



Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Lyncex: describiendo una aplicación web como conocimiento

Alumno/a: Adrián Arroyo Calle

Tutor/es/as: Miguel Ángel Martínez Prieto
María Aránzazu Simón Hurtado

Resumen

Una plataforma de desarrollo de aplicaciones web basada en Prolog y RDF. La plataforma combina un servidor web y una base de datos de tripletas. Sobre la base de datos se almacena tanto los datos propios de la aplicación, como el comportamiento que debe tener la aplicación web. De este modo, la programación de la aplicación web se realiza modificando las propias tripletas de la base de datos, como si fuesen otro dato más, a través de una API HTTP.

Abstract

A development platform for web applications based on Prolog and RDF. The platform combines a web server and a triplestore. The triplestore contains the data of the application itself along with the webapp, defined with triples also. The application programming is done by modifying the triples in the triplestore, as if they were normal data, through an HTTP API.

Índice general

Resumen	3
Abstract	5
1. Introducción	3
1.1. Motivación	3
1.2. Objetivos	3
1.3. Alcance	4
1.4. Aplicación de ejemplo (visión general)	4
1.5. Estructura de la memoria	5
2. Conocimientos previos	7
2.1. Web Semántica	7
2.1.1. RDF	7
2.1.2. RDF Schema y ontologías	10
2.1.3. SPARQL	12
2.2. Prolog	13
2.2.1. Base de conocimiento	13
2.2.2. Operadores y aritmética	15
2.2.3. Listas y diccionarios	16
2.2.4. Metapredicados	17
2.2.5. RDF	18
3. Estado del Arte	21
3.1. Frameworks web	21

3.1.1. Django	21
3.2. Bases de datos	22
3.2.1. Apache CouchDB	22
3.3. Triplestores	22
3.3.1. TerminusDB	23
3.3.2. Apache Jena	24
3.3.3. Virtuoso	25
3.3.4. ClioPatria	25
3.4. Análisis	26
4. Planificación	29
4.1. Scrum	29
4.1.1. Introducción	29
4.1.2. Actores	29
4.1.3. Eventos	30
4.1.4. Artefactos	30
4.2. Metodología de trabajo	31
4.3. Estimación del esfuerzo	32
4.4. Plan de gestión de riesgos	32
4.5. Presupuesto	35
5. Análisis y diseño	37
5.1. Historias de usuario	37
5.2. Requisitos	38
5.2.1. Requisitos funcionales	38
5.2.2. Requisitos no funcionales	40
5.3. Diseño de Lyncex	40
5.3.1. Arquitectura	40
5.3.2. API	40
5.4. Diseño de la ontología	42

5.4.1. Controladores	42
5.4.2. Parámetros	44
5.4.3. Queries	44
5.4.4. Handlers	45
6. Implementación	57
6.1. Herramientas de desarrollo	57
6.1.1. SWI Prolog	57
6.1.2. Visual Studio Code	57
6.1.3. Sistema operativo	57
6.1.4. Docker	58
6.1.5. Git y GitHub	58
6.1.6. Behave	58
6.2. Detalles de implementación	59
6.2.1. Estructura física del proyecto	59
6.2.2. Bugs localizados	61
6.2.3. Dynamic	61
7. Validación y pruebas	63
7.1. Pruebas manuales	63
7.2. Tests unitarios	63
7.3. Test E2E	64
7.4. GitHub Actions	65
7.5. Conclusiones	66
8. Aplicación de ejemplo	69
8.1. Objetivo y alcance	69
8.2. Análisis y Diseño	69
8.3. Arquitectura	70
8.4. Componentes	70
8.5. Implementación	72

8.6. Validación y pruebas	72
8.7. Tour	72
8.8. Conclusiones	76
9. Manuales	83
9.1. Manual de instalación	83
9.1.1. Docker	83
9.1.2. Tests	84
9.2. Manual de usuario	85
10. Conclusiones y trabajo futuro	87
10.1. Conclusiones	87
10.2. Trabajo futuro	88
10.3. Palabras finales	88
Apéndices	89
Apéndice A. Ontología	91
Apéndice B. Criterios de aceptación en Gherkin	97
Bibliografía	105

Índice de figuras

1.1. Dependencias entre las historias de usuario	5
2.1. Representación visual de las tripletas de Feynman	9
2.2. Diagrama de las tripletas expresadas en el ejemplo comic	11
3.1. Pantalla de administración de TerminusDB	23
3.2. Interfaz web de Virtuoso	25
3.3. Resultado de consulta SPARQL en ClioPatria	26
4.1. Matriz de riesgos	34
5.1. Arquitectura de Lyncex	41
5.2. Ontología de Lyncex	46
5.3. Diagrama de secuencia de la comprobación de acceso	47
5.4. Diagrama de secuencia del ContentController	48
5.5. Diagrama de secuencia del TemplateController	49
5.6. Diagrama de secuencia del FormController ante un GET	50
5.7. Diagrama de secuencia del FormController ante un POST	51
5.8. Diagrama de secuencia del LoginController ante un GET	52
5.9. Diagrama de secuencia del LoginController ante un POST	53
5.10. Resolución de las queries	54
5.11. Resolución de los handlers	55
6.1. Relación ficheros-componentes	60
8.1. Diagrama de la ontología vCard http://owlgred.lumii.lv/online_visualization/2pt2	71

8.2. HomePage de BiblioCyL	77
8.3. BookListPage	78
8.4. BookPage	79
8.5. LoginPage	79
8.6. BookForm	80
8.7. BookForm edición y borrado	81

Índice de tablas

3.1. Comparativa de características entre diferente software	26
4.1. Análisis de riesgos	33
4.2. Plan de riesgos	34

Capítulo 1

Introducción

1.1. Motivación

En ciencias de la computación, de forma recurrente se distingue entre código, lo que va a ejecutar la máquina, y datos. Esta diferencia, aunque pueda resultar evidente, es innecesaria, ya que el código no deja de ser dato, solo que con una semántica diferente. Von Neumann, en su modelo de computadora[34], elimina las diferencias a nivel de hardware entre código y datos, de modo muy exitoso, hasta tal punto que esta idea sigue siendo la base de los procesadores modernos actuales.

Hoy día, en la creación de aplicaciones web, se separa por un lado el código y por otro los datos que van a circular a través de él. Pero, ¿sería posible plantear una aplicación web descrita de forma comparable a como se describen los datos? En este Trabajo Fin de Grado exploraremos esta idea mediante un framework o plataforma de desarrollo que sirva como prueba de concepto. El servidor web pasa a ser una base de datos, donde las diferencias entre código y datos son puramente semánticas. Para ello usaremos tecnología madura como RDF para la representación de la información.[11]

De este modo, podríamos resumir esta plataforma como una base de datos y a la vez un servidor web configurable a través del contenido semántico de la propia base de datos.

La idea surge principalmente de estudiar la base de datos CouchDB y su concepto de integrar código ejecutable por HTTP en la base de datos. Apache CouchDB es una base de datos NoSQL de tipo documental. Los documentos almacenados son de tipo JSON, generalmente sin esquema. Sin embargo, hay ciertos documentos especiales que representan vistas de la base de datos. Estas se programan en JavaScript y debido a su flexibilidad, se pueden llegar a implementar aplicaciones completas bajo ese modelo, en particular aplicaciones CRUD.[8]

1.2. Objetivos

El objetivo principal de este trabajo es construir una entorno de desarrollo que implemente el concepto de base de datos y servidor web en un mismo entorno, usando tripletas para su repre-

sentación. Sobre esta plataforma deberán poder ejecutarse aplicaciones simples, y la llamaremos a partir de ahora Lyncex.

La implementación resultante no deberá ser de un nivel de producción, pero deberá ser útil para validar los conceptos con los que se va a trabajar, así como el estado de ciertas tecnologías (Prolog y RDF) con relativa poca popularidad en el mundo empresarial.[16]

Además de la implementación en código, se creará una ontología que defina el funcionamiento de la plataforma de desarrollo. Esta ontología es una interfaz sobre la que un trabajo posterior podría diseñar implementaciones alternativas, mejorando el código original o extendiendo la funcionalidad.

En este proyecto se reforzarán los conocimientos del lenguaje Prolog, en concreto en una vertiente altamente práctica, como es su utilidad para desarrollo de aplicaciones web.

1.3. Alcance

Para cumplir los objetivos identificamos cinco historias de usuario diferentes. Cada historia de usuario refleja un conjunto de funcionalidades que debemos tener implementadas para dar por completada la historia de usuario. Más adelante entraremos en el detalle de los requisitos que llevan asociados cada una de estas historias de usuario, pero su enumeración y dependencias se muestran en la figura 1.1.

De estas historias, cuatro de ellas son un incremento funcional del mismo sistema, Lyncex, y se debe completar una antes de pasar a la siguiente. La quinta, el desarrollo de la ontología, se realiza de forma paralela al resto del sistema.

El proyecto no trata de crear ninguna base de datos desde cero, ni enfrentarse a muchos de los retos comunes que existen en este tipo de sistemas como restricciones de rendimiento o problemas de escalabilidad y sincronización. Tampoco intenta crear un framework web listo para ir a producción, ya que siguen faltando componentes esenciales de cualquier framework web.

El proyecto deberá estar finalizado antes de la fecha límite, del 10 de julio de 2020, tanto la implementación como esta memoria.

1.4. Aplicación de ejemplo (visión general)

De nada serviría una plataforma de desarrollo completa sin aplicaciones que la utilicen. Para demostrar la funcionalidad del sistema, en entornos reales, se diseñará una aplicación basada en datos abiertos ya existentes.

Se usarán los datos de Bibliotecas de Castilla y León, en formato RDF/XML y se realizará un ejemplo práctico de aplicación modelo que usa Lyncex.

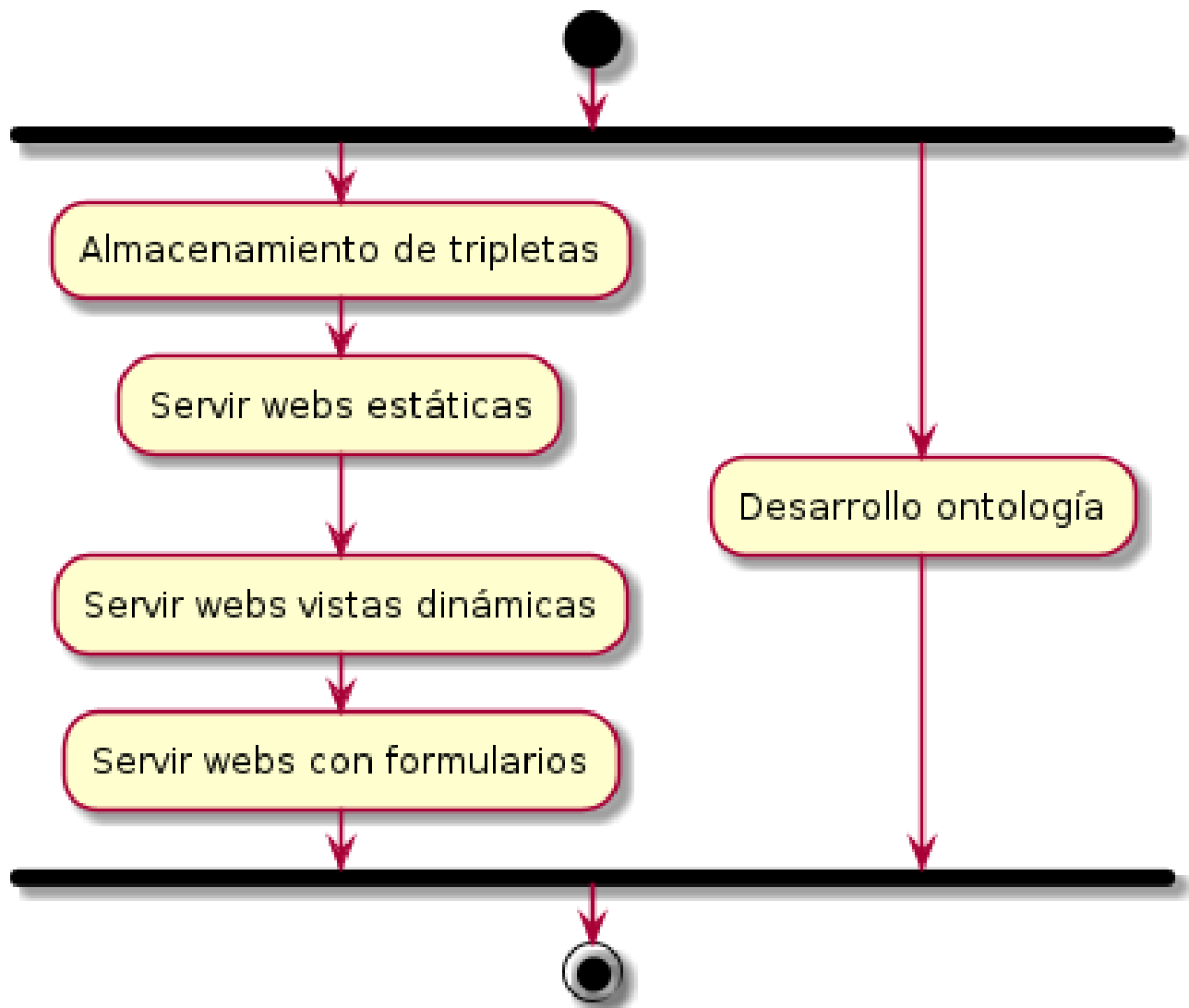


Figura 1.1: Dependencias entre las historias de usuario

1.5. Estructura de la memoria

La memoria se estructura en diferentes capítulos. En primer lugar se ubica el capítulo *Conocimientos Previos*, donde se repasarán conceptos y tecnologías necesarias para entender el trabajo en su totalidad, desde una base técnica pero no especializada. Posteriormente, en el *Estado del Arte*, se realizará el análisis de diferentes tecnologías similares, que cubren parte de los objetivos de este trabajo de alguna u otra forma.

Posteriormente se tratará la *Planificación* en el capítulo homónimo. Se hará un inciso en la metodología Scrum.

Inmediatamente después está el capítulo *Análisis y diseño*. En la etapa de análisis se descompondrán los requisitos del proyecto y las historias de usuario a realizar. La etapa de diseño, tanto del propio software, como de la ontología que lo acompaña.

A continuación, se encuentra el capítulo *Implementación*. En este capítulo primero veremos

herramientas que se han usado para la construcción de este proyecto. Después la implementación del software en sí, haciendo especial hincapié en aquellos elementos que no sean del todo evidentes para el lector.

El capítulo dedicado a *Validación y pruebas* se ubica inmediatamente después. En este capítulo se describirán las pruebas realizadas y la metodología empleada para comprobar que, efectivamente, se está implementando correctamente el software.

Le sigue el capítulo dedicado a la *Aplicación de ejemplo*, donde se llevará a cabo un mini proyecto (objetivo, análisis, diseño, implementación y pruebas) para demostrar la viabilidad del software como plataforma de desarrollo para otras aplicaciones y poder identificar posibles inconvenientes prácticos que se hayan encontrado.

A modo de referencia, el capítulo *Manuales*, incluye instrucciones de instalación y de uso de Lyncex.

Por último, en *Conclusiones y trabajo futuro* se hablará sobre las conclusiones que podemos extraer del proyecto así como trabajo que podría realizarse en el futuro siguiendo por este camino.

Capítulo 2

Conocimientos previos

En este capítulo trataremos de explicar de la forma más clara posible, conocimientos que creemos necesarios para entender el proyecto en su totalidad.

El contenido se divide en dos secciones: web semántica y Prolog. En la web semántica veremos RDF, RDF Schema y SPARQL, esta sección está más enfocada a la parte de almacenamiento e interacción exterior de la plataforma. La sección de Prolog es un tutorial rápido para conocer el lenguaje sobre el que se va a implementar Lyncex.

2.1. Web Semántica

La web semántica es un concepto amplio[28], surgido a finales de los 90 y en un principio impulsado por Tim Berners-Lee y su organización, el W3C. La idea principal es dar un paso cualitativo más en el acceso y descripción de datos respecto al modelo de la web HTML. Dentro de este paraguas existen numerosos proyectos como RDF[18], SPARQL[30], OWL[29], ... En este proyecto solo utilizamos algunas de ellas.

2.1.1. RDF

RDF es una de las tecnologías base, sobre las que se asienta prácticamente toda la web semántica. Se trata de un modelo de representación de la información basado en tripletas.[18]

Tripletas

Las tripletas son un modo de representar la información mediante ternas ordenadas de datos. Es uno de los modos fundamentales de representación de información

En una tripleta, los elementos se denominan, en este orden, sujeto, predicado y objeto. La estructura básica sigue la de la gramática del lenguaje humano para afirmar hechos. El sujeto es el ente sobre el que vamos a afirmar algo, el predicado es la acción y/o propiedad del sujeto, y el objeto es el contenido de la propiedad que se define. Veamos algunos ejemplos de este concepto:

```
maria -> likes -> chocolate
maria -> is -> human
adrian -> likes -> maria
```

Con estos tres elementos, y suponiendo que podemos combinar sujetos, predicados y objetos en otras tripletas, podemos crear grandes redes interrelacionadas, ideales para el almacenamiento de conocimiento. Estas redes se denominan grafos.

¿Por qué ternas? ¿Por qué tres elementos y no dos por ejemplo? Las ternas es la base ideal para estructurar la información. Con tres grados de libertad en nuestras expresiones básicas son suficientes para un sistema autodescriptivo, tres tiene la mayor densidad de información de cualquier entero. Podemos construir un conjunto de triples que describa cualquier cosa descriptible. Las relaciones binarias, por otro lado, que son la base de las tablas y bases de datos tradicionales, nunca pueden ser autodescriptivas: siempre necesitarán alguna lógica externa al sistema para interpretarlas, es decir, dos grados de libertad son insuficientes.[11]

Este sistema, además, tiene varias ventajas, por ejemplo, la duplicación de las tripletas no es un problema, ya que simplemente se reafirma lo mismo una y otra vez.

La idea de las tripletas es bastante genérica y existen diferentes implementaciones de la idea. La más popular, sin lugar a dudas, es RDF, que describiremos a continuación. Las tripletas tienen numerosas ventajas pero nunca han llegado a gozar de una amplia popularidad a nivel empresarial.[16] Esto, sumado a la falta de educación en este concepto, ha provocado que actualmente, el uso de tripletas sea minoritario.

