realizar una búsqueda, el servicio `SearchEngine` contacta con `CacheService` para que este último se encargue de actualizar la base de datos local, ya sea creando nuevas entidades o modificando los datos de las ya existentes.

Cuando se invoca al método `CacheService::persistEntities($entities)`, el servicio recibe del motor de búsqueda solo las entidades que se mostrarán al usuario en los resultados, pero también se deben cachear las que aparezcan en las relaciones (como los autores de una obra). Por ese motivo, `CacheService` crea una cola con todas las entidades a persistir, incluyendo las de las relaciones, que a efectos prácticos se almacena en forma de tabla *hash* para obtener un mayor rendimiento al trabajar con muchos elementos.

Una vez inicializada la cola, se procesa cada entidad una a una para decidir qué hacer sobre el caché, y por último se persisten los cambios. El diagrama de flujo correspondiente al procesamiento de cada elemento de la cola es el siguiente:

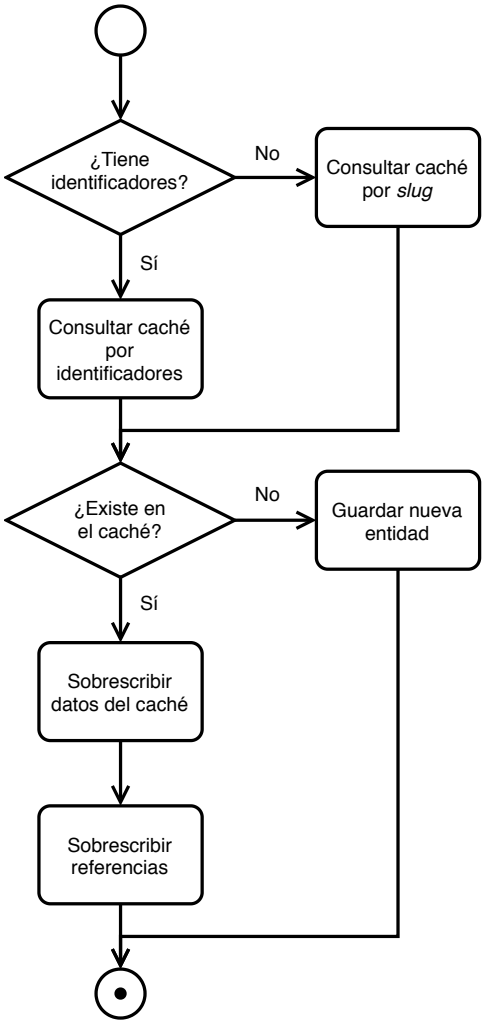


Figura 3.8: Proceso de *caching* de entidades

En primer lugar se comprueba si la entidad que se quiere cachear dispone de identificadores con los que podamos buscarla en la base de datos.

De ser así, se recuperan del caché todas las entidades que tengan cualquiera de los identificadores que hemos obtenido. En caso contrario, se buscan entidades con el mismo *slug* en el caché.

Si no existen resultados para la consulta ejecutada, se hace persistir la entidad. Si sí que existe, se convierte a la nueva entidad en la sustituta de su correspondiente entrada en el caché copiando el identificador y la fecha de creación de la entidad antigua a la nueva.

Adicionalmente, en este último caso se deben reemplazar las referencias hacia esta entidad en el resto de entidades de la cola de cacheables.

3.2. Motor de búsqueda

La tarea principal de la que se encarga el núcleo de Rosetta es la búsqueda de resultados en las distintas fuentes de datos conectadas a la plataforma. El servicio que coordina todas estas búsquedas es `SearchEngine` a través del método público `SearchEngine::search(SearchQuery $query, [$databases])`.

Para realizar una búsqueda se necesitan dos argumentos: la consulta como instancia de `SearchQuery` (véase apartado 3.2.1) y las bases de datos donde buscar, que se proveen en forma de *array* de los identificadores de las fuentes internas (véase apartado 5.3).

Rosetta realiza las búsquedas siguiendo este esquema:

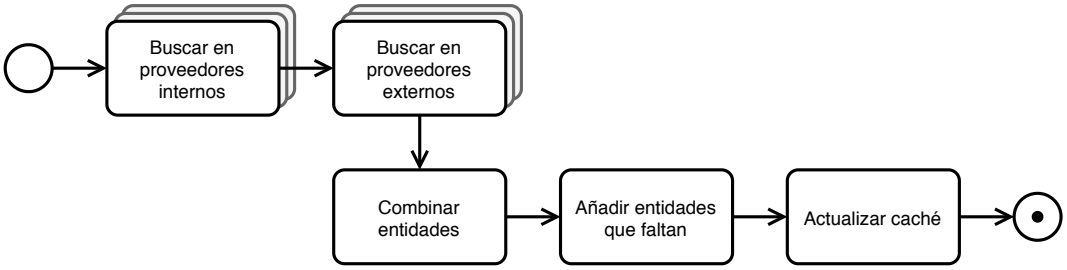


Figura 3.9: Proceso de búsqueda simplificado

En primer lugar, se instancian y configuran los proveedores de las bases de datos a buscar (fuentes internas), que se ejecutan de forma paralela. Una vez todos los proveedores han devuelto una lista de entidades como resultado de la búsqueda, se realiza una segunda ronda del mismo proceso, esta vez con los proveedores externos.

> Los proveedores externos sirven para corregir datos de los resultados devueltos por las fuentes internas y añadir nueva información, como por ejemplo el número de páginas o el idioma en el que está escrito un libro, que estos primeros no hayan sido capaces de proporcionar.

Después, se combinan las entidades de las fuentes internas y de las externas en una misma lista, que se pasa al método `SearchEngine::groupResults($results)` para agrupar sus elementos y evitar los duplicados.

Una vez agrupadas las entidades, se combinan las de cada grupo en una sola entidad a través del método `AbstractEntity::merge($entity)`, que copia los valores de los atributos de la instancia proporcionada en el argumento hacia la instancia actual (`$this`).

```
function combineGroupedResults($groupedEntities) {
    $res = [];
    foreach ($groupedEntities as $group) {
        $entity = $group[0];
        for ($i=1; $i<count($group); $i++) {
            $entity->merge($group[$i]);
        }
        $res[] = $entity;
    }
    return $res;
}
```

Código 9: Combinación de resultados de una búsqueda

Antes de terminar el proceso, se añaden nuevas entidades relacionadas con las de los resultados mediante el servicio de Wikidata (`WikidataService`). Por último se actualiza el caché local de la aplicación y se devuelven los resultados de la búsqueda en forma de listado de entidades.

3.2.1. Consultas de búsqueda

Rosetta es una aplicación que trabaja con múltiples fuentes de datos donde cada una de ellas habla un lenguaje distinto, lo que **requiere de un idioma común** para que los proveedores de dichas fuentes puedan comunicarse entre ellos y con el núcleo de Rosetta. La solución que se plantea para normalizar las consultas de búsqueda es el paquete `App\RosettaBundle\Query` y su clase principal `SearchQuery`.

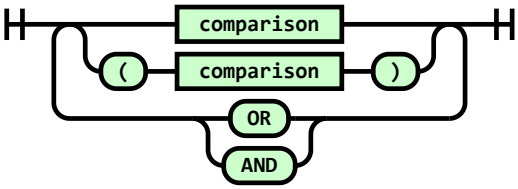
La clase `SearchQuery` representa una consulta de búsqueda y es utilizada fundamentalmente en el servicio `SearchEngine` para indicar los términos a buscar. Soporta comparaciones de campos (operadores «igual» y «contiene») y lógica sencilla de proposiciones (operadores `AND` y `OR`) para construir y representar sentencias de búsqueda avanzada (véase apartado 5.1.1).

La forma recomendada de crear una instancia de `SearchQuery` es a través del método estático `SearchQuery::of($query)`, que puede recibir como parámetro un *string* con la consulta de búsqueda introducida por el usuario o un *array* con los *tokens* a partir de los cuales se construirá la instancia de `SearchQuery`.

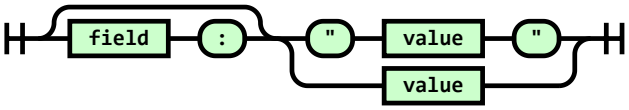
La diferencia entre el constructor de esta clase y este método es que el constructor lanzará una excepción cuando la consulta de búsqueda no sea válida (esté mal formateada), mientras que el método estático generará un `SearchQuery` que buscará la consulta errónea como un literal.

Para poder generar estas instancias, `SearchQuery` incluye un mini-compilador formado por un **analizador léxico** (junto a un pequeño analizador semántico) y un **sintetizador**. El analizador léxico toma el *string* introducido por el usuario y lo convierte a una serie de *tokens* que luego son pasados por el sintetizador para generar la instancia.

La sintaxis del lenguaje de búsqueda representada en forma de diagrama de ferrocarril (*railroad diagram*) es la mostrada en la figura 3.10, que aparece a continuación.



(a) Diagrama principal



(b) Elemento *comparison*

Figura 3.10: Sintaxis del lenguaje de búsqueda

Como se puede ver en la figura 3.10a, las sentencias de búsqueda pueden estar formadas por uno o más elementos de tipo *comparison* que se combinan utilizando los operadores `AND` y `OR`.

Para establecer la prioridad de operación se permite el uso de paréntesis que, de omitirse, se da más peso a los elementos que aparezcan antes (de izquierda a derecha). Es decir, el operador `AND` no tiene preferencia sobre `OR` ni viceversa. Dicho de otra forma, la consulta "a OR b AND c" es interpretada por la aplicación como "(a OR b) AND c".

Un *comparison* está formado por un valor que se buscará en todos los campos indexables de una entidad que opcionalmente puede ir precedido por un filtro de campo (por ejemplo, el *comparison* "quijote" se buscará en cualquier campo mientras que "title:quijote" solo es aplicable al título de una entidad).

La representación interna que utiliza Rosetta para almacenar estas consultas se basa en la recursividad de las clases `SearchQuery` y `Comparison`.

Una instancia de `SearchQuery` tiene dos atributos que la caracterizan: `operand` y `items`. El primero (`operand`) es el operador con el que se unen todos los elementos que contiene, donde un `item` puede ser una instancia de `Comparison` u otro `SearchQuery` (así se implementa el anidamiento de consultas).