Tripletas en RDF

RDF (Resource Description Framework) es un estándar desarrollado por Tim Berners-Lee en los años 90. RDF define un modelo concreto para usar tripletas derivado de la World Wide Web. Para ello se impone que los elementos o términos puedan ser solo de tres tipos: IRIs, blank nodes y literales. Las IRIs son la versión internacionalizada y ampliada de las famosas URL. Cada IRI representa un recurso, tratándose de un identificador universal, y son la base de las interconexiones entre tripletas. En RDF se pueden usar IRIs en el sujeto, el predicado y el objeto. Los literales son datos puros, sin conexión entre sí. Pueden ser cadenas de texto, números enteros, números decimales, etc... Solo los objetos pueden ser literales. Los blank nodes son conexiones anónimas, sin definir explícitamente, entre varias tripletas. Solo pueden ser blank nodes los sujetos y los objetos.

Además, RDF define un marco básico para poder describir, de forma muy elemental, la propia información representada.

RDF se define sin representación textual por defecto, aunque existen varios estándares. Uno de los originales es RDF-XML[24], que usa la sintaxis de XML para definir RDF. Esta sintaxis es especialmente compleja, ya que es verbosa y admite varios modos de uso. Otros estándares definen JSON-LD[17] que usa sintaxis de JSON, otro formato genérico popular o RDFa[1] que se integra en HTML. Sin embargo, también existen sintaxis expresamente diseñadas para RDF, que por lo general son más claras que tratar de adaptar otro formato a RDF. Algunos ejemplos de estas sintaxis son Turtle[23], N3[37], NTriples[26], TriG[5] y HDT[12].

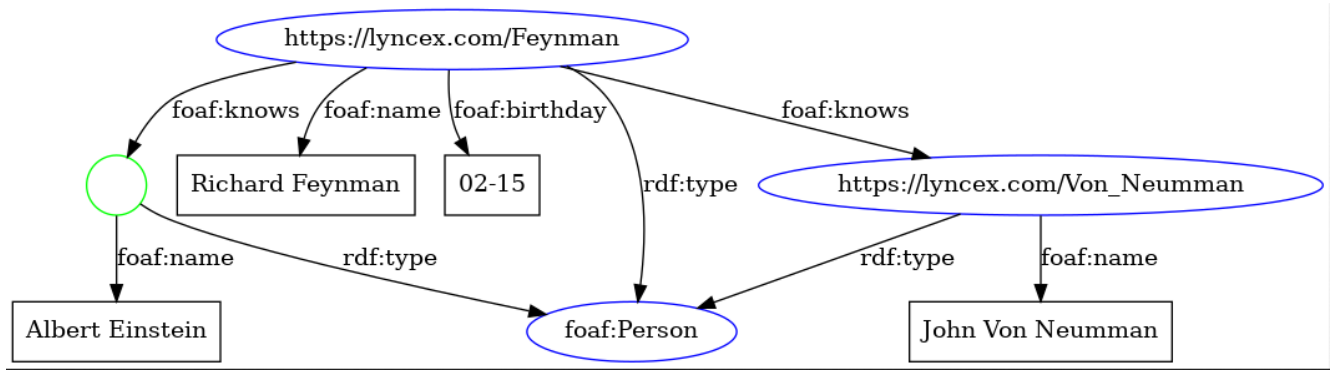


Figura 2.1: Representación visual de las tripletas de Feynman

En Lyncex se ha decidido usar sintaxis Turtle. Ejemplo de esta sintaxis a continuación. La figura 2.1 contiene una representación gráfica del contenido representado.

```
@base <https://lyncex.com/lyncex#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

```
<Feynman>
  a foaf:Person ;
  foaf:name "Richard Feynman" ;
  foaf:birthday "02-15" ;
  foaf:knows [
    a foaf:Person ;
    foaf:name "Albert Einstein"
  ] ;
  foaf:knows <Von_Neumman> .
```

```
<Von_Neumman>
  a foaf:Person ;
  foaf:name "John Von Neuman" .
```

```
https://lyncex.com/lyncex#Feynman
  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
  http://xmlns.com/foaf/0.1/Person
https://lyncex.com/lyncex#Feynman
  http://xmlns.com/foaf/0.1/name
  Richard Feynman
https://lyncex.com/lyncex#Feynman
  http://xmlns.com/foaf/0.1/birthday
  "02-15"
https://lyncex.com/lyncex#Feynman
  http://xmlns.com/foaf/0.1/knows
```

```
_:1
https://lyncex.com/lyncex#Feynman
  http://xmlns.com/foaf/0.1/knows
  https://lyncex.com/lyncex#Von_Neumman
_:1
  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
  http://xmlns.com/foaf/0.1/Person
_:1
  http://xmlns.com/foaf/0.1/name
  Albert Einstein
https://lyncex.com/lyncex#Von_Neumman
  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
  http://xmlns.com/foaf/0.1/Person
https://lyncex.com/lyncex#Von_Neumman
  http://xmlns.com/foaf/0.1/name
  John Von Neumman
```

En Turtle primero se inicia definiendo una base para las IRIs y prefijos, para no tener que escribir IRIs largas y poco legibles en el resto del documento. Las IRIs se definen usando los símbolos `<y >`. Las IRIs con prefijo llevan el prefijo y el símbolo `..` Los literales llevan comillas y los blank nodes se definen con `[y]` y en su interior se definen las tripletas teniendo como sujeto el propio blank node.

En Turtle existe una forma sencilla de no repetir el sujeto, útil para afirmar varias cosas sobre un mismo sujeto. Se utiliza el símbolo `;` para iniciar una nueva triplete sobre el mismo sujeto.

Al acabar de afirmar cosas sobre un sujeto, se pone un punto. Un azúcar sintáctico usado en el ejemplo es que la propiedad `a` es equivalente a:

```
http://www.w3.org/1999/02/22-rdf-syntax-ns#type.
```

2.1.2. RDF Schema y ontologías

RDF es un modelo de representación de la información muy potente, sin embargo, tal y como lo hemos definido, puede volverse muy caótico en poco tiempo.

La razón es que no hay ningún requisito per sé de cómo tienen que estructurarse los datos dentro de las tripletas. El principal inconveniente de esto es la interoperabilidad entre diferentes fuentes de información y entre diferentes programas.

Para solucionarlo, una idea que se puede tener es disponer de unos metadatos que informen de cómo se debe manejar esta información. La información que describe la estructura de otra información se denomina ontología. Ya que las ontologías no son más que información, se incorporan como tripletas también y no existe diferencia física de entre las tripletas de una ontología y la de una información cualquiera, la diferencia es puramente semántica.

Existen numerosos lenguajes de ontologías bajo RDF. Uno de los primeros, impulsado como estándar por el W3C es RDF Schema[14]. Existen otros lenguajes más avanzados como OWL[29], que incluyen soporte para descripciones más elaboradas.

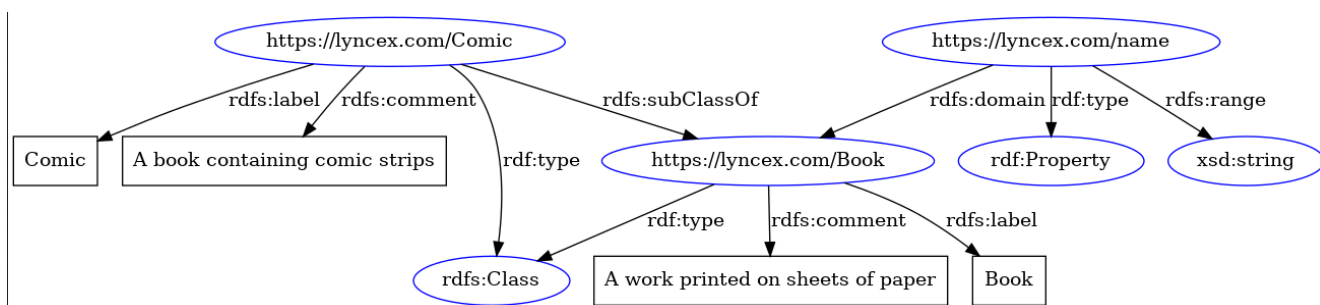


Figura 2.2: Diagrama de las tripletas expresadas en el ejemplo comic

La base de RDF Schema es la clase, o `rdfs:Class`, usando la IRI con prefijos. Las clases a su vez pueden ser subclases de otra con `rdfs:subClassOf`. Las clases podemos usarlas para definir tipos nuevos, que luego se referencian usando la propiedad `rdf:type`.

A su vez, también podemos definir los predicados, también llamados propiedades. Esto ya venía en RDF, bajo el tipo `rdf:Property`, pero RDF Schema añade predicados extras como `rdfs:domain` y `rdfs:range`. El primero permite definir en una tripleta la clase de los sujetos que tengan la propiedad. Y en el caso del range, la clase de valores que se aceptan como objeto. Dicho de otra forma, en una propiedad, `rdfs:domain` comprueba la clase correcta al lado izquierdo (sujeto) y `rdfs:range` al lado derecho (objeto). Las propiedades se pueden heredar también mediante `rdfs:subPropertyOf`.

Con estos elementos podemos crear ontologías con esquemas similares a los que tendríamos en programación orientada a objetos, aunque de forma mucho más flexible.

Veamos un ejemplo de ontología sencilla definida con RDF Schema.

```
@base <https://lyncex.com/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

```
<Book>
  a rdfs:Class ;
  rdfs:label "Book" ;
  rdfs:comment "A work printed on sheets of paper" .
```

```
<name>
  a rdf:Property ;
  rdfs:domain <Book> ;
  rdfs:range xsd:string .
```

```
<Comic>
  a rdfs:Class ;
  rdfs:subClassOf <Book> ;
  rdfs:label "Comic" ;
  rdfs:comment "A book containing comic strips" .
```

En este ejemplo se usa además la ontología XMLSchema[13], que provee de tipos básicos

derivados de XML (entre otras cosas).

Una de las ideas básicas de la web semántica es que las ontologías sean reutilizables por distintas partes. Para describir cierta información muy común existen ontologías establecidas tanto de forma general como para nichos específicos (bioinformática, geografía). Algunos de los repositorios de ontologías más importantes son la propia W3C, la iniciativa Dublin Core[7] y Schema.org[19], iniciativa esta última de los grandes buscadores (Google, Bing, Yandex). Estas ontologías pueden ir versionadas.

2.1.3. SPARQL

Otro de los estándares definidos por W3C bajo el paraguas de la web semántica es SPARQL[30]. Se trata de un lenguaje de consulta sobre RDF. Además existe una extensión del estándar SPARQL Update, que permite modificar el contenido de la base de datos. Sin embargo, a pesar de llamarse “Update”, no existe ninguna operación concreta de actualización. Solo existe borrado y creación de nuevas tripletas.

Una consulta SPARQL empieza opcionalmente con prefijos, muy similares a los de Turtle y con el mismo objetivo. Posteriormente le sigue un SELECT, que indica que variables se deben proyectar como resultado. La salida será un listado de una tupla de los valores que hayamos indicado. Si quisiésemos añadir operaciones de agregación, las indicaríamos aquí también.

Las variables empiezan siempre por interrogación y adoptan los valores que cumplen con todas las afirmaciones del WHERE. Las afirmaciones del WHERE, son tripletas que deben existir todas ellas en la base de datos (es decir, es un AND implícito). Para combinar tripletas entre sí podemos usar variables intermedias o property paths, que es una sintaxis más corta. Si queremos excluir algunas tripletas de este AND, podemos usar OPTIONAL, donde se intentará cumplir los criterios, pero si no se cumple, será como si no existiera este bloque. Dentro de los WHERE podemos incluir FILTER, que realizan comprobaciones sobre las variables.

En el siguiente ejemplo se muestra una consulta donde se obtiene el nombre y el autor de un libro (una instancia de la clase lyncex:Book).

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX lyncex: <https://lyncex.com/>

SELECT ?name ?author
WHERE {
    ?id rdf:type lyncex:Book .
    ?id lyncex:name ?name .
    ?id lyncex:author ?author .
}
```

Un ejemplo algo más avanzado, con datos de Wikidata. Esta consulta, muestra los pares de ciudades hermanadas, usando los nombres en español.


```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?city_name ?brother_name
WHERE {
    ?city wdt:P31 wd:Q515 .
    ?city wdt:P17 wd:Q29 .
    ?city rdfs:label ?city_name .
    OPTIONAL {
        ?city wdt:P190/rdfs:label ?brother_name .
        FILTER (lang(?brother_name) = 'es') .
    }
    FILTER (lang(?city_name) = 'es')
}
```

Existen más modos de consulta además de SELECT. En particular ASK, que simplemente verifica si se cumplen las condiciones de un WHERE en la base de datos, y CONSTRUCT, que permite obtener tripletas, en vez de un listado de tuplas.

2.2. Prolog

Prolog^[33] es un lenguaje de programación lógico diseñado por Alain Colmerauer, Philippe Roussel y Robert Kowalski en 1972 en Marsella, Francia. Se trata de uno de los primeros y más populares lenguajes de programación lógica. Este se basa en la lógica de primer orden con una base de conocimiento compuesta de hechos y reglas.

El flujo de ejecución de Prolog es una búsqueda con backtracking. Ante un predicado, Prolog trata de demostrar su veracidad, identificando valores para variables si es necesario. En caso de encontrarse con algo indemostrable, vuelve para atrás deshaciendo todo lo hecho hasta llegar a un punto de elección, donde toma otro valor para las variables o elige otra regla que satisfaga el predicado.

Para una implementación eficiente de estos conceptos, Prolog usa cláusulas Horn y un procedimiento conocido como unificación.

2.2.1. Base de conocimiento

Un programa Prolog se define por su base de conocimiento. La base de conocimiento es una base de datos donde, además de información puramente dicha, se incluyen reglas que permiten operar sobre la información para generar nueva información. El intérprete de Prolog lo único que hace es recorrer su base de conocimiento de arriba a abajo tratando de demostrar el predicado que se le indica.

La forma principal de construir una base de conocimiento es mediante ficheros de texto, sin embargo, esta base de conocimiento también se puede modificar en tiempo de ejecución mediante los predicados `assert` y `retractall`, es por ello que se dice que Prolog es un lenguaje homoicónico, ya que la diferencia entre código y datos es inexistente.

Un código cualquiera está compuesto de comentarios y términos. Los términos pueden ser hechos o reglas. La única diferencia es que las reglas tienen condiciones que tienen que verificarse también para poder verificar la validez del término. Con un hecho, su mera definición es suficiente. A su vez los términos se componen de átomos (simples o compuestos), que son elementos constantes y variables, que pueden adoptar un valor. Los átomos empiezan en minúscula y las variables empiezan por mayúscula.

Este programa incluye las siguientes hechos: Sócrates es humano y Kant es humano. Además incluye la siguiente regla: para que X sea mortal, X tiene que ser humano. Si lo cargamos podremos ejecutar diferentes consultas.

```
human(socrates).
human(kant).
mortal(X) :- human(X).

?- human(socrates). % ¿Es humano Sócrates?
true.

?- human(pepito). % ¿Es humano Pepito?
false.

?- mortal(X). % Todos los mortales
X = socrates ;
X = kant.

?- assertz(god(zeus)). % Añade Zeus es Dios
true.

?- god(X). % Todos los dioses
X = zeus.

?- retractall(god(zeus)). % Elimina Zeus es Dios
true.

?- god(X). % Todos los dioses
false.
```

A pesar de ser un lenguaje enfocado a consultas, estas pueden ser transparentes al usuario, ya que existen muchos tipos de predicados especiales como, leer de teclado, imprimir pantalla o incluso ejecutar un servidor HTTP.

2.2.2. Operadores y aritmética

En Prolog, muchos operadores tradicionales son ligeramente diferentes por lo que haremos un repaso. El operador lógico AND es simplemente la coma, usado frecuentemente en reglas complejas. El operador lógico OR es el punto y coma, aunque normalmente se redefine otra regla, y se deja que el backtracking salte a la otra regla, ya que queda más legible. Por último, el operador NOT se define como `\+`. En Prolog la negación es negación por fallo. Es decir, tratará como fallo aquello que no puede demostrar como true. Esto solo tiene sentido bajo ciertas suposiciones, algunas de las cuales RDF no tiene, como la de mundo cerrado (RDF es mundo abierto).

En Prolog existen varios operadores con un funcionamiento similar a la igualdad. Por un lado el operador `=` realiza la operación de unificación a ambos lados (simétrico). La operación de unificación, de forma resumida, trata de que los átomos o variables que tengan a los lados coincidan. Por poner un ejemplo sencillo, si en un lado del `=` tenemos un átomo y en otro una variable, la forma de que coincidan es que la variable adopte el valor del átomo y siempre se verificará a no ser que la variable ya hubiese adoptado otro valor antes que no sea el del átomo. Si por ejemplo hay dos átomos, se comprueba que sean exactamente iguales. Esta operación es recursiva, por lo que en términos compuestos, se realiza la misma operación a un nivel más profundo tantas veces como haga falta.

Merece especial mención el caso de las operaciones aritméticas, ya que en Prolog los símbolos aritméticos se quedan formando parte del átomo y no se evalúan, salvo que se pida explícitamente. Por ello, podemos decir que `4+5` no unifica con `3+6`, aunque sus evaluaciones matemáticas sí unificarían entre sí.

Para forzar la evaluación podemos usar el operador `is`, que es asimétrico y evalúa solo el lado derecho. También existe el operador `==` que evalúa de forma simétrica.

Veamos algunos ejemplos de todo esto:

```
% Dos términos unidos por AND. Cada término escribe un átomo
?- write('Un término'), write('Otro término').
Un término Otro término
true.
```

```
% Dos términos unidos por OR. Cada término escribe un átomo
?- write('Un término'); write('Otro término').
Un término
true .
```

```
% Falla a posta
?- fail.
false.
```

```
% Niega el fallo a posta
?- \+ fail.
true.
```

```
%X unifica con Y. Al no tener valores todavía, es cierto.
?- X = Y.
X = Y.
```

```
% Sócrates unifica con X. Al X no tener ningún valor,
%X adopta el valor de Sócrates.
?- socrates = X.
X = socrates.
```

```
% Sócrates unifica con Platón. Son dos átomos diferentes, falla.
?- socrates = platon.
false.
```

```
% Dos términos compuestos unifican entre sí, dando valor a las
% variables Name y Author. Esto es porque siempre que hay dos
% átomos, coinciden (book, book) y cuando hay un átomo y una
% variable sin valor siempre se adopta el valor del átomo.
?- book(Name, 'Cervantes') = book('Don Quijote', Author).
Name = 'Don Quijote',
Author = 'Cervantes'.
```

```
% Los términos no unifican ya que usan átomos diferentes
?- 4+5=3+6.
false.
```

```
% Se evalúa el lado derecho y se unifica con X
?- X is 3+6.
X = 9.
```

```
% Se evalúan ambos lados y se comprueba si se unifica
?- 4+5:=3+6.
true.
```

2.2.3. Listas y diccionarios

Aparte de los términos, existen otros tipos de estructuras de datos como las listas y los diccionarios. En el fondo siguen siendo combinaciones de átomos y variables, pero disponen de sintaxis especial.

Las listas se definen usando corchetes, y sus elementos se separan por comas. Además, podemos incluir una barra vertical para separar el primer elemento (head) del resto (tail). Una regla de suma de una lista sería así:

```
% Caso base. Para la lista vacía, la suma es 0
sum([], 0).
```

```
% Caso recursivo. Para la lista con elementos, se
% divide el primer elemento H, del resto T.
% La suma es Out.
sum([H|T], Out) :-
    sum(T, X),
    Out is H+X.
```

```
?- sum([1,2,3], X).
X = 6.
```

Algunos términos de interés para trabajar con listas son: `member`, `append`, `length`, `nth1` y `nth0`.

Los diccionarios por otro lado son estructuras de datos clave valor. Se definen mediante llaves y se separan los pares por coma, mientras que cada elemento del par se separa por dos puntos. Los diccionarios en Prolog además pueden tener una etiqueta, aunque si no queremos, podemos usar la barra baja para aceptar todo tipo de etiqueta. Existe un acceso mediante la sintaxis punto, pero además existen diversos términos por defecto como `dict_create`, `dict_pairs`, `get_dict`, `put_dict`, `dicts_join` y operadores como `:<y>` y `>:<`.

```
% Se unifica el diccionario con X
% y se escribe los átomos almacenados en los campos
?- X = _{world:mundo, hello:hola},
    write(X.hello),
    write(X.world).
holamundo
X = _9158{hello:hola, world:mundo}.
```

2.2.4. Metapredicados

Uno de los detalles más importantes de Prolog son sus capacidades de metaprogramación. Además de la flexibilidad que ofrecen, permiten hacer código mucho más compacto.

El metapredicado principal es `call`, que permite llamar al término que deseemos. La interfaz de comandos de Prolog no deja de ser un `call` por debajo. La flexibilidad es que el término al que llamamos no tiene por qué ser siempre el mismo. Sin embargo, este término suele ser demasiado bajo nivel para la mayoría de casos.

Por encima tenemos `forall`. Este toma primero un término que generan varios puntos de elección (por ejemplo, con `member`, generará un punto por cada elemento de la lista). El segundo parámetro es un término que tiene que ser verdadero para todos los puntos de elección.

```
% Para cada elemento de la lista comprueba que es impar
```

```
?- List = [1,3,5,7], forall(member(X, List), 1 is X mod 2).  
List = [1, 3, 5, 7].
```

```
% Para cada elemento de la lista comprueba que es par  
?- List = [1,3,5,7], forall(member(X, List), 0 is X mod 2).  
false.
```

El término `findall` nos permite encontrar todos los posibles resultados de un término. Para ello, primero indicaremos la variable que vamos a tratar de obtener todos sus valores, después el término en sí que genera soluciones diferentes para las variables y por último la lista donde se van a almacenar las soluciones.

```
% Encuentra a todos los humanos  
?- findall(Name, human(Name), Names).  
Names = [socrates, kant].
```

Por último, `maplist` nos permite realizar un map sobre una lista. En primera posición marcamos el término que se va a aplicar para cada elemento, con N parámetros. A continuación N listas correspondientes a los N parámetros. Pueden ser tanto de entrada como de salida.

```
% Escribe la lista  
?- maplist(write, [1,2,3]).  
123  
true.
```

```
% Suma cada sublista  
?- maplist(sum, [[1,2,3],[4,5,6]], OutList).  
OutList = [6, 15].  
% Siendo sum el predicado definido anteriormente
```

Otros metapredicados de interés pueden ser `include`, `findnsols`, `bagof` o `setof`.

2.2.5. RDF

Por último, vamos a comentar las facilidades que tiene Prolog para trabajar con RDF.

El término esencial es `rdf`, que representa el estado de las tripletas. `rdf_assert` y `rdf_retractall` permiten añadir y eliminar tripletas. También admite un sistema de prefijos similares a otros formatos de RDF. También existen términos para trabajar con RDF Schema como `rdfs_class_property` (para comprobar, generar las propiedades asociadas a una clase) o `rdfs_individual_of` (para comprobar si un recurso es individuo de una clase).

Además, la librería de RDF soporta diferentes tipos de literales mediante la sintaxis `^^`. Estos tipos suelen estar definidos bajo la ontología de XML Schema, que ya hemos mencionado antes.

Veamos un ejemplo de su uso añadiendo una tripleta al triplestore y luego consultándola.

```
% Importar la librería de RDF
?- use_module(library(semweb/rdf11)).
true.

% Registrar el prefijo foaf
?- rdf_register_prefix(foaf, 'http://xmlns.com/foaf/0.1/').
true.

% Añadir la tripleta al triplestore
?- rdf_assert('http://example.com/Cervantes',
             foaf:name,
             "Miguel de Cervantes Saavedra").
true.

% Obtener todas las tripletas del triplestore
?- rdf(S,P,O).
S = 'http://example.com/Cervantes',
P = 'http://xmlns.com/foaf/0.1/name',
O = "Miguel de Cervantes Saavedra"^^'http://www.w3.org/2001/XMLSchema
  #string'.

% Obtener todas las tripltas del triplestore cuyo objeto
% sea de tipo xsd:string
?- rdf(S,P,O^^xsd:string).
S = 'http://example.com/Cervantes',
P = 'http://xmlns.com/foaf/0.1/name',
O = "Miguel de Cervantes Saavedra".
```


Capítulo 3

Estado del Arte

Antes de adentrarnos en los detalles de Lyncex conviene recapitular ideas similares ya existentes así como herramientas que nos puedan ayudar a comprender los fundamentos de este proyecto.

3.1. Frameworks web

3.1.1. Django

Django es uno de los frameworks web más populares dentro del mundo Python y es una referencia dentro del mundo de los frameworks web por su potencia y claridad de código[10]. Fuertemente inspirado por **Ruby on Rails**, adopta de él su filosofía DRY (Don't Repeat Yourself) y su *convention over configuration*.

DRY[36] es un principio de desarrollo de software que promueve que no exista código duplicado en un mismo proyecto. Se deben obviar las redundancias usando módulos, paquetes y librerías, aunque sean muy pequeñas, de modo que se puedan evitar inconsistencias futuras entre dos códigos que deben evolucionar a la vez ante un cambio.

Por otro lado, *convention over configuration*[35] es un principio de diseño que adoptan numerosos frameworks de cara al programador. La idea es que el framework tenga por defecto opciones muy bien ajustadas y que cubran la mayoría de los casos. Así, solo es necesario configurar aquellas cosas que se salgan de la norma por defecto del framework y que al, estar bien elegidos, deberían ser un número reducido del total.

El framework trabaja con bases de datos relacionales principalmente, pero es una buena base para diseñar la semántica del servidor web.

Django sigue una arquitectura MTV, es decir, Model-Template-View. Es bastante similar al popular MVC, aunque según sus creadores, el controlador es el propio framework. En Django el modelo se define como una clase de Python, que gracias al ORM (Object-Relational Mapping), tiene persistencia en la base de datos. La parte de las plantillas se programa con un lenguaje similar a HTML, pero con capacidad de mostrar variables y de cierta lógica (condicional, bucle) con metaetiquetas. La parte de vista es código Python puro. Existe un fichero especial llamado `urls.py` que contiene un listado que relaciona las URLs con las vistas.

Las vistas pueden acceder a los parámetros GET o POST directamente y operar sobre ellos, así como a la sesión del usuario que visita la página. Las vistas en Django pueden protegerse mediante un sistema de autenticación sencillo pero seguro, que permite bloquear fácilmente o redirigir si el usuario no ha iniciado sesión o si no tiene los permisos suficientes. Estas características de Django tienen el denominador común de que son implementadas usando middleware, es decir, componentes del framework que se introducen entre la petición original y la vista.

Otra característica interesante de Django, apoyada en el middleware, es la gestión de formularios. Se definen en Django como una clase más, parecida a un modelo de base de datos (de hecho, se puede hacer que sea la misma clase). Con esta clase, Django puede generar parte de la plantilla automáticamente. Posteriormente, en el método POST, el middleware puede limpiar y validar los campos, de modo que a la vista llegan ya los datos limpios.

Por último, existe un componente llamado **Django REST Framework**, que añade funcionalidad para diseñar una API REST con los mismos patrones y facilidades que el desarrollo web HTML.

3.2. Bases de datos

3.2.1. Apache CouchDB

Apache CouchDB es una base de datos NoSQL de tipo documental, que almacena los datos en JSON[2]. Su peculiaridad respecto a otras bases de datos similares, como MongoDB, viene por la forma de acceder a los datos. Aunque en las últimas versiones CouchDB también dispone de un lenguaje de consulta, la forma original de acceder a la información era programando lo que se denominan vistas, en JavaScript. Estas vistas generan un endpoint HTTP, que al ser llamado ejecuta el código JavaScript, en dos pasos. Primero un paso de “map”, para seleccionar los datos y realizar alguna transformación individual. Posteriormente un paso de “reduce”, permite realizar agregaciones. El código JavaScript que se ejecuta se almacena en la base de datos junto al resto de documentos.

Para interactuar con la base de datos se dispone de una API HTTP, tanto para crear como para borrar y o modificar los documentos. Además, dispone de un sistema de replicación entre nodos, eventualmente consistente y relativamente simple, de forma que hay otros proyectos que implementan el mismo protocolo (PouchDB para navegadores web por ejemplo).

3.3. Triplestores

En esta sección analizaremos las bases de datos del tipo Triplestore. Estas bases de datos son muy similares, por no decir conceptualmente iguales, a las bases de datos de grafos, sin embargo, solo vamos a analizar triplestores que soporten explícitamente el modelo de RDF.

3.3.1. TerminusDB

TerminusDB es una base de datos programada en Prolog/Rust que comparte junto a CouchDB la característica de ser de tipo NoSQL[9]. Sin embargo, TerminusDB almacena tripletas RDF. TerminusDB dispone de un esquema basado en OWL, muy potente y extendido por sus desarrolladores, que permite modelar ontologías muy completas, con relaciones avanzadas y restricciones potentes. Además incluye unas vistas basadas en el concepto de documento, que generan formularios para el administrador y una API que permite interactuar en bloque con ciertas tripletas, tal y como si fuese un documento de CouchDB.

Las consultas se realizan mediante un lenguaje llamado WOQL, basado en JSON-LD. Además existen librerías para JavaScript y Python. Para editar las ontologías se puede usar su editor web, como se aprecia en la figura 3.1.

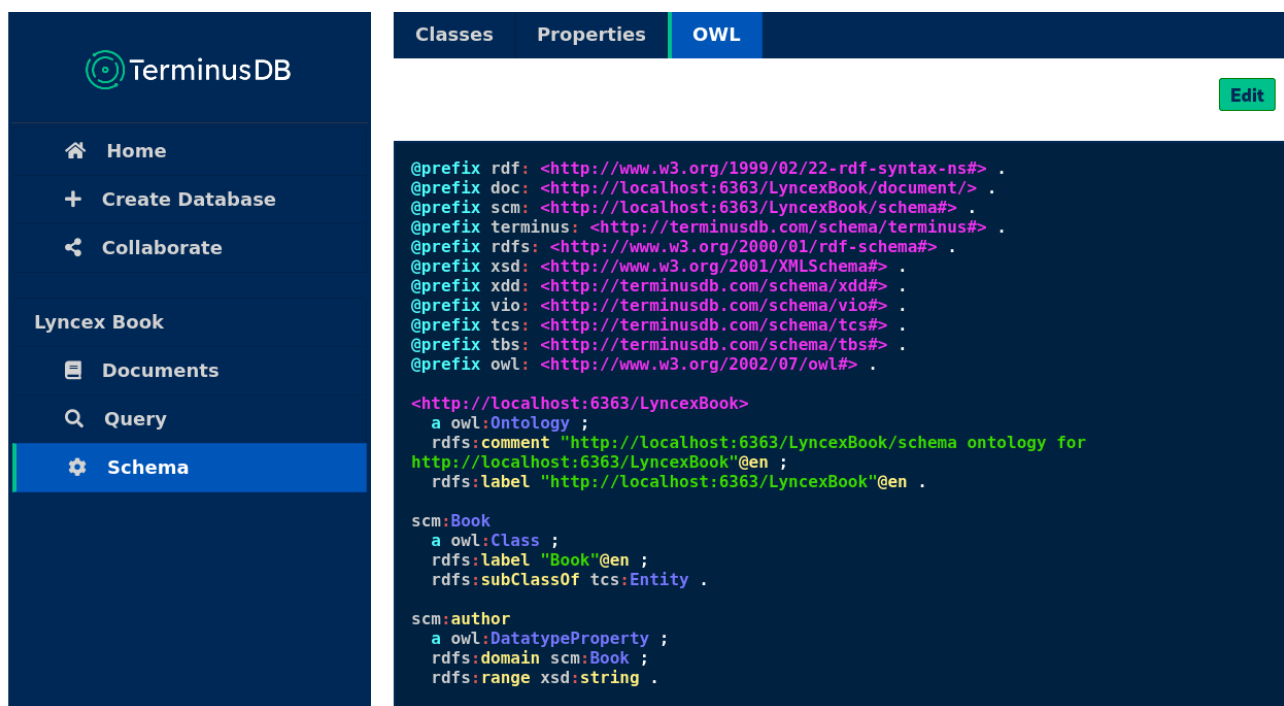


Figura 3.1: Pantalla de administración de TerminusDB

Un ejemplo de WOQL.js y de WOQL en JSON-LD que muestra todas las tripletas cuyo sujeto es de tipo `scm:Book` podemos encontrarlo a continuación.

```
// WOQL.js
WOQL.and(
    WOQL.triple("v:Subject","type","scm:Book"),
    WOQL.opt().triple("v:Subject","v:Property","v:Value")
)