La clase `Comparison` se define por el triple `field` (el campo de una entidad donde realizar la comparación), `operand` (el tipo de operador) y `value` (el valor por el que filtrar). Una instancia de `Comparison` puede tener como operador `Operand::EQUALS` (por defecto) u `Operand::CONTAINS`.

Si el elemento *value* de la figura 3.10b se encuentra englobado entre los caracteres porcentaje («%»), Rosetta entiende que el operador de la comparación es `Operand::CONTAINS` en vez de `Operand::EQUALS`. Adicionalmente, si un elemento *value* contiene espacios se deberá englobar entre comillas dobles o simples para escaparlos.

Ejemplos de consultas

Tomemos como ejemplo la consulta de búsqueda `author:%cervantes% AND title:'la galatea'`. El primer paso que realiza `SearchQuery` es pasar la cadena de texto a un *array* de *tokens*:

```
[
    'author:"%cervantes%"',
    'AND',
    'title:"la galatea"'
]
```

Código 10: Listado de *tokens* para consulta de ejemplo

Después, analiza cada *token* y lo convierte a su correspondiente representación interna basada en clases, como se muestra en la siguiente figura:

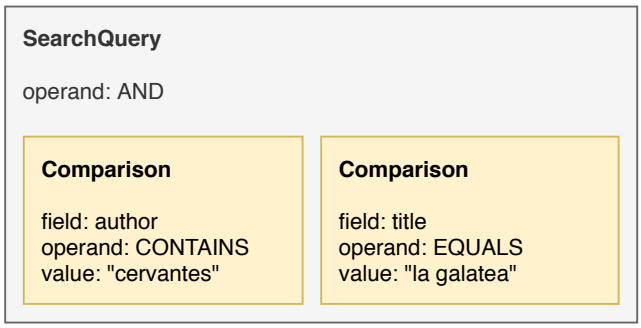


Figura 3.11: Representación de clases para consulta de ejemplo

En el caso de que la consulta tenga sentencias anidadas o mezcle tipos de operadores, existe un paso adicional dentro de la tokenización. Supongamos que partimos de la consulta `author:%cervantes% AND ('la galatea' OR title:%quijote%)`.

La primera ronda de tokenización devuelve el siguiente *array*:

```
[
    'author:"%cervantes%"',
    'AND',
    "'la galatea' OR title:%quijote%"
]
```

Código 11: Listado de *tokens* para consulta avanzada de ejemplo

En la segunda ronda, Rosetta procesará los *tokens* que están semi-extendidos, como el tercer elemento de la lista anterior:

```
[
    'author:"%cervantes%"',
    'AND',
    ['"la galatea"', 'OR', 'title:"%quijote%"']
]
```

Código 12: Listado de *tokens* extendidos para consulta avanzada de ejemplo

Por último, y al igual que el ejemplo anterior, se convierten los *tokens* a una representación de clases:

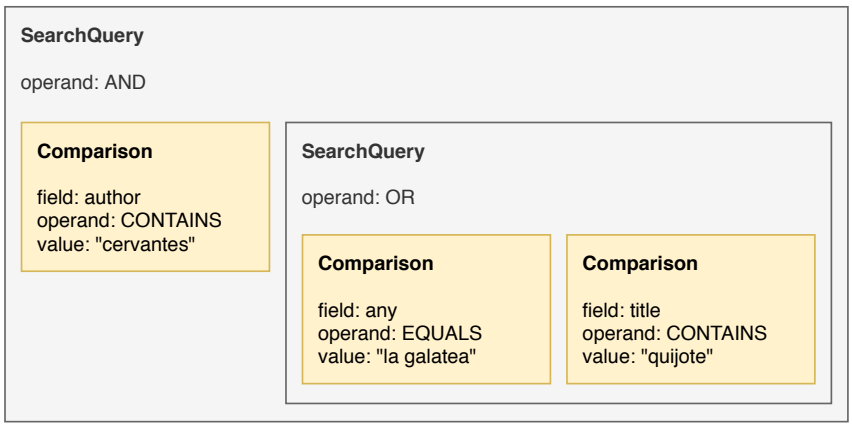


Figura 3.12: Representación de clases para consulta avanzada de ejemplo

3.2.2. Proveedores

Los proveedores son la pieza fundamental del motor de búsqueda de Rosetta, pues juegan el papel de "intérpretes" o mediadores entre las fuentes de datos y la aplicación. Estos son clases que implementan `App\RosettaBundle\Provider\AbstractProvider` y obtienen las entidades como resultado de una búsqueda en cinco pasos:

1. Instanciación: se crea un nuevo objeto de un proveedor de un tipo determinado que se utilizará para realizar la búsqueda.

2. `AbstractProvider::configure($config, $query)`: el proveedor recibe la configuración de la base de datos con la que conectar y la consulta de búsqueda (`SearchQuery`) a ejecutar.

3. `AbstractProvider::prepare()`: prepara la instancia para ser ejecutada, generalmente utilizado para sincronizar múltiples objetos de un mismo tipo de proveedor a la hora de realizar una búsqueda paralelizada.

4. `AbstractProvider::execute()`: ejecuta la búsqueda, siendo lo habitual que este método llame a otro método estático del proveedor para ejecutar varias búsquedas a la vez e ignorando las llamadas del resto de instancias de dicho proveedor.

5. `AbstractProvider::getResults()`: devuelve un *array* de `AbstractEntity` con los resultados obtenidos y limpia la memoria o estado interno del proveedor, por lo que una vez llamado a este método el servicio de búsqueda (`SearchEngine`) guarda una referencia a los resultados para no perderlos.

De los métodos mencionados en los pasos anteriores, todos son abstractos (deben ser implementados por los distintos tipos de proveedores) salvo el primero, pues la configuración de `AbstractProvider` se realiza de forma automática y soporta la detección de *presets* o configuraciones predeterminadas a través del método `AbstractProvider::getPresets()`, que puede ser implementado por cada tipo de proveedor.

Es importante entender que todos estas etapas o pasos son realizados por el motor de búsqueda **en tandas** a excepción de `AbstractProvider::prepare()` que se ejecuta inmediatamente después de configurar la instancia del proveedor.

Esto puede apreciarse en el método `getResultsFromProviders($query, $providersConfig)` de `SearchEngine`, donde se ve claramente el ciclo de vida de un proveedor (código 13 que aparece a continuación).

```php
private function getResultsFromProviders($query, $configs) {
    $providers = [];
    foreach ($configs as $config) {
        $provider = new $config['type']($this->logger);
        $provider->configure($config, $query);
        $provider->prepare();
        $providers[] = $provider;
    }

    foreach ($providers as $provider) $provider->execute();

    $results = [];
    foreach ($providers as $provider) {
        $providerResults = $provider->getResults();
        $results = array_merge($results, $providerResults);
    }

    foreach ($providers as $provider) unset($provider);
    return $results;
}
```

Código 13: Método `SearchEngine::getResultsFromProviders()`

Este proceso es el mismo a seguir tanto para los proveedores internos como para los externos pues, aunque la consulta de búsqueda no es la misma en ambos casos, las entidades finales se tratan igual independientemente de su origen.

Por último, mencionar que la configuración de la instancia del proveedor y la consulta de búsquedas pueden ser accedidas en tiempo de ejecución desde los atributos protegidos `AbstractProvider::config` y `AbstractProvider::query`, respectivamente.

```php
public function configure(array $config, SearchQuery $query) {
    $presets = $this->getPresets();
    foreach ($presets[$config['preset']] as $prop=>$val) {
        if (empty($config[$prop])) $config[$prop] = $val;
    }
    $this->config = $config;
    $this->query = $query;
}
```

Código 14: Método `AbstractProvider::configure()` simplificado

Diferencias entre proveedores internos y externos

A nivel de instancias de clases que heredan de `AbstractProvider` no existe ninguna diferencia que modifique su comportamiento en función de si ejerce como proveedor interno o externo (**un proveedor se limita a recibir una consulta y a devolver resultados**).

La discordancia entre estas dos formas de actuar de un proveedor reside en la clase `SearchEngine` y en sus métodos `getLocalResults($query, $databases)` y `getExternalResults($entities)`. Ambos métodos acaban llamando a `SearchEngine::getResultsFromProviders($query, $configs)` aunque **con consultas de búsqueda diferentes**.

Para el caso de los proveedores internos (locales), se toma la consulta proporcionada por el usuario y se pasa tal cual por referencia a los proveedores que se instancian en ese el mismo método.

Para los proveedores externos, la `SearchQuery` a utilizar se genera a partir de los resultados obtenidos de los proveedores internos, extrayendo los identificadores de dichas entidades y creando una expresión fruto de combinar múltiples comparaciones con operadores `Operand::OR`.

```php
$identifiers = [];
foreach ($entities as $entity) {
    // [...]
}

$query = [];
foreach ($identifiers as $expression) {
    $query[] = $expression;
    $query[] = Operand::OR;
}
array_pop($query);
$query = SearchQuery::of($query);
```

Código 15: Generación de consulta de búsqueda para proveedores externos

3.2.3. Agrupación de resultados

Como ya se ha mencionado al inicio de este apartado en la página 51, después de obtener los resultados de los proveedores internos y externos **las entidades** resultantes que sean **idénticas se agrupan** para después combinar esos grupos en una sola entidad siguiendo el algoritmo definido en el bloque de código 9.