// WOQL
{
  "and": [
    {
      "triple": [
        "v:Subject",
        "rdf:type",
        "scm:Book"
      ]
    },
    {
      "opt": [
        {
          "triple": [
            "v:Subject",
            "v:Property",
            "v:Value"
          ]
        }
      ]
    }
  ]
}
```

3.3.2. Apache Jena

Apache Jena es una base de datos de tripletas RDF, una de las más maduras si nos atenemos a su longevidad (Jena 1.0 es del año 2000)[3]. Se trata de un proyecto con diversos componentes. Por un lado una API en Java para tratar con RDF, un motor de alto rendimiento SPARQL (ARQ) con soporte al estándar SPARQL Update, un almacenamiento persistente llamado TDB y un servidor que combina todo a través de una API, Fuseki. Además dispone de una API para tratar con ontologías OWL y una API de inferencias para usar diferentes razonadores RDF Schema y OWL. Las búsquedas sobre texto pueden realizarse usando por debajo el motor Apache Lucene, usado en otras bases de datos de búsqueda como Elasticsearch.

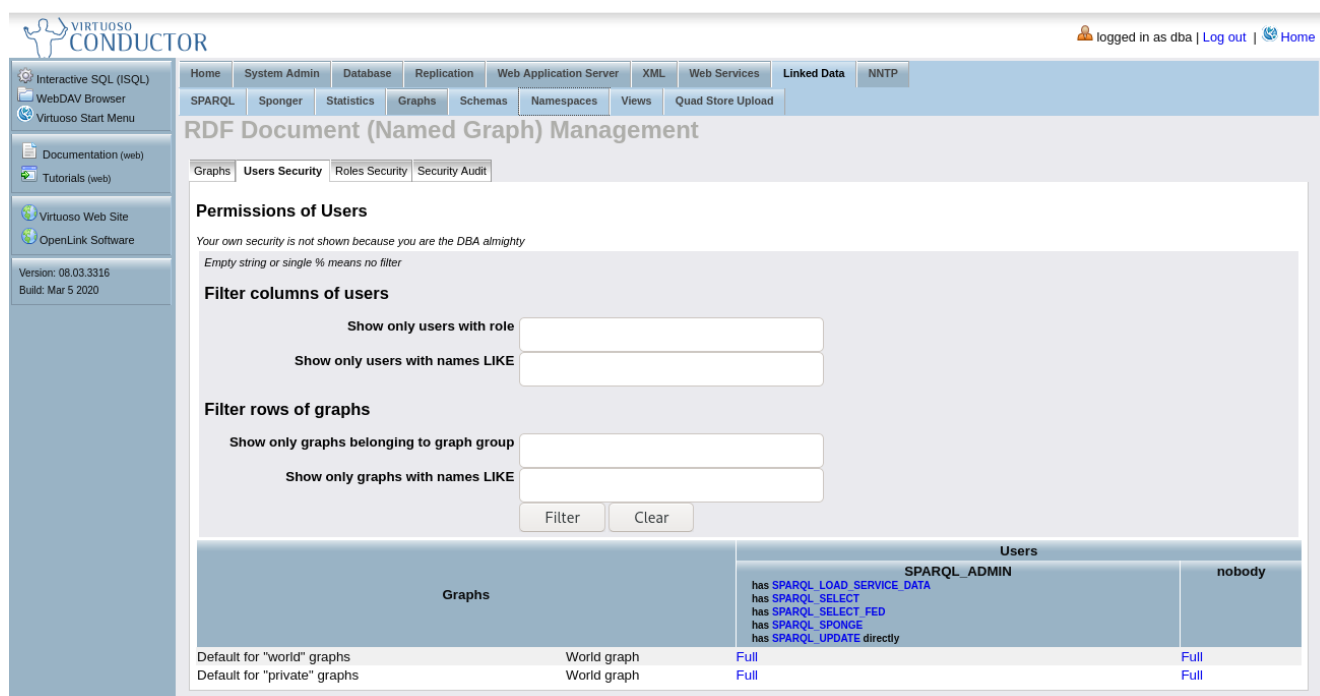


Figura 3.2: Interfaz web de Virtuoso

3.3.3. Virtuoso

OpenLink Virtuoso es una base de datos multimodelo, entre los modelos soportan el almacenamiento de tripletas[22].

Una de sus características principales es que permite que estos modelos interactúen entre sí. De este modo, es posible acceder a contenido mediante RDF/SPARQL que realmente está almacenado como una tabla SQL. También dispone de servicios web, configurables desde la propia interfaz web. Estos servicios web sin embargo, están muy enfocados a APIs y no a aplicaciones finales. En la figura 3.2 se aprecia la interfaz web de Virtuoso.

Virtuoso es un software de pago, aunque tiene una versión opensource usada detrás de grandes silos de conocimiento como DBPedia o Wikidata. Ambas soportan queries vía SPARQL.

3.3.4. ClioPatria

ClioPatria es una aplicación opensource que combina las librerías de RDF y HTTP de SWI-Prolog para ofrecer una base de datos semántica completa[32]. Soporta queries en SPARQL, SeRQL y en código Prolog. Incluye una potente interfaz de administración web, con soporte a diversos usuarios. Desde esta interfaz también es posible asignar prefijos RDF a nivel global. La interfaz permite cargar tripletas desde tres formatos (RDF/XML, Turtle y NTriples), soportando que estén de forma remota y comprimidos en formatos como Zip. Desde la interfaz también podemos escribir código SPARQL con un editor interactivo y podemos visualizar grafos. El resultado de una consulta SPARQL puede verse en la figura 3.3.

ClioPatria además destaca por tener un buen motor de búsqueda de texto, permitiendo buscar por subcadenas dentro de las tripletas. El servidor es fácilmente extensible mediante un gestor de

x	y	z
http://book.lyncex.com/aarroyoc/dev/lyncex/apps/1191563299709	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2006/vcard/ns#VCard
http://book.lyncex.com/aarroyoc/dev/lyncex/apps/1191563299709	http://www.w3.org/2006/vcard/ns#email	mailto: bibliotecaunamuno@aytopalencia.es
http://book.lyncex.com/aarroyoc/dev/lyncex/apps/1191563299709	http://www.w3.org/2006/vcard/ns#fn	"Biblioteca Pública Municipal "Miguel de Unamuno" de Palencia"
http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl1	http://www.w3.org/2006/vcard/ns#latitude	"42.00368968150856"
http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl1	http://www.w3.org/2006/vcard/ns#longitude	"-4.521281719207764"
http://book.lyncex.com/aarroyoc/dev/lyncex/apps/1191563299709	http://www.w3.org/2006/vcard/ns#geo	http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl1
http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl2	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.w3.org/2006/vcard/ns#Organization
http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl2	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://xmlns.com/foaf/0.1/Agent
http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl3	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://xmlns.com/foaf/0.1/Agent
http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl4	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://xmlns.com/foaf/0.1/Agent
http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl4	http://xmlns.com/foaf/0.1/name	"Bibliotecas"
http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl3	http://purl.org/dc/terms/isPartOf	http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl4
http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl3	http://xmlns.com/foaf/0.1/name	"Palencia"
http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl2	http://purl.org/dc/terms/isPartOf	http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl3
http://adrianistan:3020/data/uploaded/C:\fakepath\bibliocyl.ttl2	http://xmlns.com/foaf/0.1/name	"Biblioteca Pública Municipal "Miguel de Unamuno" de Palencia"

Figura 3.3: Resultado de consulta SPARQL en ClioPatria

paquetes propio, llamado CPACK. Al estar basada la aplicación en componentes independientes, podemos usarlos como base de nuestra aplicación.

3.4. Análisis

Esta tabla analiza las características de las tecnologías anteriores respecto a los objetivos de nuestro proyecto y de posibles extensiones futuras.

	CouchDB	TerminusDB	Django	Jena	Virtuoso	ClioPatria	Lyncex
Almacén tripletas	No	Sí	Sí (1)	Sí	Sí	Sí	Sí
Ontología disponible	No	No	No	No	No	No	Sí
Webs estáticas	Sí	No	Sí	No	No	No	Sí
Webs plantillas	Sí	No	Sí	No	No	No	Sí
Webs CRUD	Sí	No	Sí	No	No	No	Sí
Autenticación	Sí	No	Sí	No	No	No	No
APIs	Sí	No	Sí (2)	No	No	No	No
SPARQL	No	No	No	Sí	Sí	Sí	No

Tabla 3.1: Comparativa de características entre diferente software

Notas:

1. Aunque Django cuenta con un ORM muy potente para trabajar sobre el modelo relacional, nada impide usar otros modelos de almacenamiento.
2. Django por sí solo no está especialmente indicado realizar APIs HTTP (está más enfocado en HTML), es posible usar la librería Django REST Framework para añadir APIs web sobre proyectos Django.

Podemos concluir que no existe ningún software que cumpla el objetivo que nos hemos marcado al inicio del proyecto. Sin embargo, sí existe software que implementa en cierta medida muchas características que serían deseables en este sistema, por lo que aprovecharemos su experiencia, en la medida de lo posible, para construir Lyncex.

Capítulo 4

Planificación

En este capítulo hablaremos sobre el marco de trabajo Scrum[25], primero de forma genérica y luego adaptada a nuestro proyecto en particular. Además, veremos el plan de riesgos.

4.1. Scrum

4.1.1. Introducción

Scrum es un marco de trabajo ágil creado por Ken Schwaber y Jeff Sutherland a principios de los años 90. Fue diseñado para que equipos pequeños pudiesen desarrollar productos complejos de forma efectiva, adaptándose a los cambios de requisitos del producto. El pilar fundamental de Scrum es el empirismo. Esta metodología sostiene que la única forma de obtener información correcta y precisa es mediante la experiencia. Sin embargo, muchas metodologías de gestión de proyectos apenas tienen en cuenta la retroalimentación constante que genera el trabajo que se realiza semana a semana. Los pilares sobre los que se fundamenta Scrum se pueden resumir en: transparencia, inspección y adaptación.

La transparencia implica que todos los procesos deben tener propiedades de interés observables por aquellos a los que les interese. Además, todos los actores implicados deben tener estándares comunes de comunicación y entendimiento en el proyecto.

La inspección implica que todos los artefactos deben poder ser auditables para detectar posibles variaciones respecto al plan establecido.

Por último, la adaptación implica que si un inspector determina que un proceso no sigue el rumbo adecuado, posiblemente generando artefactos alejados del objetivo, se puede y debe ajustar el trabajo.

4.1.2. Actores

En Scrum se definen varios actores dentro del proyecto: Product Owner, Scrum Master y equipo de desarrollo. Todos ellos forman parte del equipo Scrum, el cual es independiente y flexible en su

forma de trabajar.

El **Product Owner** es el encargado de organizar el Product Backlog. Tiene que expresar claramente las tareas que hay que realizar, de forma transparente y asegurándose que el equipo de desarrollo entiende las tareas. Además, debe priorizar las tareas de forma que maximice el valor del producto final y teniendo en cuenta el desempeño de los miembros del equipo de desarrollo. Define las condiciones para que una tarea pueda ser clasificada como completada.

El **equipo de desarrollo** es el equipo de profesionales que debe completar las tareas. Ante un Sprint backlog, deben organizarse entre sí para poder completar la mayoría de tareas allí descritas. Los equipos deben ser multidisciplinarios, y aunque haya personas especializadas en algún punto, es el equipo como conjunto el que responde ante las tareas. El tamaño de los equipos ideal estaría entre 3 y 9 personas.

El **Scrum Master** es la persona encargada de promover el uso de Scrum y de aconsejar a los integrantes del equipo de formas de actuar válidas en Scrum.

4.1.3. Eventos

El evento principal en Scrum es el sprint. Los sprints son periodos de tiempo, de duración inferior a un mes (habitualmente dos semanas), donde se mejora el valor del producto de forma incremental. Los sprints comienzan justo cuando ha acabado el anterior. Durante el sprint no se debe modificar el objetivo del sprint y debe ser considerado como un proyecto independiente. Dentro de los sprints existen varios eventos.

El **sprint planning** es la reunión de todo el equipo, moderada por el Scrum Master que tiene lugar al inicio del sprint. En esta reunión se tienen que decidir de forma consensuada qué tareas van a formar parte del Sprint Backlog (y serán realizadas en ese sprint). En primer lugar se debe tener claro el objetivo del sprint, las tareas seleccionadas del Product Backlog para formar parte del Sprint Backlog deben ser tareas que acerquen a todo el equipo a mejorar el objetivo. Además, deben ser tareas realizables. Posteriormente, el equipo de desarrollo debe discutir cómo va a realizar las tareas, pidiendo ayuda si hace falta al Product Owner para que especifique.

Cada día dentro del sprint el equipo de desarrollo tendrá un **daily scrum**, una reunión de 15 minutos donde decidirá qué va a hacer cada miembro del equipo durante las siguientes 24 horas.

Al acabar el sprint, tiene lugar el **sprint review** y el **sprint retrospective**. La primera es una reunión donde se valora el trabajo realizado, el trabajo restante, se reevalúan prioridades y se comentan posibles dificultades que haya podido haber y en general, cualquier cosa que pueda servir para la mejora del producto.

La segunda reunión es mucho más introspectiva, y el objetivo es la mejora del equipo, cómo hacer que trabaje mejor, problemas interpersonales, etc

4.1.4. Artefactos

Durante un proyecto Scrum se usan diversos artefactos. Siguiendo los valores fundacionales, estos deben ser transparentes, sujetos a la inspección y adaptables a las circunstancias.

El **Product Backlog** es una lista ordenada de tareas que se deben realizar en el proyecto. Es responsabilidad del Product Owner gestionar esta lista. La lista está ordenada según la prioridad de las tareas. La lista es dinámica y nunca está completa. Continuamente se van añadiendo tareas (funcionalidad nueva, errores, pruebas, ...) y es un reflejo de cómo el producto evoluciona. Es muy recomendable que las tareas, además de ser claras, incluyan una definición de “completo” para saber de forma objetiva cuando una tarea ha sido finalizada.

El **Sprint Backlog** es el conjunto de tareas seleccionadas del Product Backlog para realizar durante el sprint más tareas que surgen durante el propio sprint. Todas ellas deben ser mencionadas en los daily scrum. Solo el equipo de desarrollo puede modificar el sprint backlog una vez ha comenzado el sprint y siempre y cuando no modifique el objetivo del sprint. Idealmente, todas las tareas del sprint backlog deberían estar completas al acabar el sprint.

El **incremento** son las tareas finalizadas al acabar el sprint y que incrementan el valor del producto. El incremento debe ser algo inspeccionable y un paso adelante hacia el objetivo. El incremento debe estar en condiciones de uso aunque el Product Owner puede decidir no publicar una nueva versión.

4.2. Metodología de trabajo

Para el proyecto se ha seguido una adaptación de Scrum para entornos académicos (UVagile[20]) considerando que este TFG es un proyecto de trabajo individual.

En primer lugar se fija el tiempo de los sprints a 3 semanas. Tiempo suficiente para avanzar de forma significativa y poder acabar bastantes tareas, sin con ello perder el foco para poder iterar en caso de que haya que hacer cambios o cierta parte se quede atascada. No habrá dailies, aunque existirá una comunicación constante mediante la herramienta Slack con los tutores por si hubiese dudas puntuales.

El product backlog se define en un tablero de Trello, donde se anotan todas las tareas pendientes, más grandes o más pequeñas. Estas tareas son definidas por mí mismo, haciendo el rol de Product Owner.

Al principio del sprint tiene lugar una reunión que combina el sprint review y el sprint planning. Esta reunión se realizaba principalmente usando la herramienta Google Hangouts. Primero se repasaba el sprint backlog del sprint anterior, para ver qué se había movido al incremento. Además, se mencionaban ciertos problemas que habían sido encontrados: problemas en la definición de completado, dificultades a la hora de modelar ciertos aspectos, bugs pertenecientes a librerías que usamos, etc.

Además, se repasa la parte del sprint backlog que no ha sido completada. Se repasa para ver los motivos por los que no había podido ser completada y si tiene sentido seguir con ella en el siguiente sprint o si es mejor eliminarla.

Una vez acabada esta parte, comienza la parte de sprint planning, donde se seleccionan las tareas del product backlog que pasan al sprint backlog para el sprint siguiente. Además, se analizan por encima como se piensan implementar, aunque esto no es concluyente en ningún caso.

Durante la reunión están presentes uno o dos tutores, desempeñando ellos un rol de Scrum

Master (Miguel Ángel Martínez Prieto y María Aránzazu Simón Hurtado).

El resto del sprint, el equipo de desarrollo (yo mismo) va modificando las tareas sobre Trello según se cumplen las condiciones de completado o si se encuentra algún obstáculo.

4.3. Estimación del esfuerzo

En la sección de alcance vimos las cinco historias de usuario propuestas para llevar a cabo los objetivos del proyecto. Posteriormente en la sección de análisis las descompondremos, pero antes de ello, vamos a realizar una estimación del esfuerzo a realizar.

Para ello, vamos a asignar a cada historia de usuario unos puntos de historia. Estos puntos de historia de usuario representan el esfuerzo que a priori creemos que hay que realizar para completar esa historia de usuario. Estas estimaciones mejoran con la experiencia del equipo de desarrollo, por lo que en este proyecto podemos esperar que sean unas estimaciones bastante poco precisas. Estos puntos pueden traducirse en horas de trabajo, con una conversión que solo puede calcularse una vez el equipo de desarrollo haya finalizado ya algunas historias de usuario.

Para asignar puntos de historia existen diversos métodos. Uno de ellos es planning poker, donde el equipo de desarrollo tiene cartas enumeradas. De forma simultánea cada desarrollador elige una carta con los puntos que considera debe llevar esa historia de usuario y la muestra. El objetivo es alcanzar el consenso de todo el equipo, de forma que se debate y se vuelve a realizar la tirada de cartas hasta que una amplia mayoría otorgue los mismos puntos.

Las cartas pueden ir enumeradas de distinta forma, pero en este proyecto se ha optado por la sucesión de Fibonacci. Esta sucesión funciona bien para estimar puntos de historia ya que tiene en consideración la ley de Weber sobre las comparaciones que realizamos los seres humanos[6].

Después de la sesión de planning poker, el resultado fue el siguiente:

- Almacenar datos junto a sus ontologías en el mismo espacio, validándose - 5 puntos
- Desarrollar una ontología que permita definir aplicaciones web sobre datos almacenados en el propio almacenamiento - 8 puntos
- Servir páginas web estáticas definidas mediante esa ontología - 5 puntos
- Servir páginas web que necesiten leer datos del almacenamiento (plantillas) - 13 puntos
- Servir páginas web CRUD con formularios - 21 puntos

4.4. Plan de gestión de riesgos

Para estar prevenidos ante posibles riesgos que pudieran darse en la ejecución de este proyecto, se planifican con antelación ciertos problemas que podrían surgir. Este plan se elabora en cuatro fases:

ID	Descripción	Daño	Probabilidad	RE
R1	Pandemia global	5	0.5	2.5
R2	Enfermedad desarrollador	7	2	14
R3	Bug inesperado en las librerías	4	4	16
R4	Equipos se estropean	9	1	9
R5	Se encuentra fallo grave de diseño tarde	6	3	18
R6	La implementación tarda más de lo previsto	6	5	30
R7	Cambian los requisitos durante la implementación	7	5	35
R8	Historias de usuario poco definidas	3	5	15
R9	Tests poco realistas	4	4	16
R10	Conocimiento de tecnologías insuficiente	2	6	12

Tabla 4.1: Análisis de riesgos

1. Identificación de los riesgos
2. Análisis y priorización de los riesgos
3. Planificación sobre los riesgos
4. Monitorización de los riesgos

Para priorizar los riesgos tendremos en cuenta la siguiente expresión: $RE = d * p$ donde RE es el Risk Exposure, d es el daño potencial que puede causar y p es la probabilidad de que suceda. El daño puede medirse como una cuantía económica, o en nuestro caso, en una escala de 0 a 10. Las probabilidades usan la misma escala.

Una vez realizado este análisis, visible en la tabla 4.1, planificamos qué hacer con cada uno de ellos. Las opciones disponibles son:

- Aceptar que los riesgos puedan ocurrir
- Evitar el riesgo
- Reducir el riesgo
- Transferir el riesgo
- Mitigar el riesgo

La diferencia principal entre evitar/reducir y mitigar un riesgo es que en el primer caso, hemos de realizar una inversión aunque el riesgo finalmente no se materialice, mientras que en el segundo caso, solo deberíamos dedicarle atención una vez sucede. Las soluciones propuestas se encuentran descritas en la tabla 4.2.

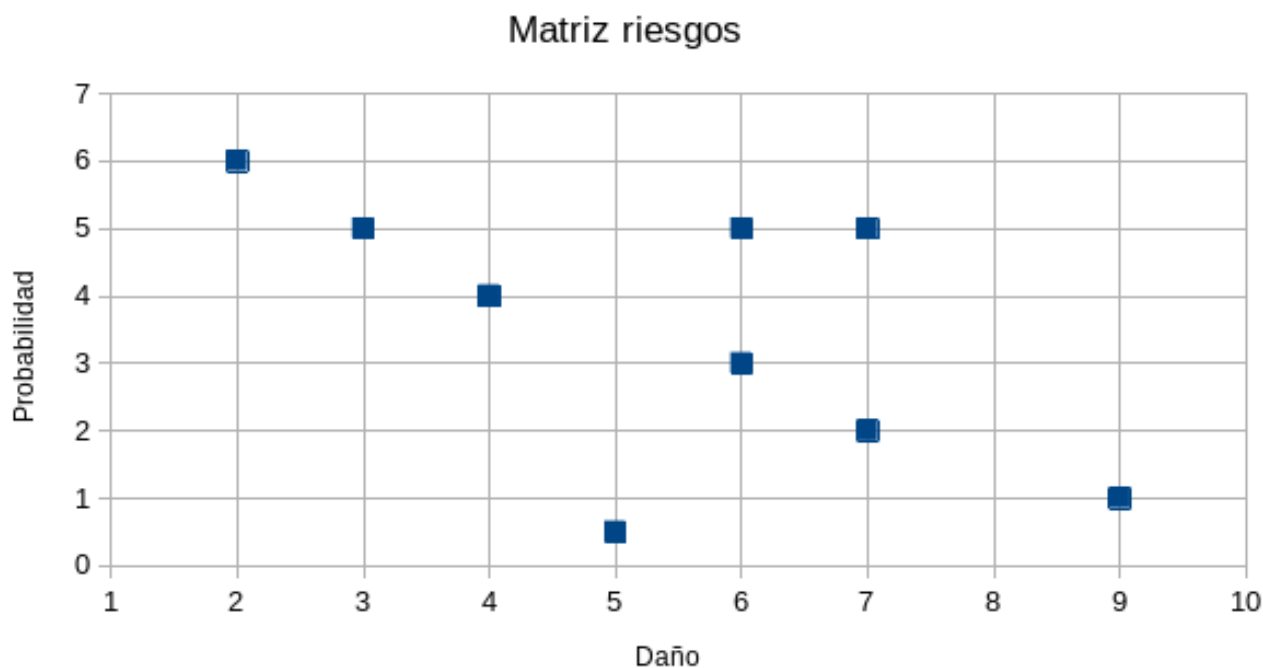


Figura 4.1: Matriz de riesgos

ID	Descripción	Plan
R1	Pandemia global	Aceptar el riesgo
R2	Enfermedad desarrollador	Aceptar el riesgo
R3	Bug inesperado en las librerías	Reducir el riesgo tratando de usar la menor cantidad de librerías posibles
R4	Equipos se estropean	Mitigar el riesgo usando otros PCs de casa y reducir el riesgo manteniendo copias de seguridad
R5	Se encuentra fallo grave de diseño tarde	Reducir y transferir el riesgo haciendo sprints reviews y plannings junto al tutor