Esta agrupación de elementos es realizada por el servicio de búsqueda en el método `SearchEngine::groupResults($entities)` y consta de los siguientes pasos:

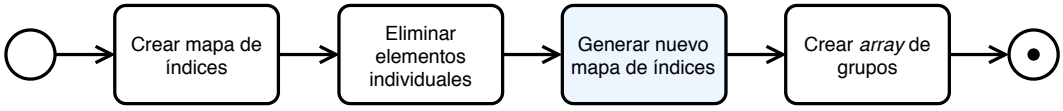


Figura 3.13: Proceso simplificado de agrupación de resultados

En primer lugar, se toma un *array* de resultados (entidades) de las que se extraen sus respectivos identificadores para acabar creando un mapa que relacione a ambos:

```
$ids = [];
foreach ($entities as $i=>$entity) {
    foreach ($entity->getIdentifiers() as $identifier) {
        if ($identifier->getType() !== Identifier::ISBN_13) {
            $id = (string) $identifier;
            if (!isset($ids[$id])) $ids[$id] = [];
            $ids[$id][] = $i;
        }
    }
}
```

Código 16: Generación de mapa de índices para agrupación

Como se puede ver en el código anterior, guardamos grupos de números enteros, correspondientes a los índices de las entidades en el *array* `$entities`, en vez de las referencias a las entidades. Una vez generado este *array* bidimensional, se eliminan (filtran) los elementos que solo tengan un identificador, pues esos no hace falta agruparlos:

```
$ids = array_filter($ids, function($elem) {
    return count($elem) > 1;
});
$ids = array_values($ids);
```

Código 17: Eliminación de índices duplicados para agrupación

Para generar el nuevo mapa (al que llamaremos `$parsedIds`) se recorren todos los grupos de índices sacando la cabeza de la lista en cada iteración para comparar sus elementos con el segundo grupo del *array*, de acuerdo al siguiente proceso:

1. Sea `$n` igual a 0
2. Se extrae la cabeza del *array* `$ids`, que se almacena en `$indexes`
3. Se buscan elementos comunes entre `$indexes` e `$ids[$n]`
   a) Si no hay elementos comunes, se marca un *flag* para inserta el grupo actual (`$indexes`) al final de la iteración en `$parsedIds`
   b) Si hay elementos comunes significa que son la misma entidad, por lo que se añaden los índices de `$indexes` a `$ids[$n]` eliminando duplicados y ordenándolos de forma ascendente
4. Se suma una unidad a `$n` y se vuelve al paso 3 hasta recorrer por completo el *array* `$ids`
5. Se vuelve al paso 2 mientras queden elementos en `$ids`

Por último, con el *array* `$parsedIds` ya generado que contiene los índices de las entidades que son idénticas agrupados, se reemplazan estos números enteros por la instancia de su correspondiente entidad extraída de la variable `$entities`.

Ejemplo de agrupación

Supongamos que partimos de las siguientes entidades numeradas del 1 al 5 que tienen asociadas unos identificadores `(A, B, C, D, E, F)`:

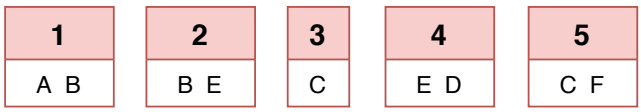


Figura 3.14: Entidades de partida para ejemplo de agrupación

Para realizar su agrupación en entidades idénticas, en primer lugar tendríamos que crear un mapa de índices (invertir la relación entidades-identificadores):

```
[
  "A" => [1],
  "B" => [1, 2],
  "C" => [3, 5],
  "D" => [4],
  "E" => [2, 4],
  "F" => [5]
]
```

Código 18: Ejemplo de mapa de índices para agrupación

De este mapa nos quedaríamos solo con `[(1,2) (3,5) (2,4)]`.

Para generar el mapa de identificadores procesado (`$parsedIds`), recorremos los tres elementos que tenemos de izquierda a derecha, lo que implica las siguientes transformaciones (a la izquierda del separador vertical `$parsedIds`, a la derecha `$ids`):

```
[] | [(1,2) (3,5) (2,4)]
  [] | [(3,5) (1,2,4)]
  [(3,5)] | [(1,2,4)]
 [(3,5) (1,2,4)] | []
```

Código 19: Ejemplo de construcción de `$parsedIds`

Por último, sustituimos los números del 1 al 5 por su correspondiente entidad y habremos terminado de agrupar los resultados. Si faltasen entidades por nombrar en el *array* `$parsedIds` (por ejemplo, por ser ya únicas de por sí) se añadirían en este momento en grupos independientes.

3.2.4. Datos estructurados con Wikidata

Tal y como se ha mencionado al comienzo de este apartado en la página 53, antes de cachear y devolver los resultados de una búsqueda se llama al servicio de Wikidata para **intentar añadir entidades relacionadas** que no ha sido posible obtener de los proveedores internos o externos.

Este servicio, localizado en la clase `WikidataService`, recibe un *array* de entidades a través del método `WikidataService::fillEntities($entities)` y modifica esas mismas entidades añadiendo las nuevas que haya sido capaz de localizar.

Básicamente, el flujo de trabajo de este servicio se reduce a tres pasos:

Obtener identificadores de Wikidata

Lo primero que se debe hacer para extraer información utilizando la API de Wikidata es obtener los identificadores externos del tipo `Identifier::WIKIDATA` con los que se corresponden nuestras entidades.

Para ello se preparan una serie de llamadas hacia el *endpoint* `https://www.wikidata.org/w/api.php?action=wbsearchentities` con el nombre de la entidad como término de búsqueda. Estas llamadas HTTP se ejecutan en paralelo gracias a la clase `HttpClient` que provee el núcleo de Rosetta.

Obtener datos de entidades de Wikidata

Una vez se tienen los identificadores correspondientes a las entidades encontradas en Wikidata, se realiza una sola petición `GET` hacia `https://www.wikidata.org/w/api.php?action=wbgetentities` para obtener los datos de todas las entidades de Wikidata.

Rellenar entidades de Rosetta

Por último, se recorre cada una de las entidades obtenidas en el paso anterior y se rellenan los atributos existentes comunes entre la entidad de Wikidata y la de Rosetta, añadiendo nuevas relaciones e información a los resultados de la búsqueda.

Para este último paso se llaman a varios métodos privados del estilo `WikidataService::fill<tipo>($entity, $data)` en función del tipo de entidad. Por ejemplo, para una instancia que represente a una persona se llamaría primero a `WikidataService::fillEntity($entity, $data)` y después a `WikidataService::fillPerson($entity, $data)`.

3.3. Interfaz gráfica

La interfaz gráfica de Rosetta se encuentra estructurada en controladores (que contienen la lógica a ejecutar cuando se visitan determinadas páginas) y plantillas (que definen la respuesta que devuelve el servidor al cliente).

Los controlares se ubican en el directorio `src/Controller` y son clases que extienden de `AbstractController`, esta última proporcionada por Symfony.

La tarea principal de un controlador es **definir rutas** y devolver las respuestas a las peticiones de un cliente. Estas rutas o direcciones URL visitables por un usuario se configuran utilizando anotaciones escritas antes de la cabecera de un método:

```
class MainController extends AbstractController {
    /** @Route("/", name="homepage") */
    public function homepage() {
        return $this->render("pages/homepage.html.twig");
    }
}
```

Código 20: Controlador de la página principal

Para el caso de páginas estáticas como la de inicio, el controlador se define en apenas unas líneas como se ve en el bloque de código 20, aunque lo habitual es que los métodos de estos controladores tenga más lógica dedicada a interactuar con otros

servicios para recuperar los resultados de una búsqueda, obtener recursos de otras máquinas, etc.

Las plantillas se localizan en los directorios `templates/pages` para las páginas usadas en los controladores y `templates/components` para los componentes reutilizables entre plantillas, y funcionan con el motor de renderizado «Twig».

Todas las páginas heredan del fichero `templates/base.html.twig` que provee de una definición de documento HTML común para toda la aplicación:

```
<!doctype html>
<html lang="{{ app.request.getLocale() }}">
  <head>
    <meta charset="utf-8">
    <title>{{ rosetta.opac.app_name }}</title>
    {% block stylesheets %}
        {{ encore_entry_link_tags('app') }}
    {% endblock %}
  </head>
  <body>
    {% block content %}{% endblock %}
    {% block javascripts %}
        {{ encore_entry_script_tags('app') }}
    {% endblock %}
  </body>
</html>
```

Código 21: Extracto de `base.html.twig`

Esta plantilla base contiene unos bloques (entre `block` y `endblock`) que pueden ser sustituidos o extendidos por las plantillas que heredan de esta:

```
{% extends 'base.html.twig' %}
{% block content %}
  <div class="search-container">
    <div class="search-wrapper">
      {% include 'components/logo.html.twig' %}
      {% include 'components/search-bar.html.twig' %}
    </div>
  </div>
  <div class="version-watermark">v{{ rosetta.version }}</div>
{% endblock %}
```

Código 22: Extracto de `homepage.html.twig`

Esta misma arquitectura jerárquica se utiliza para renderizar los atributos y componentes propios de cada una de las entidades de Rosetta a partir de las plantillas del directorio `templates/pages/details`.

3.3.1. Extensión de Twig

Las plantillas de Twig están aisladas del resto del entorno de la aplicación y disponen de un set de funciones muy limitado. Aunque esto está hecho adrede para aislar la lógica de la aplicación de la interfaz gráfica, hay momentos en los que es necesario establecer puentes entre ambas piezas de un software para que puedan interactuar entre sí determinados componentes. La forma en la que Twig permite extender las funciones que pueden ser llamadas desde dentro de una plantilla es a través de las extensiones.

Rosetta incluye una extensión llamada `AppExtension` que añade constantes en forma de variables globales y funciones al motor de renderizado para que las interfaces gráficas puedan obtener información de determinados servicios.

> Nótese que esta clase está fuera del núcleo de la aplicación (`RosettaBundle`) al añadir funcionalidad a la interfaz gráfica y no al núcleo en sí.

Las constantes proporcionadas por `AppExtension` se engloban dentro del objeto `rosetta` y contiene las siguientes propiedades:

```
"rosetta" => [
    "version"   => ConfigService::getVersion(),
    "opac"      => ConfigService::getOpacSettings(),
    "databases" => [
        {identificador} => {propiedades},
        // [...]
    ],
    "context" => [
        "db"      => {identificador},
        "logo"    => {ruta_hacia_logo},
        "leading" => {ruta_hacia_leading}
    ]
]
```

Código 23: Propiedades de la variable global `rosetta`

La constante `rosetta.database` es un *array* asociativo o mapa con todas las bases de datos de la aplicación. El contexto (`rosetta.context`) está formado por las constantes asociadas a la base de datos actualmente seleccionada, por ejemplo, en una búsqueda.

Las funciones que añade `AppExtension` son:

- `rosetta_asset(tag)`: devuelve la ruta de un recurso del directorio de *assets* para un nombre de fichero o "tag" dado
- `rosetta_date(date)`: devuelve la representación en texto de una fecha localizada al idioma del usuario
- `rosetta_language(code)`: a partir del código ISO 639-1 de un idioma, devuelve su nombre traducido al idioma del usuario
- `rosetta_source_name(sourceId, getShortName=false)`: obtiene el nombre de una fuente de datos a partir de su identificador
- `rosetta_entity_path(entity)`: devuelve la dirección URL de una entidad dentro de la aplicación
- `rosetta_external_links(entity)`: genera la lista de enlaces externos de una entidad
- `rosetta_get_map(holding)`: llama a `MapsService` para obtener el mapa correspondiente a un ejemplar

3.3.2. Mapas de ubicación

Rosetta es capaz de gestionar y renderizar mapas que indiquen la ubicación física de los distintos recursos de una institución o biblioteca. Estos mapas son ficheros de imagen SVG modificados, almacenados en el directorio `assets/custom/maps` y gestionados por el servicio `MapsService`, que incluyen atributos adicionales para definir las categorías que contienen las estanterías de una sala o el nombre de la misma.