R6	La implementación tarda más de lo previsto	Mitigar el riesgo teniendo siempre entregables listos para presentar al final de cada sprint
R7	Cambian los requisitos durante la implementación	Reducir el riesgo introduciendo un sistema de testing flexible y polivalente
R8	Historias de usuario poco definidas	Reducir y transferir el riesgo haciendo sprints reviews y plannings junto al tutor
R9	Tests poco realistas	Reducir el riesgo introduciendo un sistema de testing flexible y polivalente
R10	Conocimiento de tecnologías insuficiente	Evitar el riesgo estudiando más sobre Prolog y RDF.

Tabla 4.2: Plan de riesgos

4.5. Presupuesto

En esta sección vamos a analizar el costo que tendrá realizar el proyecto. Vamos a asumir dos tipos de gastos: gastos de personal y gastos de material.

Los gastos de material van a ser de 0€, ya que vamos a usar servicios en su modalidad gratuita (GitHub, GitHub Actions) y se va a usar para programar el equipo personal del desarrollador.

Los gastos de personal también van a ser de 0€, ya que el desarrollador no va a cobrar por el proyecto. Solo tenemos en cuenta al desarrollador ya que el tutor y debe tener asignados varios proyectos de forma simultánea y su sueldo no depende sustancialmente de la realización de este proyecto.

Podemos estimar el gasto de personal del desarrollador en varios escenarios. Vamos a suponer que solo va a cobrar las horas estimadas que serían 300 horas. Si dividiésemos las 300 horas en jornadas de 8 horas, tendríamos 37,5 jornadas. Redondeamos a 38 jornadas. Según el SMI de 2020, al ser un trabajo por un periodo inferior a 120 días, el precio diario se fija en 44,99€. Así pues las 38 jornadas a 44,99€, daría 1709,62€ de salario bruto.^[38] Esto sería un precio por debajo de mercado ya que en el sector es difícil encontrar trabajos que utilicen el SMI.

Otro escenario más realista sería asumir un sueldo de mercado. En ese caso, cada jornada van a ser 98€ brutos, que multiplicado por el número de jornadas (37,5) serían 3675€. Esta estimación es mucho más realista.

Capítulo 5

Análisis y diseño

En este capítulo realizaremos el análisis del proyecto a realizar. Primero explicaremos en detalle las historias de usuario ya mencionadas en el primer capítulo. Después analizaremos los requisitos que surgen de esto, tanto funcionales como no funcionales.

5.1. Historias de usuario

Las historias de usuario que se han seleccionado para este proyecto son las siguientes:

1. **HU1.** Almacenar datos junto a sus ontologías en el mismo espacio, validándose. El objetivo es tener un sistema de almacenamiento de tripletas, que sirva de base para el resto de historias. Este almacenamiento deberá ser compatible con tripletas, preferiblemente en formato Turtle y deberá ser manipulable desde el exterior también con operaciones de búsqueda, inserción y eliminación de datos.
2. **HU2.** Desarrollar una ontología que permita definir aplicaciones web sobre datos almacenados en el propio almacenamiento. Esta ontología deberá aprovechar otras ontologías existentes, en la medida de la posible. La ontología deberá definirse usando tecnologías interoperables. Se deja la puerta abierta a que se defina una ontología más amplia de la que pueda manejar Lyncex al finalizar el proyecto. El objetivo es poder tener una documentación formal de cara a desarrolladores externos.
3. **HU3.** Servir páginas web estáticas definidas mediante esa ontología. Se entiende por páginas estáticas aquellas cuyo contenido no depende más que de sí mismo.
4. **HU4.** Servir páginas web que necesiten leer datos del almacenamiento (plantillas). Estas páginas son aquellas que implementan un procesamiento extra (plantillas) y, además, pueden leer y escribir datos en el almacenamiento. Pueden acceder a valores de parámetros GET y POST y pueden realizar validaciones.
5. **HU5.** Servir páginas web CRUD con formularios. Los formularios son una abstracción para la manipulación de datos que facilita el trabajo en datos agrupados sobre un sujeto. Esta abstracción deberá permitir realizar formularios automáticamente en HTML, validarlos. Editar y borrar su contenido. Además habrá páginas públicas y páginas privadas protegidas por un usuario/contraseña.

5.2. Requisitos

A continuación analizamos los requisitos que surgen de estas historias de usuario.

5.2.1. Requisitos funcionales

Los requisitos funcionales encontrados son:

HU1

1. **RF1.** El sistema deberá permitir consultar todas las tripletas almacenadas de forma externa mediante una API.
2. **RF2.** El sistema deberá permitir consultar las tripletas almacenadas aplicando filtros de forma externa mediante una API.
3. **RF3.** El sistema deberá ser capaz de admitir tripletas nuevas de forma externa mediante una API.
4. **RF4.** El sistema deberá ser capaz de eliminar tripletas de forma externa mediante una API.
5. **RF5.** El sistema deberá ser capaz de eliminar tripletas aplicando filtros de forma externa mediante una API.
6. **RF6.** El sistema deberá validar que la propiedad `rdfs:domain` definida en las ontologías residentes en el almacenamiento se cumple.
7. **RF7.** El sistema deberá validar que la propiedad `rdfs:range` definida en las ontologías residentes en el almacenamiento se cumple.

HU2

1. **RF8.** Deberá existir una ontología en formato Turtle que permita usar el sistema

HU3

1. **RF9.** El sistema deberá mostrar un error cuando la ruta especificada no concuerde con ninguna página.
2. **RF10.** El sistema deberá mostrar contenido estático de tipo textual.
3. **RF11.** El sistema deberá mostrar contenido estático de tipo binario codificados en Base64.

HU4

1. **RF12.** El sistema deberá diferenciar entre peticiones GET y POST.
2. **RF13.** El sistema deberá disponer de un motor de plantillas con condicionales.
3. **RF14.** El sistema deberá disponer de un motor de plantillas con bucles.
4. **RF15.** El sistema deberá disponer de un motor de plantillas que permita imprimir valores del almacenamiento.
5. **RF16.** El sistema deberá poder leer parámetros GET y exponerlos en las plantillas.
6. **RF17.** El sistema deberá poder leer parámetros POST y exponerlos en las plantillas.
7. **RF18.** El sistema deberá poder ejecutar validaciones de expresiones regulares sobre el contenido de los parámetros.
8. **RF19.** El sistema deberá poder ejecutar validaciones Prolog sobre el contenido de los parámetros.
9. **RF20.** El sistema deberá poder ejecutar código Prolog para resolver las variables de salida de la plantilla

HU5

1. **RF21.** El sistema deberá generar un formulario HTML automáticamente basado en una ontología
2. **RF22.** El sistema deberá poder validar los formularios con expresiones regulares
3. **RF23.** El sistema deberá poder validar los formularios con código Prolog
4. **RF24.** El sistema deberá poder guardar directamente los formularios
5. **RF25.** El sistema deberá permitir visualizar el contenido de los formularios ya guardados
6. **RF26.** El sistema deberá poder editar formularios ya almacenados
7. **RF27.** El sistema deberá poder borrar el contenido de formularios ya almacenados
8. **RF28.** El sistema deberá poder restringir el acceso a una página mediante usuario y contraseña

Los criterios de aceptación reales que se han seguido en el proyecto se encuentran expresados en lenguaje Gherkin. Se encuentran tanto en el repositorio de código como en el Apéndice B.

5.2.2. Requisitos no funcionales

1. **RNF1.** El sistema deberá estar programado en Prolog.
2. **RNF2.** El sistema deberá funcionar sobre Linux.
3. **RNF3.** El sistema deberá almacenar las tripletas en RDF.
4. **RNF4.** El sistema deberá soportar la codificación UTF-8 en todas partes.
5. **RNF5.** La API de comunicación externa será con verbos HTTP.
6. **RNF6.** La API de comunicación externa usará el formato Turtle.
7. **RNF7.** El sistema deberá poder ser instalable por el usuario en menos de 30 minutos.
8. **RNF8.** El sistema debe ser fiable en todo momento.

5.3. Diseño de Lyncex

5.3.1. Arquitectura

En primer lugar debemos tener en cuenta que Lyncex es un sistema preparado para trabajar en un modelo cliente-servidor, donde el cliente es un navegador web. Esto nos permite dividir el framework en varias capas. Sin embargo, la capa de presentación es muy ligera y depende mucho del renderizado de las plantillas que realice el usuario. La capa de negocio es donde se implementa prácticamente toda la aplicación y la capa de datos es donde tiene lugar la persistencia.

Entrando dentro de la capa de negocio tenemos la aplicación del patrón Front Controller. Este patrón, fue diseñado para aplicaciones web donde existe un único punto de entrada. Este realiza tareas comunes y, finalmente, delega la acción concreta a un controlador. Alrededor de los controladores existen helpers que implementan funcionalidad común. Un esquema fundamental de la arquitectura aparece en la figura 5.1.

5.3.2. API

La API de Lyncex se encarga de que el sistema interactúe con el exterior. Un programador que desee crear una aplicación de Lyncex es lo único que debería tocar. Externamente se trata de una API HTTP, que define varias rutas bajo la ruta `_api`. La barra baja se ha elegido para no interferir con un posible sistema de APIs de la aplicación que use Lyncex. Las operaciones que soporta la API son:

- Creación de tripletas en la base de datos (POST, `_api`)
- Borrado de tripletas de la base de datos (DELETE, `_api/delete`) con posibilidad de aplicar filtros
- Lectura de tripletas de la base de datos (GET, `_api/query`) con posibilidad de aplicar filtros

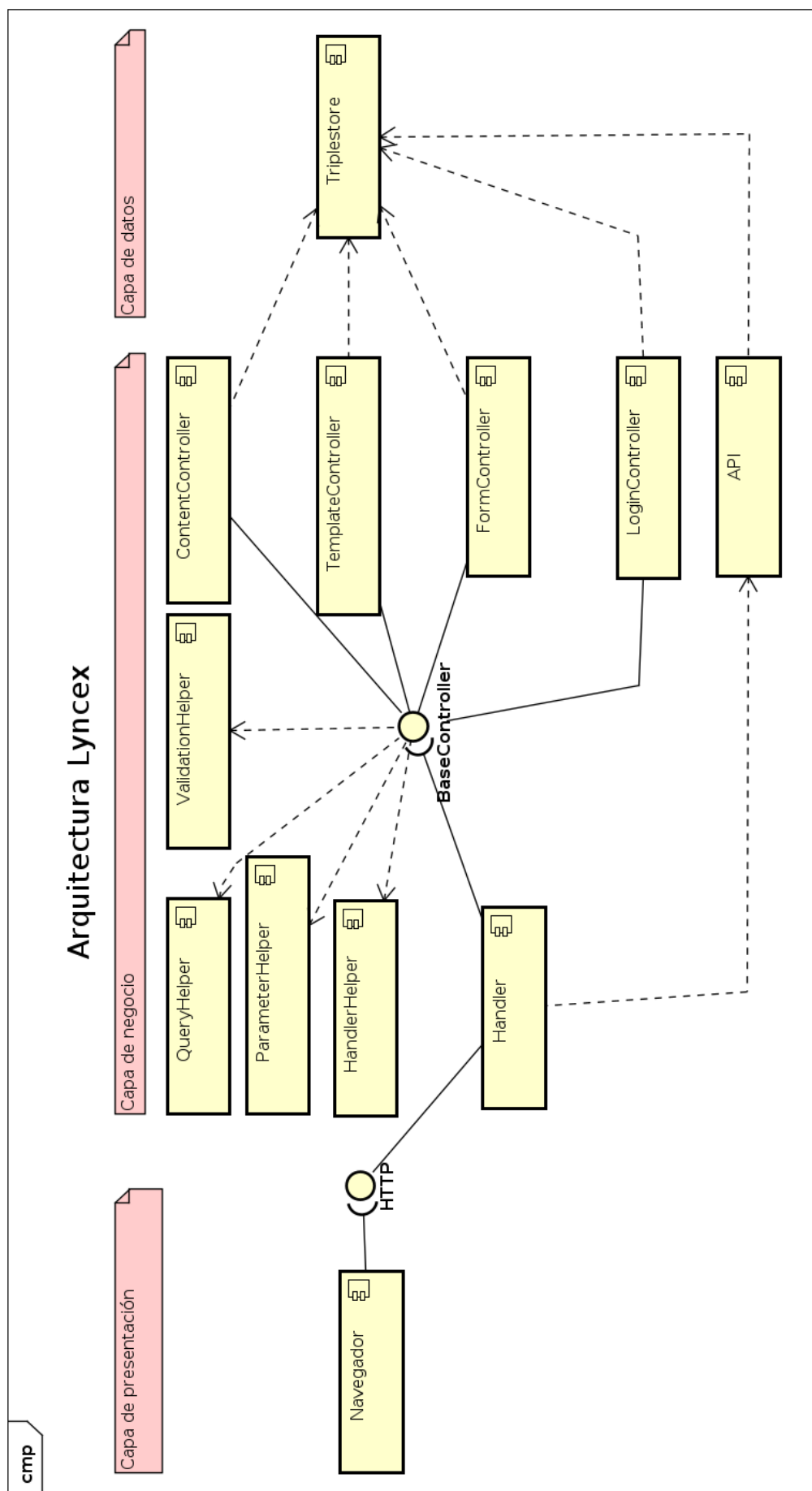


Figura 5.1: Arquitectura de Lyncex

La creación de tripletas toma como entrada un fichero de tipo Turtle, mismo formato que se encuentra a la salida de la lectura de tripletas. De esta forma, se pueden realizar backups rápidos de la aplicación.

Los filtros se implementan mediante parámetros GET.

El componente API, además, dispone de una validación elemental de RDF Schema. Esto se consigue si tanto los datos como las ontologías que los definen coexisten en la aplicación sin incumplir las restricciones de la ontología. Se verifica básicamente el uso correcto de instancias y propiedades mediante `rdfs:domain` y `rdfs:range`. Técnicamente, RDF Schema no es un lenguaje de validación al uso, sino más bien de descripción de datos, pero demasiado abierto como para hacer comprobaciones estrictas. Es por ello que solo se validan estos dos únicos comportamientos.

5.4. Diseño de la ontología

La ontología es el concepto clave de interacción de las aplicaciones con el sistema. A nivel técnico no dejan de ser tripletas, pero se les otorga un valor semántico especial, con consecuencia en Lyncex.

La ontología completa se puede ver en formato Turtle en el Anexo I. En la figura 5.2 podemos ver una representación visual de esta.

El diseño de la ontología se basa en dos clases principales, Prefix y Controller, encapsuladas dentro de una clase maestra, Application. Prefix simplemente añade prefijos RDF para poder usarlos luego en el código del Handler. Controller es la clase de la cual derivan controladores más especializados. La clase base por sí misma no tiene ninguna funcionalidad asociada más allá de la URL, el método de acceso y el control de acceso. Esto la hace imposible de usar de forma directa.

5.4.1. Controladores

Los controladores son los componentes que implementan la funcionalidad principal definida por la ontología. Al llegar una petición, se van probando los diferentes tipos de controladores hasta encontrar uno que sea del tipo al definido en la tripleta: (Controlador, `rdf:type`, `lyncex:TipoControlador`). Inicialmente se han diseñado cuatro controladores.

Antes de todos ellos se comprueba si el usuario puede llamarlos o no. Existen dos usuarios: anónimos, y el administrador.

Esto se comprueba siempre al principio, antes incluso de entrar en el código específico del controlador. Por ello no se ha incluido en los diagramas de secuencia de los controladores. Este flujo se puede ver en la figura 5.3.

ContentController

El más básico de todos, simplemente devuelve lo que tenga definido en la tripleta (Controlador, `lyncex:content`, Content) con el tipo MIME de la tripleta (Controlador, `lyncex:mime`, MimeType).

El nodo `Content` es del tipo `ContentAsText` o `ContentAsBase64`, permitiendo ambos modos de representación del contenido. Ambos tipos forman parte de la ontología de W3C llamada *Representing Content in RDF 1.0*[27]. El tipo `ContentAsText` está pensado para formatos de tipo texto (HTML, CSS, JavaScript) mientras que `ContentAsBase64` está pensado para formatos binarios (imágenes, sonidos, etc). Hay que mencionar, que en ningún caso almacenar ficheros binarios codificados en base 64 es una opción óptima, y esta opción se ofrece más como una conveniencia. Su diagrama de secuencia se presenta en la figura 5.4.

TemplateController

El `TemplateController` se encarga de gestionar las plantillas. Estas plantillas se definen siguiendo la sintaxis *Semblance* de la librería *simple-template*. El `TemplateController` realiza las siguientes operaciones en orden:

1. Procesado de parámetros GET y POST definidos previamente por instancias `lyncex:Parameter`
2. Renderizado de queries (si lo hubiera) especificadas (instancias de `lyncex:Query`)
3. Resolución de las queries (si hubiera)
4. Ejecución de los handlers (si hubiera) (instancias de `lyncex:Handler`)
5. Renderizado final

Su diagrama de secuencia lo podemos observar en la figura 5.5.

FormController

El `FormController` es una abstracción por encima del `TemplateController`. Su funcionamiento depende de si se realiza una petición POST o GET. Ante una petición POST, el controlador procesará los parámetros. Buscará la existencia de un parámetro llamado `_id` en primer lugar, ya que definirá el sujeto sobre el que se van a almacenar tripletas. A continuación recorrerá el resto de parámetros. El nombre de cada parámetro es la URL de la propiedad, y el valor será el valor de la propiedad. Actualmente no se soporta crear dos tripletas de la misma propiedad en el mismo procesado.

Si tenemos una operación GET, el controlador leerá la clase base y todas las propiedades que pueda extraer de la clase. Para ello es importante que la ontología de los datos esté cargada, si no, no será capaz de adivinar qué propiedades admite la clase. Una vez tenga el listado de propiedades, generará un formulario HTML ajustando todos los valores de forma adecuada al formato de aceptación del POST.

Existe un caso particular de la operación GET que sucede cuando existe el parámetro `_id`. Este parámetro debe indicar una IRI de una instancia de la clase. En ese caso el formulario que se genera lleva precargados los datos ya existentes, permitiendo editar el contenido y borrar todas las tripletas con ese sujeto. La operación de edición es simplemente un borrado seguido de los mismos pasos para añadir tripletas desde un formulario vacío.

En el caso de las propiedades que tengan multiplicidad mayor que 1, se debe indicar con la propiedad `lynex:multiple` igual a `true` en la ontología. Esto provocará que la representación gráfica del formulario cambie y, además, modificará ligeramente la lógica de guardado.

Podemos observar los diagramas de secuencia en las figuras 5.6 y 5.7.

LoginController

El `LoginController` es un tipo de controlador muy específico, especialmente diseñado para manejar el flujo de autenticación del usuario administrador en la aplicación. Es en esencia un `FormController` modificado para tratar únicamente el formulario de inicio de sesión. Además, no guarda datos en el almacenamiento RDF, sino que guarda datos en el `SessionStorage`.

Las figuras 5.8 y 5.9 muestran los diagramas de secuencia de este controlador.

5.4.2. Parámetros

El procesamiento de parámetros es llamado por diferentes controladores para adaptar el formato de entrada HTTP GET y POST a un formato común (un diccionario Prolog). Solo saldrán del procesamiento aquellos parámetros indicados de forma explícita, ignorando aquellos que no lo estén. Durante el procesamiento se ejecutan las validaciones si las hubiera. Existen dos tipos de validaciones: `Regex` y `Prolog`. Cualquier parámetro puede tener cero, una o ambas validaciones.

La validación `regex`, simplemente comprueba que el valor del parámetro cumpla con el patrón de una expresión regular estándar. Por debajo se implementa mediante la librería `PCRE`.

La validación `Prolog` es código Prolog que define un término `validation(X)`. Dentro de este código se puede llamar a cualquier código Prolog. La validación será exitosa si el término se puede evaluar a `true`, en caso contrario, se considerará que el parámetro está mal y fallará.

En el caso del `FormController`, las validaciones tienen que ubicarse en otro lugar. Ya que el `FormController` importa implícitamente todos los parámetros de una ontología, no se puede adjuntar una validación de la misma forma. La solución, si usamos un `FormController`, es extender la propia ontología. Así, en un sujeto de tipo `rdf:Property`, podemos añadir `lynex:validation` y `lynex:code_validation`, que funcionan como validación `regex` y validación `Prolog`, respectivamente.

5.4.3. Queries

Las queries son consultas que generan un diccionario sobre las diferentes propiedades de un sujeto. Estas consultas luego sirven para mostrar información dinámica en las plantillas. Para ello se declara una IRI de un sujeto y se recorren sus propiedades, generando un diccionario, siendo las claves las propiedades, y el valor el propio valor u objeto correspondiente. Esta abstracción no es 100 % compatible con el modelo RDF, pero es muy cómoda en determinadas ocasiones. Existen tres limitaciones básicas de las queries:

- Las propiedades toman el nombre del último componente de la IRI. Es decir, si una propiedad es 'http://www.w3.org/2006/vcard/ns#email', en el diccionario aparecerá bajo la propiedad email.
- En caso de varias propiedades que resulten tener el mismo nombre final (dos emails por ejemplo), se tomará el último valor encontrado.
- Los diccionarios no recorren las relaciones generando subdiccionarios.

Para dotar de mayor flexibilidad a las queries, se admite que la IRI del sujeto pueda ser una plantilla y que se pueda componer mediante plantillas de simple-template. Así pues, quedan a disposición de la plantilla las variables de los parámetros definidos con anterioridad.

Podemos observar el flujo de resolución de las queries en la figura 5.10.

5.4.4. Handlers

Los handlers son otro tipo de consultas. En este caso se basan en ejecutar código Prolog que genere un diccionario válido para las plantillas. Al tratarse de código Prolog, disponemos de una gran flexibilidad para realizar casi cualquier cosa. No existen limitaciones a la hora de llamar a términos estándar de Prolog, así como a términos del módulo `rdf11`. Además se incluye acceso a un diccionario de parámetros. Su inconveniente principal es una mayor dificultad de uso y la mayor verbosidad.