> Para más información sobre el formato de los mapas de Rosetta y cómo se configuran, consulte el apartado 5.3.3 de la documentación.

Dado que una plataforma que ejecute Rosetta puede contener una gran cantidad de mapas, es necesario una forma efectiva de localizar el fichero correspondiente para la ubicación de cada recurso. La forma en la que `MapsService` consigue un alto rendimiento y tiempo bajos de ejecución es a través de **un índice que se almacena en caché** y se regenera solo cuando se detectan cambios en el directorio donde están alojados los mapas.

Este índice es una especie de "listín telefónico" que se encuentra indexado por base de datos, patrón de ubicación y categoría, esta última según su código UDC[9]. De

[9] *Universal Decimal Classification*

esta forma, cuando Rosetta solicita al servicio de mapas la renderización de una ubicación, la clase `MapsService` consulta el índice para conocer el fichero del mapa correspondiente (si existe) y una vez obtenido pinta las estanterías u objetos donde se encuentra el recurso.

```
function getMapName($dbId, $subject, $location) {
    $index = $this->getIndex();
    if (!isset($index[$dbId])) return null;

    $locationData = null;
    if (is_null($location)) {
        $locationData = $index[$dbId][''] ?? null;
    } else {
        foreach ($index[$dbId] as $locationPattern=>&$data) {
            if (preg_match("/$locationPattern/", $location)) {
                $locationData = $data;
                break;
            }
        }
    }
    if (is_null($locationData)) return null;

    $mapName = null;
    while (strlen($subject) > 0) {
        if (isset($locationData[$subject])) {
            $mapName = $locationData[$subject];
            break;
        }
        $subject = substr($subject, 0, -1);
    }
    return $mapName;
}
```

Código 24: Obtención del fichero de un mapa desde el índice

Como se puede ver en el bloque de código 24, el proceso para obtener el fichero de un mapa requiere de tres parámetros (el código de la base de datos, la signatura y el patrón RegEx de la ubicación) y consiste en básicamente tres pasos:

1. Obtener el índice para la base de datos del recurso
2. Obtener las categorías disponibles para el patrón de ubicación
3. Buscar la categoría más próxima a la de la signatura del recurso y devolver su fichero de mapa correspondiente

Como las categorías UDC son jerárquicas (es decir, «004.438» tiene como padre «004.43», que a su vez tiene como padre «004.4», que a su vez hereda de «004») se utiliza un bucle *while* hasta dar con la categoría que figura en el índice y que más se acerca a la del recurso.

En caso de no poder dar con un fichero de mapa apropiado, la función `getMapName()` devolverá `null` y Rosetta no mostrará ningún mapa al usuario en la interfaz web.

3.3.3. Internacionalización

Rosetta incluye un componente de internacionalización para adecuar los contenidos de la interfaz gráfica a la configuración solicita por el navegador del cliente. Este componente es capaz de determinar el idioma en el que se deberá mostrar la página, así como de formatear fechas de acuerdo a la región del visitante.

Para gestionar las traducciones, Rosetta utiliza el *bundle* `symfony/translation` que incluye compatibilidad con plantillas de Twig. De esta forma, las interfaces gráficas contienen las cadenas de texto localizables escritas originalmente en inglés y englobadas entre las etiquetas `{% trans %}` y `{% endtrans %}` para que después *bundle* se encarga de obtener la traducción correspondiente de los ficheros del directorio `/translations`.

```
{% extends 'bundles/TwigBundle/Exception/base.twig' %}
{% set page_title = 'Page not found'|trans %}
{% block error_content %}
  <h1 class="mt-md-5 mb-3">
    {% trans %}Page not found{% endtrans %}
  </h1>
  <p class="mb-3 mb-md-5">
    {% trans %}The resource could not be found.{% endtrans %}
  </p>
  {% include 'components/search-bar.html.twig' %}
{% endblock %}
```

Código 25: Ejemplo de plantilla Twig con textos localizables

Las traducciones del directorio `/translations` pueden guardarse en formato XLIFF, *array* de PHP, JSON o YAML, aunque por defecto Rosetta incluye con un pack de cadenas de texto en español en formato JSON.

```
{
    "All catalog": "Todo el catálogo",
    "Search": "Buscar",
    "Home": "Inicio",
    "Explore catalog": "Explorar catálogo",
    // [...]
}
```

Código 26: Extracto del fichero `translations/messages.es.json`

Para añadir traducciones a otros idiomas, basta con crear un nuevo fichero con el nombre `messages.[idioma].[formato]` en dicho directorio donde el código del idioma debe estar en formato ISO 639-1. La aplicación detectará automáticamente los nuevos *strings* sin tener que cambiar ninguna otra configuración.

A modo de ayuda y para aclarar posibles dudas, a continuación se listan algunos ejemplos de nombres de archivo:

- `messages.fr.json` es válido
- `messages.fr_BE.json` es válido (también se admite ISO 3166-1)
- `messages.fr.yaml` es válido (formato YAML)
- `messages.fr.xlf` es válido (formato XLIFF)
- `messages.fr.xliff` no es válido (para traducciones de XLIFF la extensión tiene que ser `.xlf` obligatoriamente)
- `messages.spa.json` no es válido (código de idioma incorrecto)

```
class LocaleListener {
    // [...]

    public function onKernelRequest(GetResponseEvent $event) {
        $request = $event->getRequest();
        $langs = $request->getLanguages();
        $lang = $langs[0] ?? $request->getDefaultLocale();
        $request->setLocale($lang);
        $this->translator->setLocale($lang);
    }
}
```

Código 27: Extracto de `LocaleListener`

El código de idioma del usuario se extrae de las peticiones HTTP que envía, tomando el inglés como valor de *fallback* en caso de no existir traducciones para el idioma solicitado.

Para inicializar `TranslationsBundle` con este idioma, Rosetta emplea un escuchador de eventos que es disparado cuando arranca el *kernel* de Symfony, como se muestra en el bloque de código 27. Gracias a esto no son necesarias *cookies* de sesión ni ningún otro artefacto para traducir la interfaz web.

En cuanto al resto de funciones de localización, estas dependen de la extensión de internacionalización de PHP ("intl-ext")[10] y son gestionadas por la clase `AppExtension` (véase el apartado 3.3.1).

Las dos funciones que se incluyen son:

- `getFormattedDate($date)`: devuelve la representación en texto de una fecha localizada al idioma del usuario
- `getLanguageName($code)`: a partir del código ISO 639-1 de un idioma, devuelve su nombre traducido al idioma del usuario

[10] Véase `https://www.php.net/manual/book.intl.php`

CAPÍTULO

4

PRUEBAS

En los últimos años, la industria de la ingeniería de software ha ido cambiando sus hábitos hacia un desarrollo más centrado en las pruebas y en su automatización. Con la gran complejidad que pueden alcanzar las aplicaciones actuales **no tiene sentido realizar pruebas de forma manual** pues, con el tiempo, se acaban convirtiendo en una carga y se acaban abandonando. En consecuencia, no hay diferencia entre tener pruebas manuales y no tenerlas.

Para evitar caer en esta mala práctica y al igual que con el despliegue, las pruebas de Rosetta están completamente automatizadas para garantizar la calidad del producto final **en todo momento** al ejecutarse con cada cambio enviado al repositorio central del proyecto. De esta forma se asegura que, si se realizan cambios que rompen la funcionalidad de la aplicación, se acabarán disparando alertas para avisar al desarrollador de la incidencia.

Siguiendo una división tradicional, podemos distinguir entre pruebas de caja blanca y pruebas de caja negra. Las primeras estudian el comportamiento de una entrada y sus transformaciones hasta que el sistema devuelve una salida (*structural testing*), mientras que en las segundas solo nos interesa validar la salida obtenida para una entrada determinada (*data-driven testing*).

En este proyecto se consideran pruebas de caja blanca a las realizadas sobre el *backend* (para probar determinados métodos con gran carga algorítmica) y de caja negra a las aplicadas a la interfaz gráfica (comprobar de forma muy básica que las páginas del OPAC se renderizan correctamente).

Aunque todos las pruebas se encuentran en el directorio `/tests` del proyecto en forma de tests unitarios de PHPUnit y pueden ser ejecutadas en cualquier momento usando el comando `./bin/phpunit`, a continuación se muestran unas tablas con la información de estas pruebas y su comportamiento y resultado esperados.

UT-01	**Tokenización de consultas**
Clase asociada	SearchQueryTest
Método asociado	testTokenization
Componente	SearchQuery
Versión	1.0
Autor	José Miguel Moreno López
Objetivo	Comprobar la correcta interpretación de *strings* de consultas en tokens
Precondiciones	N/A

Datos de entrada	`title:'%la galatea%' AND`
	`author:%cervantes% AND`
	`publisher:'Project Gutenberg'`

Secuencia	1. El sistema crea una instancia de SearchQuery partiendo de los datos de entrada 2. El sistema obtiene la representación de la instancia en forma de tokens

Datos de salida	`(<title CONTAINS 'la galatea'> AND`
	`<author CONTAINS 'cervantes'> AND`
	`<publisher EQUALS 'Project Gutenberg'>)`

Observaciones	N/A

Tabla 4.1: Prueba “tokenización de consultas” (UT-01)

UT-02	Conversión de consultas a RPN
Clase asociada	SearchQueryTest
Método asociado	testRpn
Componente	SearchQuery
Versión	1.0
Autor	José Miguel Moreno López
Objetivo	Comprobar la correcta traducción de SearchQuery a consultas RPN
Precondiciones	N/A
Datos de entrada	`title:'%la galatea%' AND author:%cervantes% AND publisher:'Project Gutenberg'`
Secuencia	1. El sistema crea una instancia de SearchQuery partiendo de los datos de entrada 2. El sistema obtiene la representación de la instancia en forma de consulta RPN
Datos de salida	`@and @attr 1=4 'la galatea' @and @attr 1=1003 'cervantes' @attr 1=1018 'Project Gutenberg'`
Observaciones	N/A

Tabla 4.2: Prueba “conversión de consultas a RPN” (UT-02)

UT-03	Conversión de consultas a Innopac
Clase asociada	SearchQueryTest
Método asociado	testInnopac
Componente	SearchQuery
Versión	1.0
Autor	José Miguel Moreno López
Objetivo	Comprobar la correcta traducción de SearchQuery a consultas de Innopac
Precondiciones	N/A

Datos de entrada	`title:'%la galatea%' AND`
	`author:%cervantes% AND`
	`publisher:'Project Gutenberg'`

Secuencia	1. El sistema crea una instancia de SearchQuery partiendo de los datos de entrada 2. El sistema obtiene la representación de la instancia en forma de consulta de Innopac
Datos de salida	`t:'la galatea' and a:'cervantes' and 'Project Gutenberg'`
Observaciones	N/A

Tabla 4.3: Prueba “conversión de consultas a Innopac” (UT-03)

UT-04	Consultas mal formuladas
Clase asociada	SearchQueryTest
Método asociado	testMalformedQueries
Componente	SearchQuery
Versión	1.0
Autor	José Miguel Moreno López
Objetivo	Comprobar el *fallback* hacia texto literal de consultas mal escritas
Precondiciones	N/A
Datos de entrada	`this (isn't a valid:query))`
Secuencia	1. El sistema crea una instancia de SearchQuery partiendo de los datos de entrada 2. El sistema obtiene la representación de la instancia en forma de tokens
Datos de salida	`(<any CONTAINS 'this (isn\'t a valid:query))'>`
Observaciones	N/A

Tabla 4.4: Prueba “consultas mal formuladas” (UT-04)

UT-05	Agrupación de resultados
Clase asociada	SearchEngineTest
Método asociado	testGroupResults
Componente	SearchEngine
Versión	1.0
Autor	José Miguel Moreno López
Objetivo	Validar el comportamiento del algoritmo de agrupación de resultados
Precondiciones	N/A

Datos de entrada	7 entidades con los siguientes identificadores de prueba:
	
	`['red', 'scarlet']`
	`['green']`
	`['blue']`
	`['scarlet']`
	`['cyan', 'blue']`
	`['teal', 'olive']`
	`['olive', 'green']`

Secuencia	1. El sistema obtiene una referencia al servicio de búsqueda 2. El sistema modifica los permisos para llamar al método privado `groupResults` 3. El sistema pasa los datos de entrada al método

Datos de salida	Tres grupos de entidades con los siguientes identificadores únicos por grupo:
	
	`['red', 'scarlet']`
	`['blue', 'cyan']`
	`['green', 'teal', 'olive']`

Observaciones	N/A

Tabla 4.5: Prueba “agrupación de resultados” (UT-05)

UT-06	Página de inicio
Clase asociada	HomepageTest
Método asociado	testHomepage
Componente	OPAC
Versión	1.0
Autor	José Miguel Moreno López
Objetivo	Comprobar la correcta renderización de la página de inicio
Precondiciones	N/A
Datos de entrada	N/A
Secuencia	1. El sistema carga la página de inicio de la aplicación 2. El sistema comprueba que existe el formulario de búsqueda
Datos de salida	N/A
Observaciones	N/A

Tabla 4.6: Prueba “página de inicio” (UT-06)

UT-07	Formulario de búsqueda de página de inicio
Clase asociada	HomepageTest
Método asociado	testSearchForm
Componente	OPAC
Versión	1.0
Autor	José Miguel Moreno López
Objetivo	Validar el comportamiento del formulario de búsqueda
Precondiciones	N/A
Datos de entrada	Consulta de búsqueda: "kurose"
Secuencia	1. El sistema carga la página de inicio de la aplicación 2. El sistema escribe la consulta de entrada en el formulario de búsqueda 3. El sistema envía el formulario
Datos de salida	Redirección hacia la página de resultados de búsqueda para la consulta de entrada
Observaciones	N/A

Tabla 4.7: Prueba "formulario de búsqueda de página de inicio" (UT-07)

UT-08	Página de resultados de búsqueda
Clase asociada	SearchTest
Método asociado	testSearch
Componente	OPAC
Versión	1.0
Autor	José Miguel Moreno López
Objetivo	Comprobar la correcta renderización de la página de resultados de una búsqueda
Precondiciones	N/A
Datos de entrada	Consulta de búsqueda: "kurose"
Secuencia	1. El sistema carga la página de resultados de búsqueda para la consulta de entrada 2. El sistema comprueba que existe el *leading* 3. El sistema comprueba que existe el *spinner*
Datos de salida	N/A
Observaciones	N/A

Tabla 4.8: Prueba "página de resultados de búsqueda" (UT-08)

UT-09	Obtención de resultados de búsqueda
Clase asociada	SearchTest
Método asociado	testSearchPost
Componente	OPAC
Versión	1.0
Autor	José Miguel Moreno López
Objetivo	Validar los resultados obtenidos al realizar una búsqueda en la aplicación
Precondiciones	N/A
Datos de entrada	Consulta de búsqueda: "kurose"
Secuencia	1. El sistema solicita mediante una petición POST los resultados para la consulta de entrada 2. El sistema comprueba que existe al menos un resultado que sea un libro
Datos de salida	N/A
Observaciones	N/A

Tabla 4.9: Prueba "obtención de resultados de búsqueda" (UT-09)

UT-10	Página de detalles
Clase asociada	DetailsTest
Método asociado	testDetails
Componente	OPAC
Versión	1.0
Autor	José Miguel Moreno López
Objetivo	Comprobar la correcta renderización de la página de detalles de una entidad
Precondiciones	N/A
Datos de entrada	Consulta de búsqueda: “kurose”
Secuencia	1. El sistema realiza una búsqueda para la consulta de entrada 2. El sistema hacia click sobre el primer resultado 3. El sistema comprueba que la página de detalles de la entidad tiene una columna de detalles y una principal
Datos de salida	N/A
Observaciones	N/A

Tabla 4.10: Prueba “página de detalles” (UT-10)

CAPÍTULO

5

DOCUMENTACIÓN

5.1. Manual de usuario

La interfaz gráfica de Rosetta ha sido diseñada para poder ser manejada sin necesidad de tener que consultar ningún manual ni requerir de ninguna formación previa. Aún así, en este apartado se incluyen los escasos conceptos básicos de UX necesarios para utilizar la aplicación.

5.1.1. Iniciar una búsqueda

La página de inicio de la aplicación muestra un cuadro de búsqueda donde se introducen los términos a buscar dentro de las distintas bases de datos configuradas en la plataforma.

Figura 5.1: Captura de pantalla de la página de inicio

Para realizar una búsqueda se debe hacer click sobre la barra de búsqueda (en la figura 5.1 aparece bordeada de color rojo), escribir el término a buscar y pulsar la tecla INTRO del teclado o hacer click sobre el botón "Buscar".

Por defecto, los resultados que se mostrarán serán los obtenidos de todas las bases de datos de la plataforma (todo el catálogo) salvo que se cambie el valor del desplegable de la barra de búsqueda antes de ejecutarla.

Figura 5.2: Selección de base de datos en la barra de búsqueda

Búsqueda avanzada

En el cuadro de búsqueda de Rosetta se puede introducir cualquier término (palabra o frase) que será buscado tal cual en las distintas bases de datos de la plataforma. De forma adicional, se puede acotar la búsqueda escribiendo una consulta específicamente construida como se puede ver en los siguientes ejemplos:

- `title:'%quijote%'`
  obras que contengan "quijote" en el título
- `title:'la galatea'`
  obras que tengan exactamente como título "la galatea"
- `author:'Cervantes, Miguel de'`
  obras creadas por Miguel de Cervantes
- `author:'%cervantes%' AND (title:%quijote% OR title:'la galatea')`
  obras de Cervantes que contengan "quijote" o se llamen "la galatea"

5.1.2. Resultados de búsqueda

La página de resultados de búsqueda muestra en formato de cuadrícula los elementos encontrados para la consulta introducida en el paso anterior. Esta consulta puede ser modificada a través del cuadro de búsqueda que aparece en la cabecera de la página y que se maneja igual que el mostrado en la página de inicio.

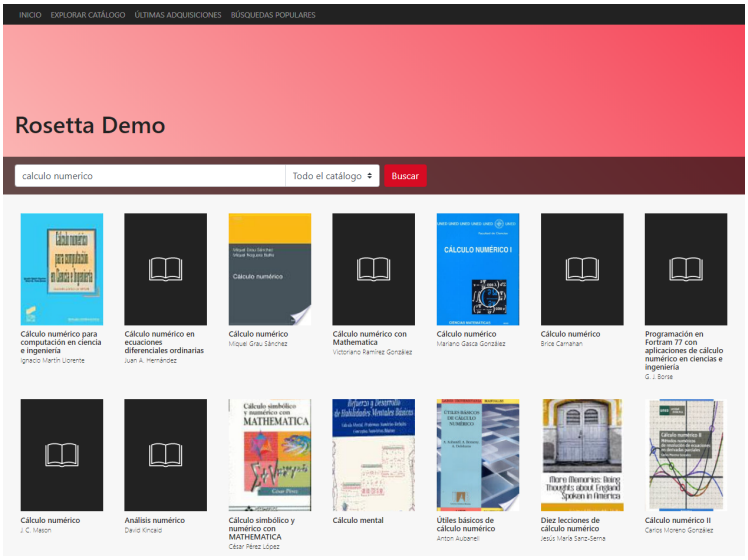


Figura 5.3: Página de resultados de la búsqueda

Para ver los detalles de un resultado, haga click sobre el elemento (ya sea la imagen de vista previa o el título) y se le llevará a la página de detalle. De igual forma, haga click sobre el subtítulo de un resultado para ir a la página de detalle de su autor principal.

5.1.3. Página de detalle

La página de detalle muestra los datos y relaciones de una entidad. Esta se estructura en tres secciones o columnas, que de izquierda a derecha son **vista previa y propiedades**, **título y datos principales** y **mapa de ubicación**.

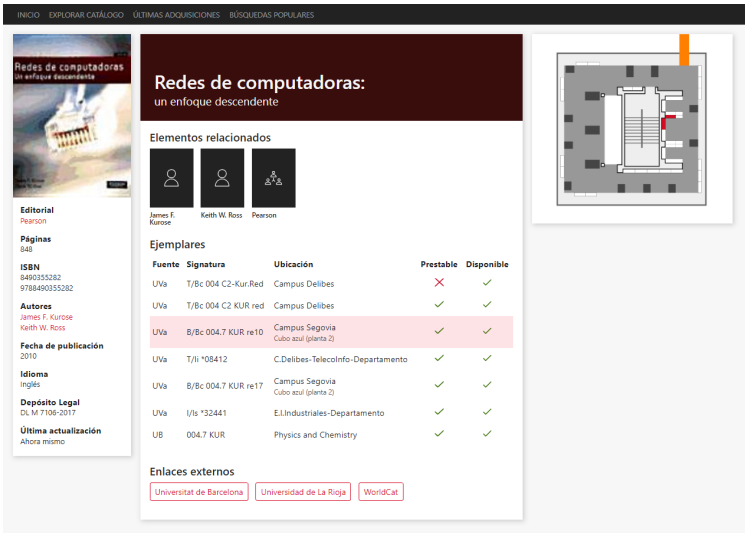


Figura 5.4: Página de detalle de una entidad

La columna más ancha (la del centro) contiene el título de la entidad, los **elementos con los que se relaciona** de forma directa, los **ejemplares** disponible y la lista de enlaces a **páginas externas** donde ver los datos de dicha entidad.

Si se hace click sobre un ejemplar que tenga en su columna de ubicación un subtítulo de color gris con el nombre de la sala en la que se encuentra, se actualizará el mapa de la derecha resaltando la estantería (o las estanterías) en las que se puede encontrar el ejemplar.

5.2. Manual de despliegue

5.2.1. Despliegue automático

El proyecto de Rosetta puede ser instalado en prácticamente cualquier máquina en apenas dos líneas de código ya que dispone de un fichero de configuración de Docker Compose[1], de forma que aquellas organizaciones que quieran montar su propia instancia de la aplicación solo deberán descargar el código fuente del repositorio y después levantar los contenedores:

```
git clone https://github.com/josemmo/rosetta.git
cd rosetta && docker-compose up -d
```

Código 28: Comandos para despliegue de Rosetta usando Docker

La configuración por defecto del proyecto incluye tres proveedores de datos — Universidad de Valladolid (requiere autenticación), Universidad de La Rioja y Universitat de Barcelona — *assets* personalizados y un mapa de ubicación para la UVa a modo de demostración.

Por supuesto, este entorno se puede modificar creando nuevos ficheros de configuración en un directorio de nuestro servidor (en este ejemplo se toma la ruta `/prod`) y mapeando sus rutas en archivo de configuración `docker-compose.yml`, como se muestra en el bloque de código 29.

Para más información sobre los ficheros de configuración, consulte el apartado de personalización en la página 92.

[1] Docker Compose es una herramienta pensada para ejecutar aplicación multi-contenedor de Docker, siendo éste último un orquestador de contenedores

```
version: '2'

services:
  # [...]
  php:
    # [...]
    volumes:
      - app-source:/rosetta
      - /prod/rosetta.yml:/rosetta/config/packages/rosetta.yml
      - /prod/theme.scss:/rosetta/assets/scss/_theme.scss
      - /prod/custom:/rosetta/assets/custom
```

Código 29: Modificación de `docker-compose.yml` para cambiar configuración

A la hora actualizar Rosetta se deben parar los contenedores activos, actualizar las imágenes y volver a levantar la aplicación:

```
git pull
docker-compose up -d --force-recreate --build
docker image prune -f # Limpieza de recursos no usados
```

Código 30: Comandos para actualización de Rosetta usando Docker

Es de especial interés mencionar que al hacer un `git pull` se reemplazará el fichero `docker-compose.yml`, por lo que deberemos tener un copia para restaurarlo al actualizar.

5.2.2. Despliegue avanzado

Para instalaciones complejas en las que se necesite mayor escalabilidad y tolerancia a fallos, es recomendable usar Docker Swarm, Kubernetes u otro orquestrador de contenedores en vez de la configuración de `docker-compose.yml` proporcionada.

Por este motivo, Rosetta incluye un `Dockerfile`[2] que define la imagen del servidor de aplicación, el cual expone un servicio público de PHP-FPM[3] en el puerto 9000 para ser conectado a un servidor web como nginx o Apache.

Aunque esta imagen puede ser integrada en producción según el desarrollador vea conveniente, la arquitectura recomendada para entornos de alto rendimiento usando Kubernetes es la siguiente:

[2] `Dockerfile` es un fichero de configuración de imagen de Docker
[3] *FastCGI Process Manager*

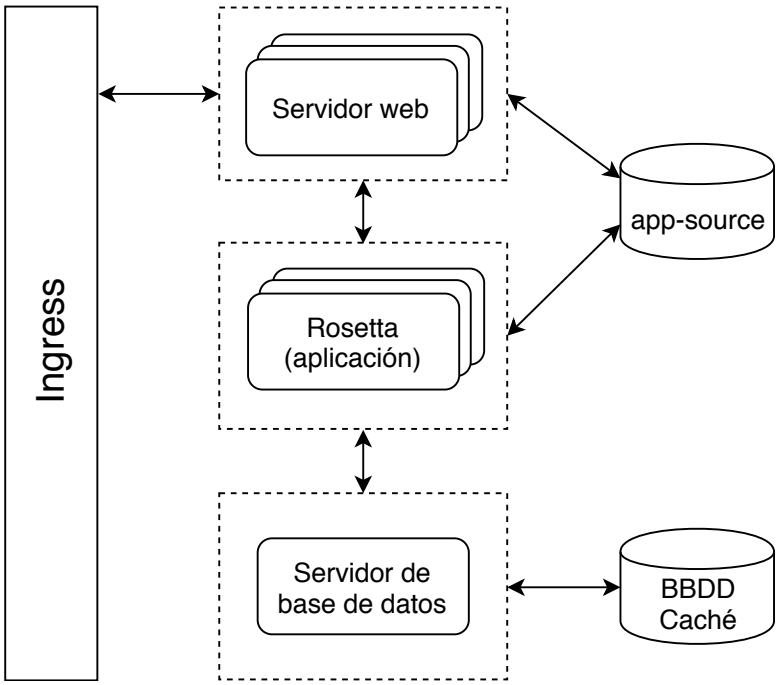


Figura 5.5: Arquitectura recomendada de despliegue con Kubernetes

La imagen de Rosetta ha sido diseñada para ser *stateless*, por lo que puede ser escalada horizontalmente creando múltiples instancias de un *pod*.

Para escalar el servidor de base de datos no podemos replicar *pods* ya que se corrompería el estado interno de los datos si se ejecutasen múltiples instancias a la vez. En su lugar, debemos crear un clúster replicado de MariaDB formado por un *pod* maestro y uno o más esclavos.

5.2.3. Despliegue manual

Tanto el fichero `docker-compose.yml` como `Dockerfile` permiten automatizar la instalación de dependencias y la preparación del entorno de ejecución necesario para ejecutar Rosetta. En caso de querer desplegar la aplicación sin utilizar Docker o Kubernetes, se deberán seguir los mismos pasos que realizan estos dos ficheros de forma manual.

Los comandos que aparecen a continuación están especialmente indicados para sistemas operativos basados en **Debian GNU/Linux con procesador x86** al ser el entorno de ejecución más popular en servidores web.

Instalación de Node.js®

Rosetta depende de Node.js® para la compilación de recursos estáticos (imágenes, hojas de estilos y scripts de JavaScript), por lo que deberá estar instalado en nuestra máquina antes de realizar la primera ejecución de la aplicación:

```
curl -sL https://deb.nodesource.com/setup_10.x | bash -
apt install nodejs
```

Código 31: Comandos para instalación de Node.js®

También necesitaremos el gestor de paquetes Yarn, que por defecto no se instala con Node.js®:

```
curl -sL https://dl.yarnpkg.com/debian/pubkey.gpg | \
  apt-key add -
echo "deb https://dl.yarnpkg.com/debian/ stable main" | \
  tee /etc/apt/sources.list.d/yarn.list
apt update && apt install yarn
```

Código 32: Comandos para instalación de Yarn

Instalación de PHP

Una vez instalado Node.js®, se deberá instalar la última versión de PHP disponible (o cualquiera igual o superior a PHP 7.1.3):

```
apt install ca-certificates apt-transport-https
wget -q https://packages.sury.org/php/apt.gpg -O- | \
  apt-key add -
echo "deb https://packages.sury.org/php/ stretch main" | \
  tee /etc/apt/sources.list.d/php.list
apt update && apt install php-cli php-common php-fpm \
  php-curl php-mbstring php-zip php-pdo-mysql php-intl \
  php-dev php-pear
```

Código 33: Comandos para instalación de PHP

Junto a PHP necesitamos instalar el gestor de dependencias Composer:

```
curl -sS https://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

Código 34: Comandos para instalación de Composer

Para los proveedores de Z39.50, es requisito indispensable la extensión YAZ, que hay que compilar desde el código fuente:

```
apt install yaz libyaz-dev
pecl install yaz
```

Código 35: Comandos para compilación de YAZ

Instalación del servidor de base de datos

Rosetta necesita un gestor de base de datos para manejar el caché local, siendo MariaDB un software más que apto para este propósito y muy fácil de configurar:

```
apt install mariadb-server
```

Código 36: Comando para instalación de MariaDB

Por defecto, MariaDB en Debian se ejecuta en el puerto 3306 con usuario `root` y sin contraseña, por lo que es recomendable cambiar estos ajustes antes de pasar a producción.

Compilación de Rosetta

Ahora que el entorno de ejecución está listo, descargamos la última versión de Rosetta desde el repositorio remoto:

```
git clone https://github.com/josemmo/rosetta.git
```

Código 37: Comando para descargar Rosetta

En este momento se deben modificar los ficheros de configuración y, una vez hecho, ya podremos compilar los recursos estáticos:

```
yarn install && yarn build
composer install --no-dev --optimize-autoloader
chmod -R 777 var
php bin/console doctrine:database:create --if-not-exists
php bin/console doctrine:schema:update --force
```

Código 38: Comandos para compilar Rosetta

Instalación del servidor web

Symfony, el *framework* en el que está basado Rosetta, es compatible con un amplio abanico de servidores web, incluyendo nginx y Apache, dos de los más utilizados.

Si optamos por usar Apache, lo más sencillo es modificar el fichero `httpd.conf` y apuntar la ruta pública hacia el directorio `public` de Rosetta e instalar el `apache-pack` de Symfony:

```
composer require symfony/apache-pack
```

Código 39: Comando para instalar `apache-pack` en Symfony

En el caso de nginx (el servidor web recomendado por tener el mejor rendimiento con Rosetta) solo deberemos cambia los ajustes del fichero `nginx.conf` por los siguientes:

```
server {
  server_name localhost;
  root /ruta/hacia/rosetta/public;

  location / {
    try_files $uri /index.php$is_args$args;
  }

  location ~ ^/index\.php(/|$) {
    fastcgi_pass unix:/var/run/php/php-fpm.sock;
    fastcgi_split_path_info ^(.+\.php)(/.*)$;
    include fastcgi_params;
    internal;
  }

  location ~ \.php$ {
    return 404;
  }
}
```

Código 40: Configuración de nginx recomendada

5.3. Manual de personalización

Rosetta ha sido diseñado para ser un software de "marca blanca" que provee de múltiples archivos de configuración y otros recursos modificables para adaptarse a prácticamente cualquier biblioteca u organización.

Los tres recursos principales a modificar son `rosetta.yml`, `_theme.scss` y el directorio `assets/custom`.

5.3.1. Configuración de la aplicación

La configuración esencial del núcleo de Rosetta y algunos ajustes de la interfaz gráfica se especifican en el fichero `rosetta.yml` del directorio `config/packages`.

Este archivo sigue el formato de serialización YAML, aunque también es posible formatearlo en XML o en un *array* de PHP según se detalla en el apartado de configuración del manual de Symfony[4].

La estructura de `rosetta.yml` se divide en los siguientes apartados:

- `opac`: configuración de la interfaz web
- `wikidata`: ajustes del servicio de Wikidata
- `databases`: configuración de los distintos proveedores de datos (fuentes internas)
- `external_providers`: configuración de los proveedores de terceros (fuentes externas)

El grupo `opac` puede tener las siguientes propiedades:

- `app_name`: nombre de la aplicación que se mostrará desde la interfaz web
- `admin_email`: dirección de correo electrónico de contacto del administrador de la plataforma
- `covers_expiration`: caducidad de las portadas y otras imágenes de entidades guardadas en caché en formato *DateTime string* de PHP (por ejemplo, "+90 days")

El grupo `wikidata` solo puede tener un atributo, `language`, que define en qué idioma se obtendrán los datos de Wikidata siguiendo el código de dos letras del país especificado en el ISO 639-1.

[4] `https://symfony.com/doc/current/configuration.html`

El grupo `databases` contiene las siguientes propiedades:

- `id`: Identificador de la base de datos a nivel interno
- `name`: Nombre de la base de datos
- `short_name`: Nombre corto de la base de datos
- `external_link` (opcional): Patrón de URL para enlaces externos
- `provider`: Configuración del proveedor

La propiedad `external_link` sirve para redirigir al usuario a la fuente original de los datos y puede contener variables que se reemplazarán en tiempo de ejecución. Por ejemplo, para la Universidad de Valladolid, un posible valor para este campo sería `https://almena.uva.es/search/i?search={{isbn13}}`, donde Rosetta sustituiría `{{isbn13}}` por el ISBN de 13 dígitos del recurso. Por último, el grupo `external_providers` contiene una lista de configuraciones de proveedores.

La estructura de una configuración de proveedor, que es la misma tanto para `databases/provider` como para los elementos de `external_providers`, es la mostrada en la tabla 5.1.

Propiedad	Por defecto	Z39.50	Google Books
`type`	-	✓	✓
`preset`	-	✓	✓
`url`	-	-	✓
`user`	-	-	✓
`password`	-	-	✓
`group`	-	-	✓
`charset`	-	-	✓
`syntax`	-	-	✓
`oclc_field`	935	-	✓
`country`	-	✓	-
`key`	-	✓	-
`get_holdings`	-	✓	✓
`covers_url`	-	✓	-
`timeout`	3	✓	✓
`max_results`	20	✓	✓

Tabla 5.1: Propiedades configurables de un `provider`

Esta tabla incluye qué propiedades se pueden usar con qué proveedor, ya que algunas son específicas de determinadas fuentes de datos.

Dentro de esta estructura, la propiedad `type` determina el proveedor a utilizar tomando como valor la ruta completa de la clase. Los dos proveedores que proporciona Rosetta en su instalación son:

- `App\RosettaBundle\Provider\Z3950`: para el protocolo Z39.50
- `App\RosettaBundle\Provider\GoogleBooks`: para Google Books

Los proveedores de las fuentes internas se utilizan para obtener los **resultados de una búsqueda y los ejemplares** disponibles dentro de una biblioteca o institución, mientras que las fuentes externas (que son opcionales) reemplazan y añaden información adicional que no se pudo obtener de las fuentes internas. El funcionamiento de los proveedores se detalla en el apartado 3.2.2.

5.3.2. Personalización de la interfaz web

Rosetta permite cambiar la paleta de colores del tema de la interfaz web sin tener que hacer grandes cambios en su código a través del fichero `assets/scss/_theme.scss`.

Esta hoja de estilos está pensada para almacenar reglas específicas de un tema y contiene las siguientes variables de SCSS cuyos valores se utilizarán a lo largo de toda la interfaz:

```
$foreground-color: #222;
$background-color: #f7f7f7;
$accent-color: #d70b23;

// Auto-generated variables
$muted-color: lighten($foreground-color, 30%);
```

Código 41: Contenido por defecto del fichero `_theme.scss`

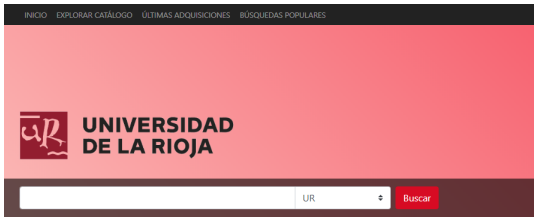
Tenga en cuenta que deberá recompilar la aplicación cada vez que haga cambios en las reglas de estilo de Rosetta usando el comando `yarn build` o, en su defecto, `npm run build`.

Si tomamos la página de inicio como ejemplo, cada variable mostrada en el *snippet* anterior se corresponde con los mostrados en la figura 5.6, que aparece inmediatamente a continuación.

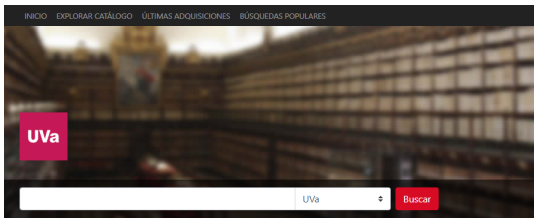
Figura 5.6: Variables del tema para la página de inicio

Además de la paleta, los logotipos y *leadings*[5] de las bases de datos también se pueden personalizar. Estos recursos de imagen deben guardarse en la raíz del directorio `assets/custom` siguiendo el esquema de nombres `[dbId]-logo.png` o `[dbId]-leading.jpg`, donde "dbId" es el identificador de la base de datos que figura en el fichero de configuración `rosetta.yml`.

(a) Sin logotipo ni *leading*



(b) Solo logotipo



(c) Con logotipo y *leading*

Figura 5.7: Interfaces de las cabeceras de búsqueda

[5]Dentro de Rosetta se conoce como *leading* a la imagen de fondo ancha y estrecha que se utiliza en la cabecera de los resultados de una búsqueda

Estas imágenes son opcionales y solo se mostrarán si el recurso existe. En caso de no existir, se mostrará en su lugar el nombre de la base de datos y el color de fondo primario (`$accent-color`) de la aplicación.

Si se quisiera aplicar cambios más específicos a los estilos, se tendría que modificar el resto de ficheros del directorio `assets/scss`. Esta carpeta contiene las reglas correspondientes a los componentes reutilizables en el subdirectorio `components` y las reglas aplicables a páginas concretas en `pages`.

Una vez más, se tendría que recompilar la aplicación después de hacer cambios sobre estos archivos.

5.3.3. Configuración de mapas

Rosetta permite mostrar al usuario a través de una serie de mapas la ubicación de un recurso dentro de las instalaciones de la biblioteca u organización. Para que la aplicación sea capaz de localizar los recursos se deben introducir, a través de ficheros basados en SVG, mapas con los códigos UDC[6] asociados a su correspondiente estantería o ubicación.

Estos mapas se guardan en el directorio `assets/custom/maps` y pueden tener el nombre que se considere conveniente, siempre y cuando tengan la extensión `.svg` y sigan el siguiente formato de archivo:

```
<?xml version="1.0" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
 xmlns:xlink="http://www.w3.org/1999/xlink"
 xmlns:rosetta="https://github.com/josemmo/rosetta"
 viewBox="[...]"
 rosetta:database="uva"
 rosetta:locationPattern="(.+)Segovia"
 rosetta:room="Cubo azul (planta 2)">
  <!-- [...] -->
  <rect rosetta:udc="001, 002"
   x="1295" y="245" width="27" height="88" />
  <rect rosetta:udc="003, 004"
   x="1295" y="333" width="27" height="88" />
  <rect rosetta:udc="004.2, 004.3"
   x="1295" y="421" width="27" height="88" />
  <!-- [...] -->
</svg>
```

Código 42: Ejemplo de mapa de Rosetta

[6] *Universal Decimal Classification*

Como se puede apreciar en el código del ejemplo anterior, los mapas de Rosetta no son más que imágenes SVG que contienen un nombre de espacios de XML adicional (en este caso, definido en `xmlns:rosetta`). Estos archivos pueden ser creados con un software de edición de imágenes vectoriales como Inkscape o Adobe Illustrator® y después modificados con un editor de texto para añadir las propiedades adicionales que convierten la imagen en un mapa de Rosetta.

El atributo `rosetta:database` contiene el identificador de la base de datos con la que se corresponde el mapa, `rosetta:locationPattern` es un patrón regex que deberá coincidir con la ubicación del recurso devuelta por el proveedor para que el mapa sea cargado; y, por último, `rosetta:room` es el nombre que recibirá la sala o ubicación que se mostrará al usuario.

Para localizar al recurso, la aplicación extrae la categoría de su signatura (correspondiente al código UDC) y la busca dentro de un mapa. Por tanto, el atributo `rosetta:udc` que contienen las estanterías es una **lista separada por comas de los códigos UDC de los recursos que alojan**.

Gracias a esta arquitectura, una estantería puede contener múltiples categorías y una categoría puede expandirse por varias estanterías. Para más información sobre cómo Rosetta es capaz de encontrar el mapa correspondiente a cada categoría de forma eficiente, consulte el apartado 3.3.2.

CAPÍTULO

6

CONCLUSIONES

Dentro de la motivación y los objetivos planteados, se ha conseguido desarrollar un nuevo tipo de software capaz de conectar con casi cualquier fuente de datos bibliográfica, independientemente de la plataforma subyacente. Para crear el producto final se ha propuesto una nueva forma de almacenar la información de la base de conocimiento de una biblioteca a partir de entidades jerárquicas que se indexan con identificadores y se relacionan entre sí.

Junto al núcleo (el *broker* de la aplicación) se ha desarrollado un OPAC fácil de usar, con una interfaz moderna y clara, que proporciona las funcionalidades básicas y necesarias para un buscador de este tipo, incluyendo la renderización de mapas de ubicación de los recursos (algo hasta la fecha impensable). Todo esto, además, con una idea de "marca blanca" siempre en mente, permitiendo cambiar la configuración de la aplicación y adaptar su interfaz gráfica a las necesidades de cada institución sin tener que modificar el código fuente.

La implementación de Rosetta hace evidente el hecho de que hay un enorme margen de mejora en el contexto del software de bibliotecas. Gracias a esta implementación es posible descentralizar la información para que cada institución o centro gestione sus datos sin tener que venderlos a multinacionales y sin sacrificar funcionalidad.

Aunque la historia reciente diga lo contrario, es posible la colaboración entre bibliotecas para que compartan sus catálogos con otras instituciones como Wikidata, enriqueciendo las bases de datos de la industria y de otros ámbitos de la cultura al crear relaciones que van más allá de lo meramente bibliotecológico.

La siguiente gran revolución en bibliotecas consiste en la catalogación no solo de libros, artículos y otros documentos, sino de todo el conocimiento humano. Sin una estructura de entidades similar a la planteada en Rosetta y sin acuerdos, esta revolución nunca tendrá lugar.

6.1. Posibles mejoras

Ante las limitaciones de tiempo y recursos, se han tenido que realizar sacrificios en funcionalidades a implementar en el producto final. Entre las más relevantes se encuentran:

- **Caché avanzado de resultados**: además de cachear las entidades, una mejora prioritaria consistiría en cachear las consultas de búsqueda y sus resultados asociados. Adicionalmente, establecer un sistema de expiración de entradas del caché para actualizar las copias locales.

- **Proveedores adicionales**: por defecto Rosetta incluye un número limitado de proveedores de datos, de entre los que se podría añadir alguno para fuentes filmográficas abiertas (como TMDb) y otros tipos de entidades.

- **Más funcionalidades en el OPAC**: tras implementar un caché avanzado, se podrían añadir nuevas páginas a la interfaz web para consultar búsquedas recientes, entidades populares, *widgets*, etc.

- ***Endpoint* de datos abiertos**: muy a largo plazo, sería buena idea permitir la interconexión de nodos de Rosetta para compartir datos entre sí, alcanzado una descentralización que permita la democratización de los datos.

ÍNDICE DE FIGURAS

1.1. Página de inicio de Bibliozambrano 2
1.2. Ventana de creación de usuarios de Millennium ILS 5
1.3. Interfaces gráficas de Sierra 6
1.4. Página de inicio de Alma 7
1.5. Edición de entradas de SirsiDynix Symphony 7
1.6. Página de inicio de administración de Koha 8
1.7. Esquema de ramas de *gitflow workflow* [5] 13
1.8. Diagrama de Gantt estimado 16
1.9. Diagrama de Gantt real 17

2.1. Árbol de características 22
2.2. Diagrama de casos de uso de usuario 24
2.3. Diagrama entidad-relación 29

3.1. *Stack* de componentes de Rosetta 36
3.2. Arquitectura simplificada de Millennium® ILS 38
3.3. Representación habitual de registros 38
3.4. Representación relacional de registros 39
3.5. Arquitectura simplificada de Rosetta 40
3.6. Diagrama de clases de entidades 40
3.7. Diagrama de clases de `AbstractEntity` 44
3.8. Proceso de *caching* de entidades 51
3.9. Proceso de búsqueda simplificado 52
3.10. Sintaxis del lenguaje de búsqueda 54
3.11. Representación de clases para consulta de ejemplo 55

3.12. Representación de clases para consulta avanzada de ejemplo 56
3.13. Proceso simplificado de agrupación de resultados 60
3.14. Entidades de partida para ejemplo de agrupación 61

5.1. Captura de pantalla de la página de inicio 83
5.2. Selección de base de datos en la barra de búsqueda 84
5.3. Página de resultados de la búsqueda 85
5.4. Página de detalle de una entidad 85
5.5. Arquitectura recomendada de despliegue con Kubernetes 88
5.6. Variables del tema para la página de inicio 95
5.7. Interfaces de las cabeceras de búsqueda 95

ÍNDICE DE TABLAS

1.1. Comparativa de software para bibliotecas 9
1.2. Costes de hardware 18
1.3. Costes de servicios 18
1.4. Costes de licencias de software 18
1.5. Costes estimados de nóminas 19
1.6. Desglose total de costes estimados 19
1.7. Costes reales de nóminas 19
1.8. Desglose total de costes reales 20

2.1. Caso de uso “realizar búsqueda” (CU-01) 25
2.2. Caso de uso “ver detalles de entidad” (CU-02) 26
2.3. Caso de uso “ver ubicación de ejemplar” (CU-03) 27
2.4. Caso de uso “ver entidad en página externa” (CU-04) 28
2.5. Diccionario de datos de “entity” 31
2.6. Diccionario de datos de “holding” 32
2.7. Diccionario de datos de “identifier” 32
2.8. Diccionario de datos de “relation” 32

4.1. Prueba “tokenización de consultas” (UT-01) 72
4.2. Prueba “conversión de consultas a RPN” (UT-02) 73
4.3. Prueba “conversión de consultas a Innopac” (UT-03) 74
4.4. Prueba “consultas mal formuladas” (UT-04) 75
4.5. Prueba “agrupación de resultados” (UT-05) 76
4.6. Prueba “página de inicio” (UT-06) 77
4.7. Prueba “formulario de búsqueda de página de inicio” (UT-07) . . . 78
4.8. Prueba “página de resultados de búsqueda” (UT-08) 79
4.9. Prueba “obtención de resultados de búsqueda” (UT-09) 80
4.10. Prueba “página de detalles” (UT-10) 81

5.1. Propiedades configurables de un `provider` 93

ÍNDICE DE CÓDIGOS

1. Ejemplo de inyección de dependencias en un servicio 37
2. Métodos de identificadores de `AbstractEntity` 42
3. Método `Relation::getOther($subject)` 44
4. Métodos de relaciones de `AbstractEntity` 45
5. Anotaciones ORM de `AbstractEntity` 47
6. Ejemplo de relación ORM en `AbstractEntity` 48
7. Implementación de relaciones en `AbstractEntity` 49
8. Ejemplo de escuchador ORM en `AbstractEntity` 50
9. Combinación de resultados de una búsqueda 52
10. Listado de *tokens* para consulta de ejemplo 55
11. Listado de *tokens* para consulta avanzada de ejemplo 56
12. Listado de *tokens* extendidos para consulta avanzada de ejemplo . . 56
13. Método `SearchEngine::getResultsFromProviders()` 58
14. Método `AbstractProvider::configure()` simplificado 58
15. Generación de consulta de búsqueda para proveedores externos . . 59
16. Generación de mapa de índices para agrupación 60
17. Eliminación de índices duplicados para agrupación 60
18. Ejemplo de mapa de índices para agrupación 61
19. Ejemplo de construcción de `$parsedIds` 62
20. Controlador de la página principal 63
21. Extracto de `base.html.twig` 64
22. Extracto de `homepage.html.twig` 64
23. Propiedades de la variable global `rosetta` 65
24. Obtención del fichero de un mapa desde el índice 67
25. Ejemplo de plantilla Twig con textos localizables 68
26. Extracto del fichero `translations/messages.es.json` 69

27. Extracto de `LocaleListener` 69
28. Comandos para despliegue de Rosetta usando Docker 86
29. Modificación de `docker-compose.yml` para cambiar configuración 87
30. Comandos para actualización de Rosetta usando Docker 87
31. Comandos para instalación de Node.js® 89
32. Comandos para instalación de Yarn 89
33. Comandos para instalación de PHP 89
34. Comandos para instalación de Composer 89
35. Comandos para compilación de YAZ 90
36. Comando para instalación de MariaDB 90
37. Comando para descargar Rosetta 90
38. Comandos para compilar Rosetta 90
39. Comando para instalar `apache-pack` en Symfony 91
40. Configuración de nginx recomendada 91
41. Contenido por defecto del fichero `_theme.scss` 94
42. Ejemplo de mapa de Rosetta 96

REFERENCIAS

[1] Ranganathan, S., Sivaswamy Aiyer, P., & Sayers, W. (2006). *Las cinco leyes de ciencia de la bilbiotecología*. Nueva Delhi [India]: Ess Ess Publications.

[2] *MARC standards* (2019). Disponible en `https://en.wikipedia.org/wiki/MARC_standards` [Consultado en mayo de 2019].

[3] Wood, D., Lanthaler M., & Cyganiak, R. (2014). *RDF 1.1 Concepts and Abstract Syntax*. Disponible en `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/` [Consultado en mayo de 2019].

[4] *Schema.org* (2019). Disponible en `https://en.wikipedia.org/wiki/Schema.org` [Consultado en mayo de 2019].

[5] *Gitflow Workflow* (2017). Disponible en `https://es.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow` [Consultado en mayo de 2019].

[6] *Ex Libris Announces the Cloud-Based Alma Library Management Service* (6 de enero de 2011). Disponible en `https://www.exlibrisgroup.com/press-release/ex-libris-announces-the-cloud-based-alma-library-management-service/` [Consultado en mayo de 2019].

[7] Varios autores (2019). *Symfony: The Best Practices Book*. [ebook] Disponible en `https://symfony.com/pdf/Symfony_best_practices_4.2.pdf` [Consultado en mayo de 2019].