Podemos observar el flujo de resolución de los handlers en la figura 5.11.

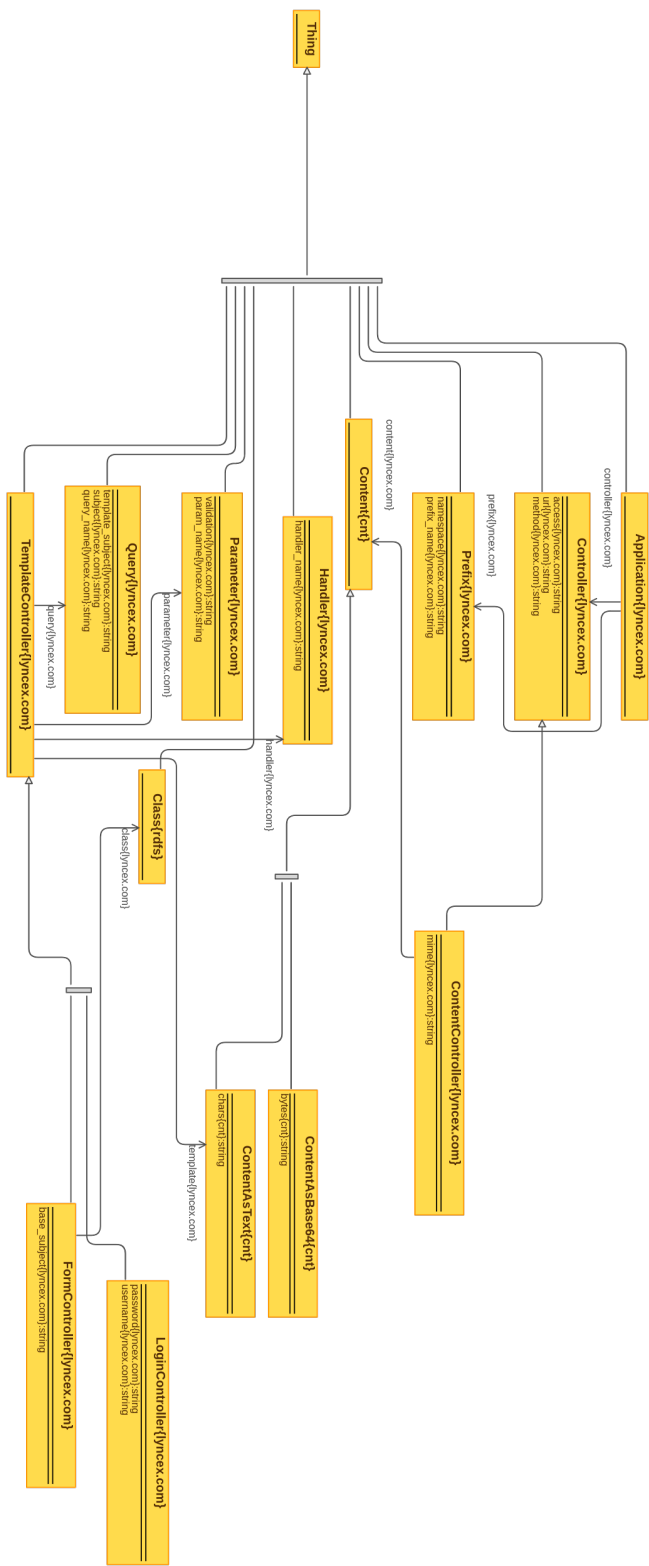


Figura 5.2: Ontología de Lynce

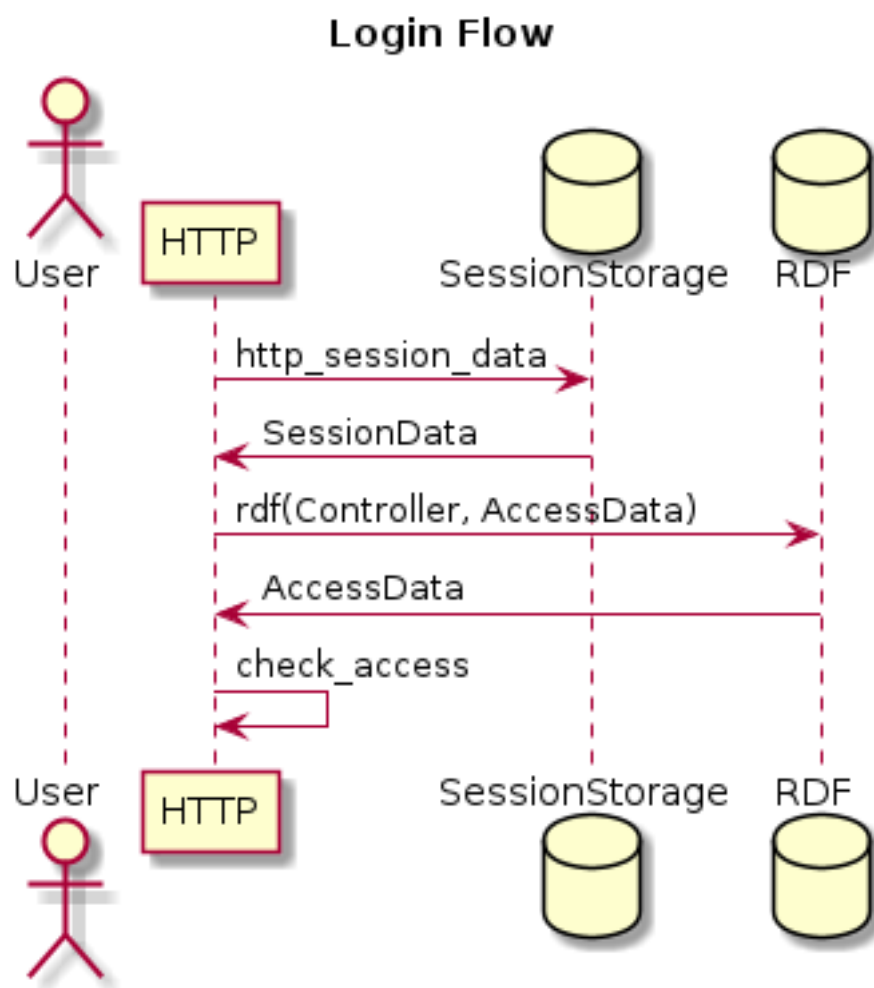


Figura 5.3: Diagrama de secuencia de la comprobación de acceso

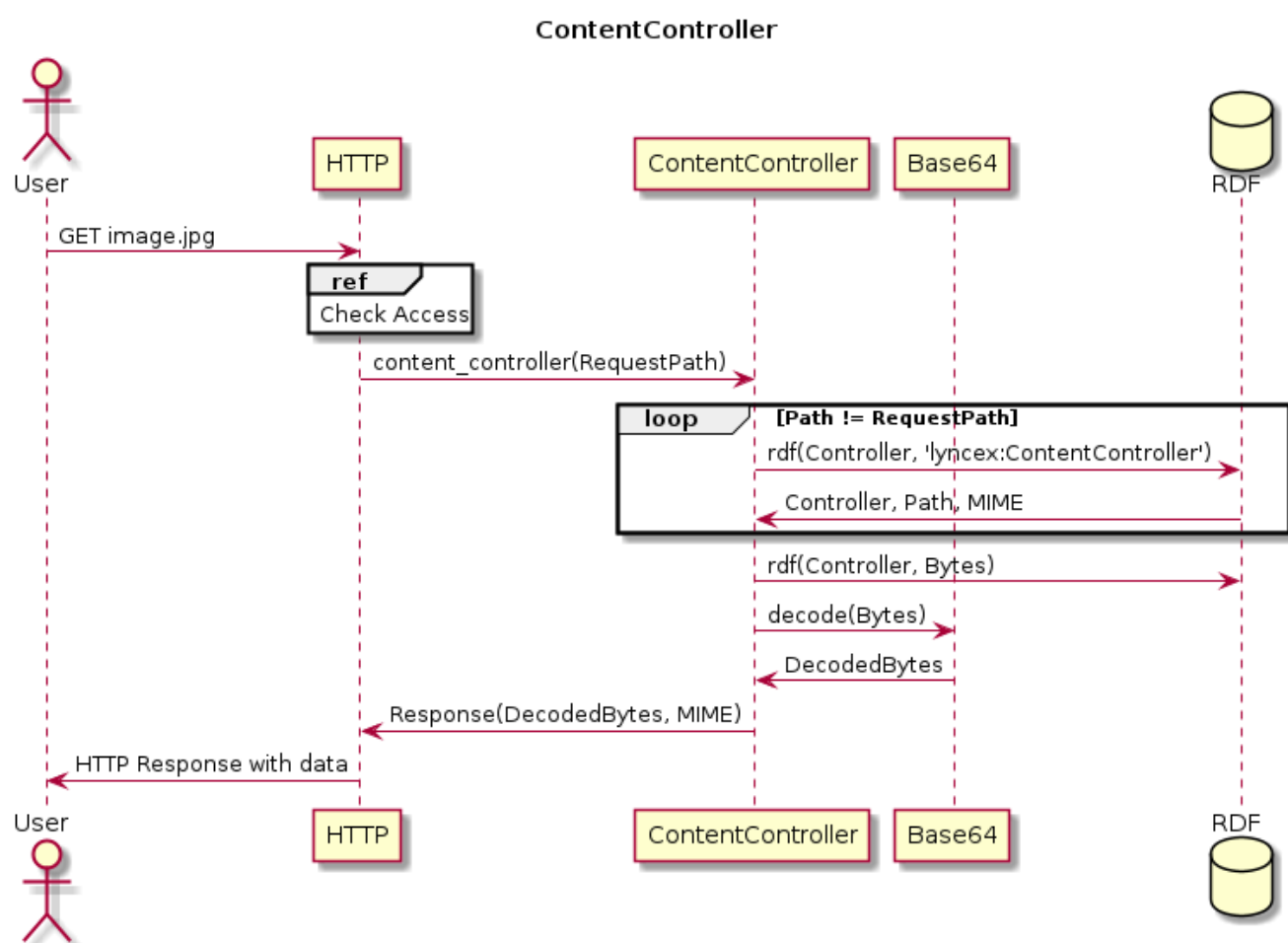


Figura 5.4: Diagrama de secuencia del ContentController

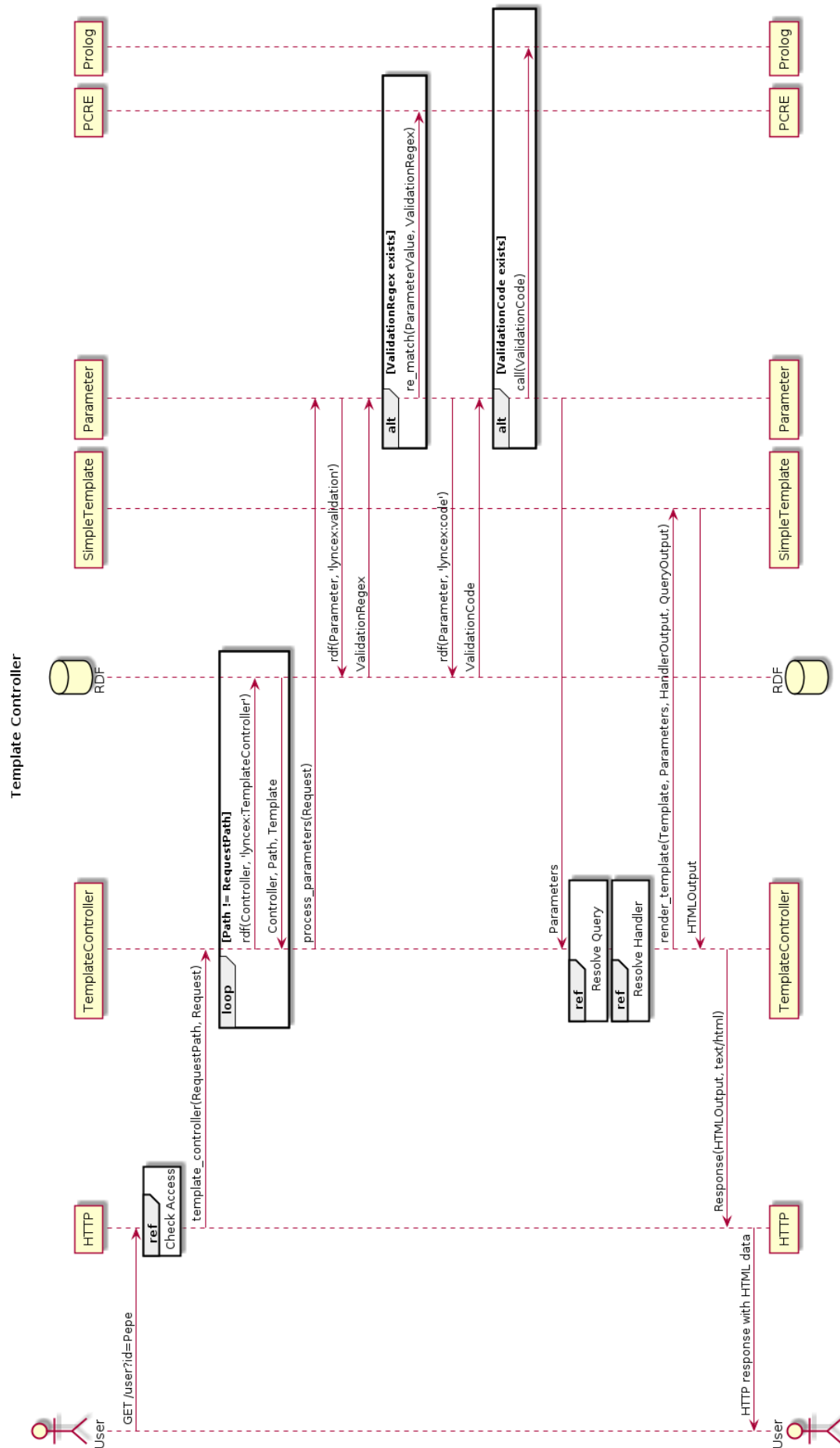


Figura 5.5: Diagrama de secuencia del TemplateController

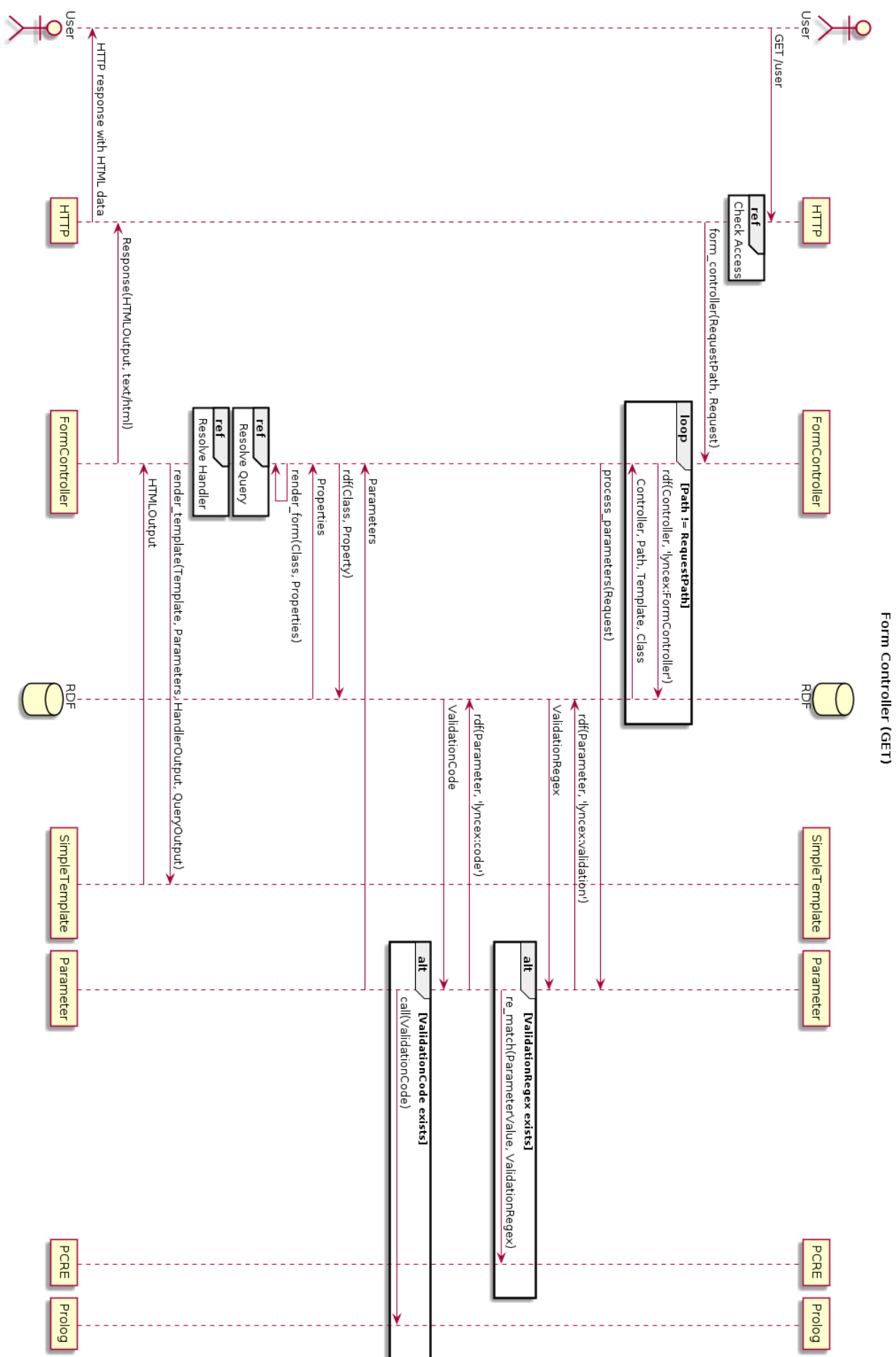


Figura 5.6: Diagrama de secuencia del FormController ante un GET

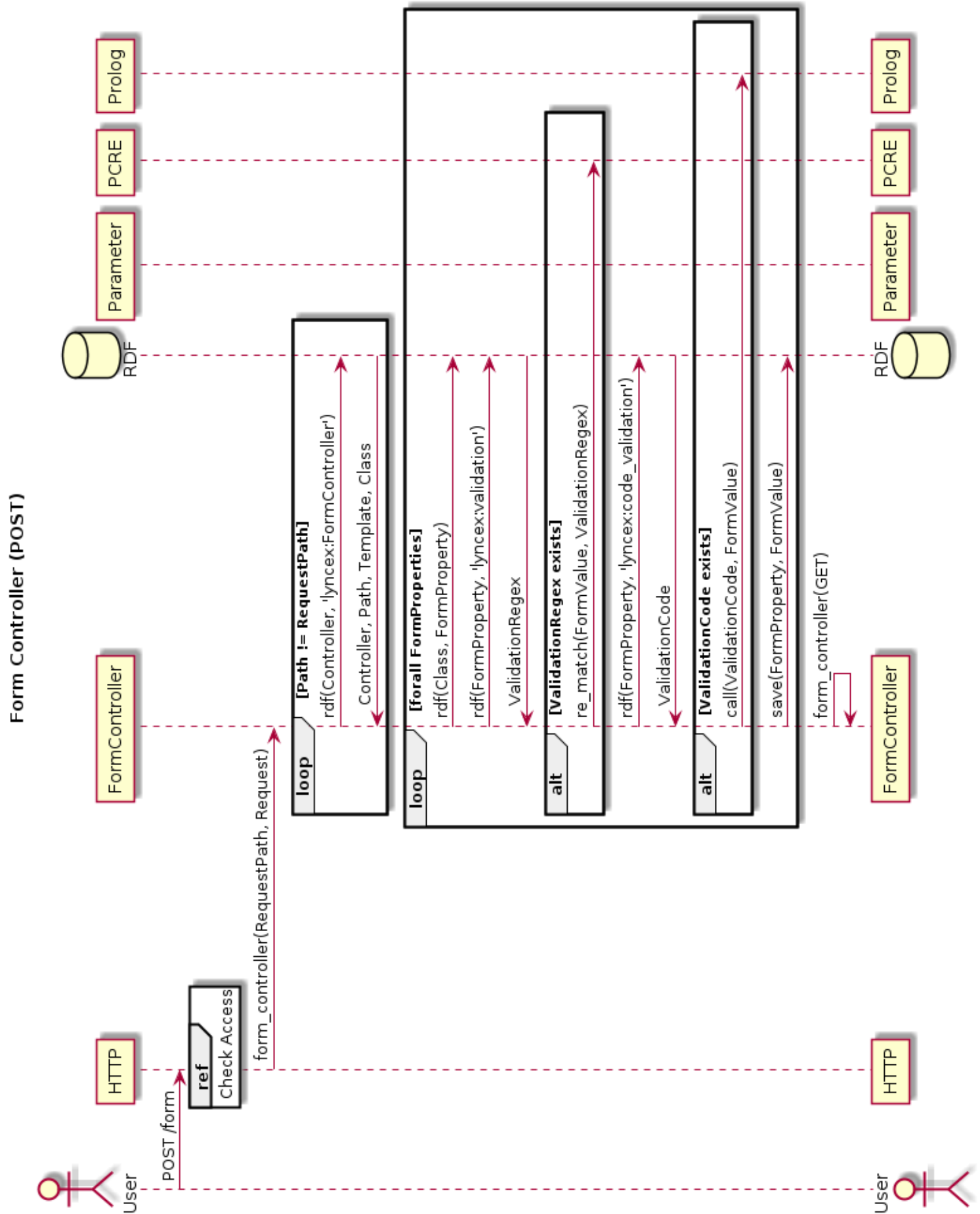


Figura 5.7: Diagrama de secuencia del FormController ante un POST

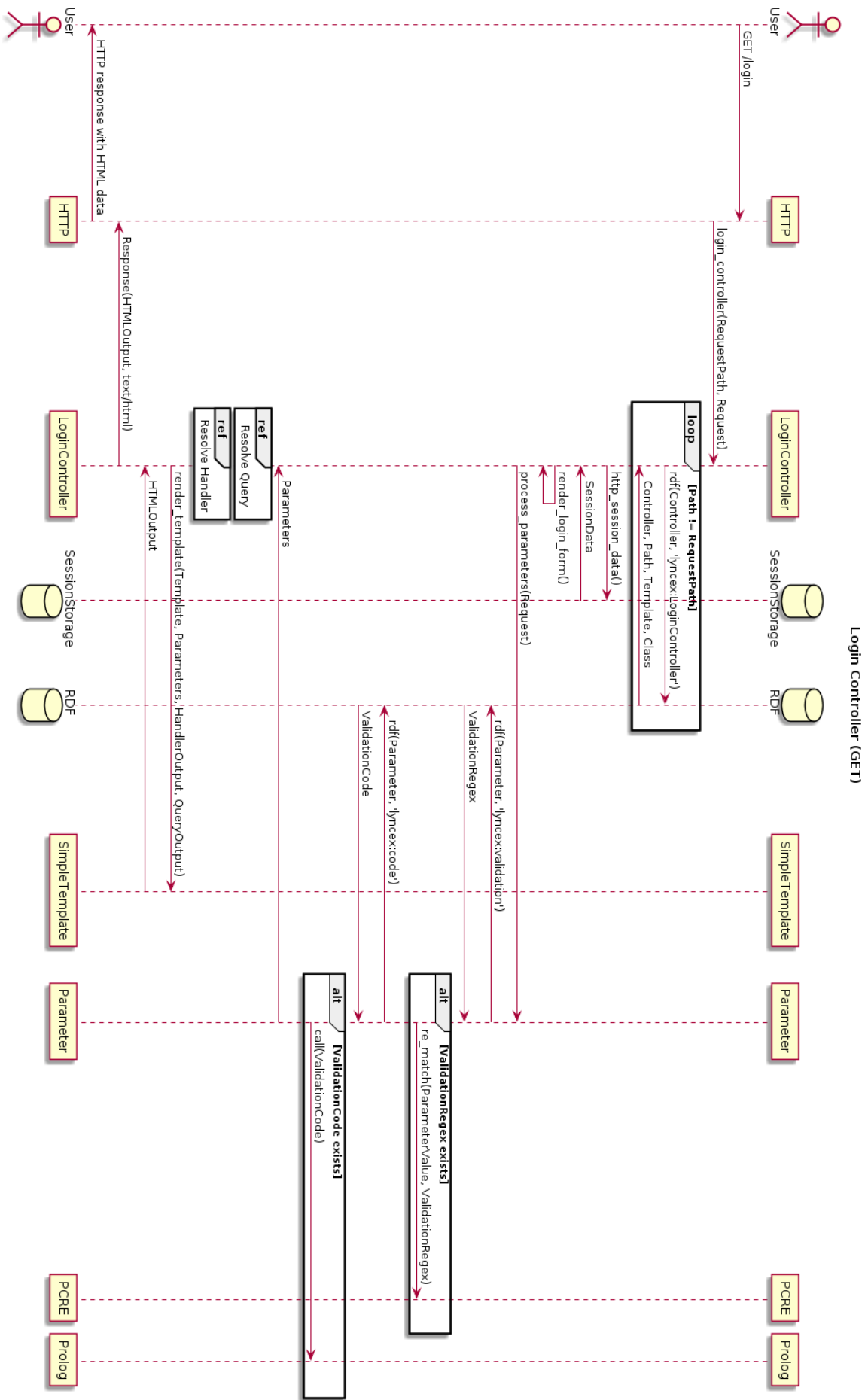


Figura 5.8: Diagrama de secuencia del LoginController ante un GET

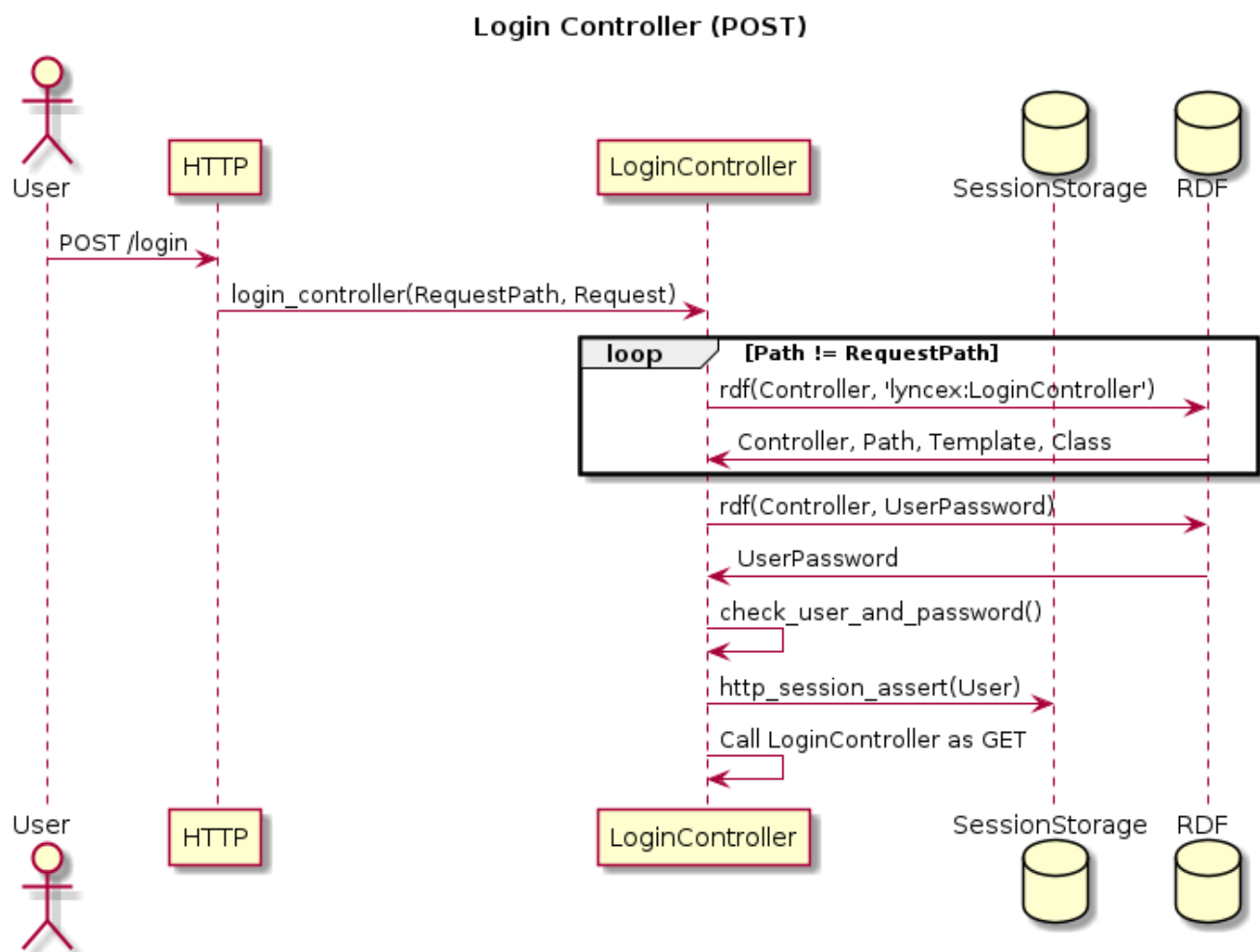


Figura 5.9: Diagrama de secuencia del LoginController ante un POST

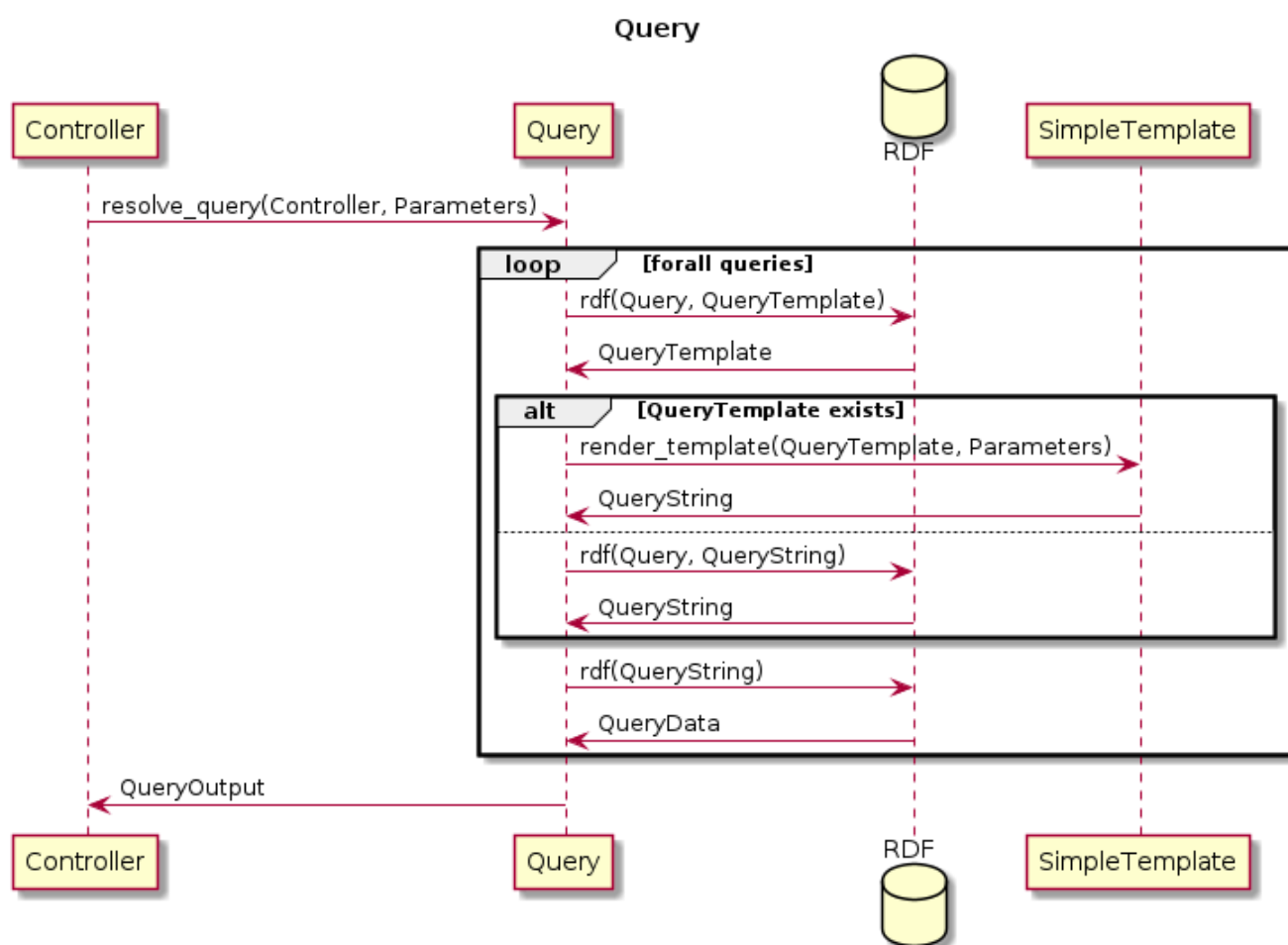


Figura 5.10: Resolución de las queries

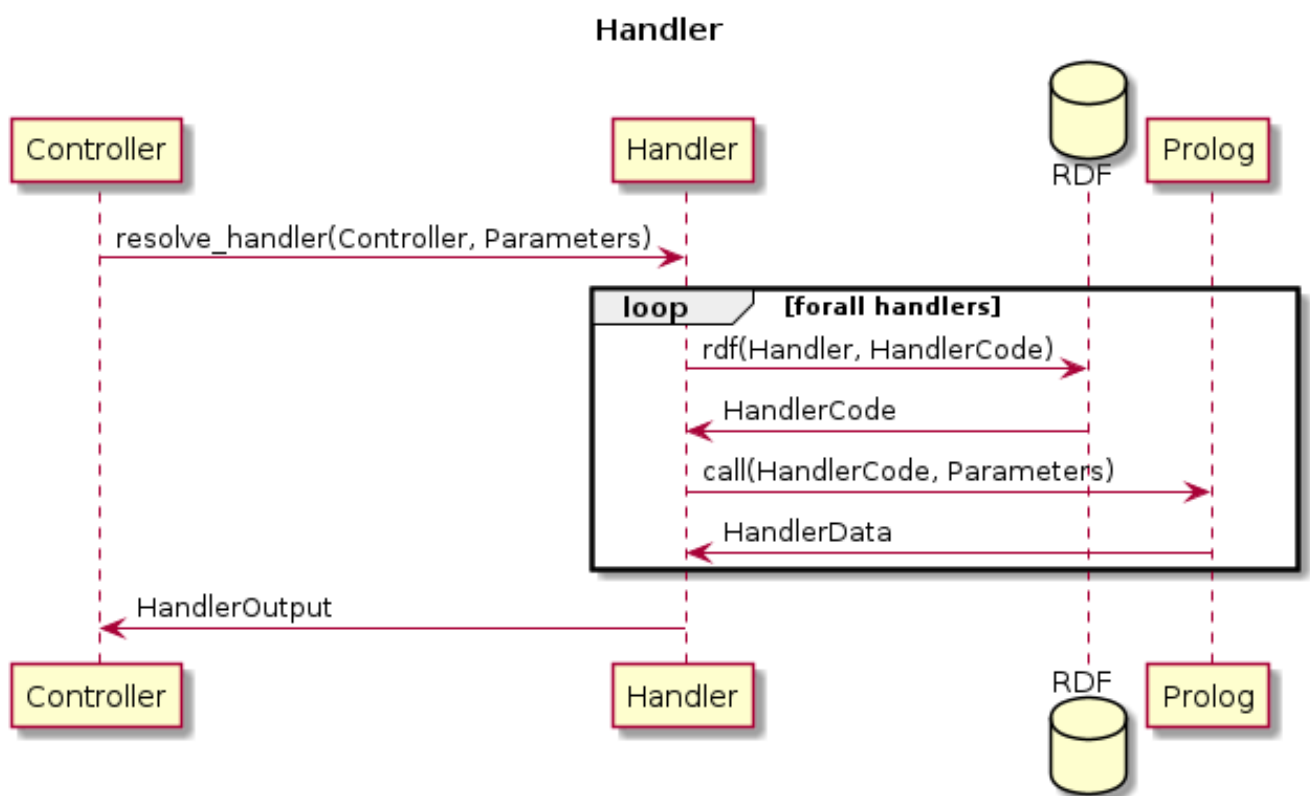


Figura 5.11: Resolución de los handlers

Capítulo 6

Implementación

6.1. Herramientas de desarrollo

6.1.1. SWI Prolog

SWI Prolog[33] es un entorno de programación Prolog, ampliamente usado gracias a su condición de software libre, su portabilidad y su estrategia de baterías incluidas, que hace que disponga de gran cantidad de librerías y módulos por defecto. La comunidad de SWI Prolog es pequeña pero activa y a parte de la gran cantidad de librerías, existe un sistema de packs, que permiten instalar librerías de terceros.

Entre los módulos que tiene SWI Prolog y que es difícil encontrar en otras implementaciones, debemos destacar los módulos de web semántica y RDF y los de HTTP. Sin estos módulos, la realización de Lyncex hubiese sido mucho más compleja y alargada en el tiempo.

6.1.2. Visual Studio Code

Como IDE se ha utilizado Visual Studio Code[21], un editor propiedad de Microsoft, pero multiplataforma. Es un IDE con el que ya había familiaridad previa y no supone ningún esfuerzo usarlo para este proyecto. Visual Studio Code dispone de un sistema de plugins, y entre otros, hay plugins de Prolog y de Turtle. El último fue usado para obtener resaltado de sintaxis, mientras que el primero se evaluó y finalmente se deshechó. El motivo es que el plugin ejecuta Prolog por debajo para detectar errores de sintaxis, pero al tratarse de una aplicación que es un servidor, al ejecutarse toma los puertos y no podemos usarlos en pruebas lanzadas a través de la terminal.

6.1.3. Sistema operativo

La totalidad del proyecto se ha realizado sobre Debian, en su versión Sid. Se trata de un sistema operativo con kernel Linux, de amplio uso en servidores, y también usado, en menor medida, en portátiles y workstations.

6.1.4. Docker

Para que los entornos de prueba y de ejecución sean reproducibles, se ha optado por usar Docker y su utilidad, Docker Compose. Docker es un sistema de contenedores para Linux que proporciona capacidades de aislamiento y reproducibilidad de entornos similares a las máquinas virtuales, sin el overhead que usar estas conlleva. Con Docker podemos generar versiones del software inmutables y ejecutables en cualquier kernel Linux sin necesidad de ninguna dependencia más. Además mejora mucho la seguridad por defecto de los procesos que se ejecutan en su interior. Docker además permite ejecutar contenedores en MacOS y Windows mediante una virtualización transparente de Linux. Existen otros sistemas parecidos, como LXD, pero Docker tiene varias ventajas: poder describir entornos como código versionable (Dockerfile) y poder orquestar de forma sencilla entornos pequeños y medianos (docker-compose).

El fichero Dockerfile (y Dockerfile.test para los tests E2E) es una definición de construcción de imágenes. Parte de una imagen base (FROM) y ejecuta comandos para construir una imagen derivada de esta (RUN, COPY) y realiza configuraciones (WORKDIR, CMD, USER). Para ejecutar el programa, debemos construir un contenedor que parta de esa imagen inmutable. El contenedor puede realizar cambios en su sistema de archivos aislado, pero salvo que lo configuremos de forma explícita, estos cambios se perderán al borrar el contenedor. Los contenedores también se pueden parar y reanudar. Estas imágenes generadas quedan almacenadas por defecto en el repositorio local, pero se pueden subir a cualquier repositorio de Docker. Los más populares son Docker Hub, Quay.io y Azure Container Registry.

El fichero docker-compose.yml es una definición de los contenedores que tienen que lanzarse, con sus puertos, volúmenes, etc. Podemos basarnos en imágenes locales y también podemos sobrescribir ciertas configuraciones (como el comando). Además, se pueden definir dependencias entre contenedores a la hora de arrancar.

6.1.5. Git y GitHub

Para versionar el código fuente se ha usado el sistema Git, usando como almacenamiento GitHub. Principalmente se ha usado a través de la línea de comandos. No se han aprovechado muchas de sus características de ramas y merges, ya que al ser un proyecto personal, solo se iba modificando una parte del programa a la vez. Adicionalmente se ha hecho uso de la integración continua gratuita ofrecida por GitHub llamada GitHub Actions. De este modo se ha definido una acción que se ejecuta al recibir nuevo código en el repositorio. Esta acción construía las imágenes de Docker necesarias y pasaba los tests de Behave y de PlUnit.

6.1.6. Behave

Para asegurarnos que las historias de usuario se implementaban correctamente se ha decidido usar Behave. Se trata de un framework para definir tests en lenguaje Gherkin, un lenguaje muy similar al lenguaje natural. Estos tests se componen de pasos, los cuáles son implementados en Python. Como las historias de usuario nos hablan de cómo debe reaccionar el sistema con el exterior, no hay problema en implementar los tests de este tipo en otro lenguaje.

Más detalles sobre Behave en el capítulo de Validación y pruebas.

6.2. Detalles de implementación

En este apartado vamos a mencionar detalles a nivel de implementación surgidos no triviales teniendo en cuenta los apartados anteriores.

6.2.1. Estructura física del proyecto

El código se encuentra disponible en GitHub (<https://github.com/aarroyoc/lyncex>). Se encuentra dividido en varias carpetas. La carpeta `.github` incluye el pipeline de CI (Continuous Integration). La carpeta `apps` contiene la aplicación *BiblioCyL* de ejemplo. La carpeta `features` contiene las especificaciones de test E2E, así como las implementaciones de los tests. La carpeta `lyncex` contiene el código Prolog de la aplicación. La carpeta `tfg` incluye esta misma memoria junto con diagramas. En la carpeta raíz existen además ficheros de Licencia y de Docker.

Dentro de la carpeta `lyncex` existen numerosos ficheros. Los tests unitarios de los términos se definen en el mismo fichero donde está la implementación.

- `api.pl` - La API HTTP para interactuar con el triplestore
- `errorpage.pl` - Páginas de error
- `handler.pl` - Resolución de los handlers
- `main.pl` - Punto de entrada HTTP, control de acceso
- `parameters.pl` - Procesado de parámetros
- `prefix.pl` - Definiciones de prefijos RDF
- `query.pl` - Resolución de las queries
- `setup.pl` - Procesado de prefijos y funciones de Semblance
- `start.pl` - Punto de entrada del proceso
- `validation.pl` - Validación de ontologías RDF Schema
- `controllers/content.pl` - ContentController
- `controllers/form.pl` - FormController
- `controllers/login.pl` - LoginController
- `controllers/template.pl` - TemplateController

La separación entre `start.pl` y `main.pl` tiene el objetivo de no cargar los tests unitarios en memoria cuando no se van a pasar los tests. `start.pl` desactiva los tests, mientras que si se llama directamente a `main.pl` para arrancar el servidor, se cargan los tests. Su relación con los componentes se describe en la figura 6.1.

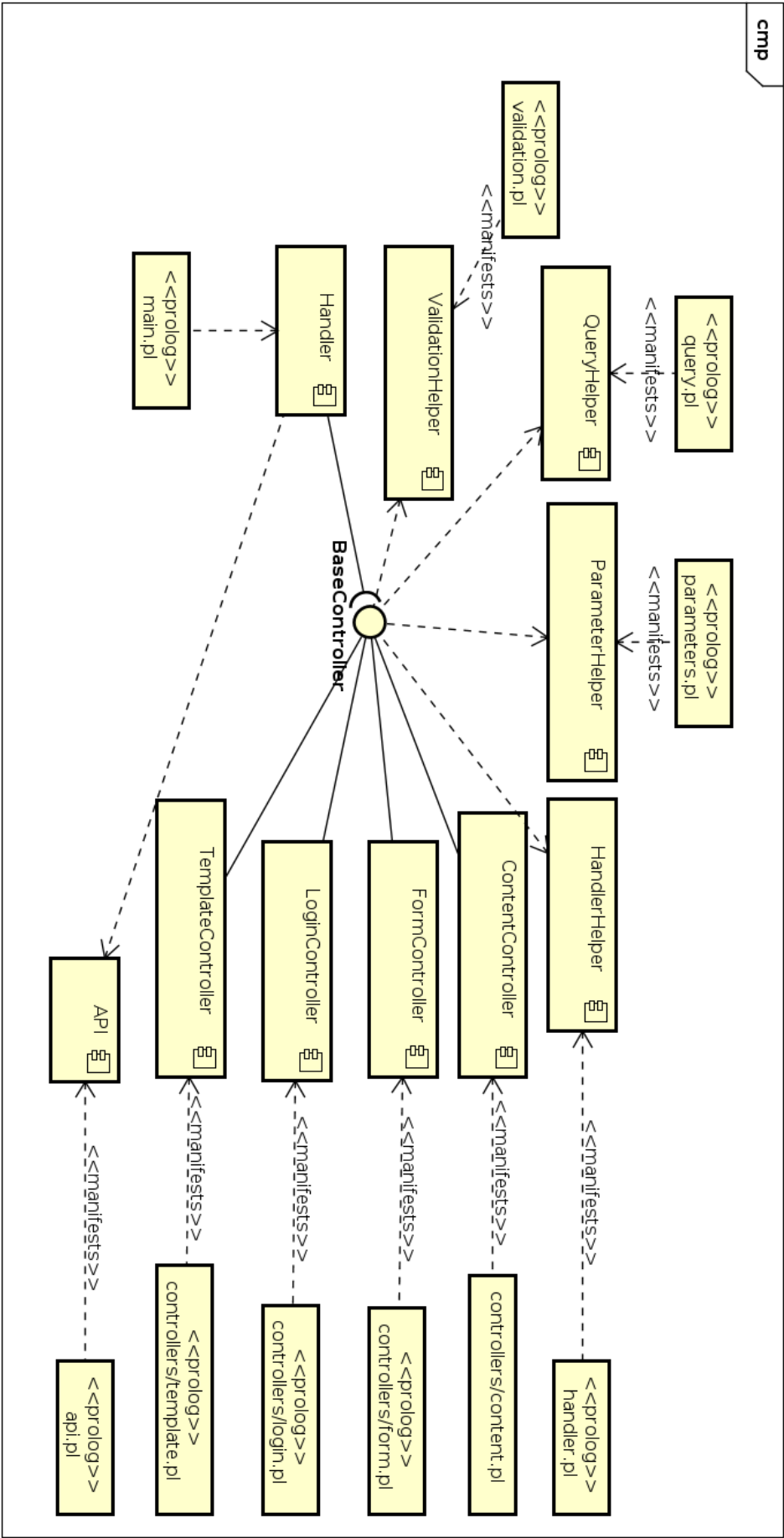


Figura 6.1: Relación ficheros-componentes

6.2.2. Bugs localizados

Durante la codificación, se trató de usar el término `rdf_save_turtle` con la opción `expand` para filtrar las tripletas que se debían devolver al usuario. Aquí se descubrió un fallo reconocido por los autores de SWI-Prolog respecto a la compatibilidad de diferentes representaciones de RDF que tiene la librería internamente. Afortunadamente, fue solventado a partir de la versión 8.1.25 de SWI-Prolog[31].

6.2.3. Dynamic

Existen en el código varios puntos que usan términos “dynamic”. Esto es, modifican el programa en tiempo de ejecución. Se ha recurrido a esta solución por ser la más simple para cargar código Prolog nuevo y para tener un estado en la API en la gestión de los blank nodes de RDF. Usar `dynamic` está altamente desaconsejado, en primer lugar porque no es thread-safe (el resto de la aplicación lo es) y en segundo lugar es subóptimo a nivel de rendimiento.

Capítulo 7

Validación y pruebas

Para comprobar la validez de la implementación se realizan pruebas de diversos tipos.

7.1. Pruebas manuales

El primer tipo de pruebas que se ha realizado son aquellas con intervención manual, realizadas por el programador. En estas principalmente se evalúa que el concepto sigue funcionando. Estas pruebas tienen poco valor, ya que son difíciles de replicar en idénticas condiciones.

No obstante, hay una prueba manual completa en forma de la aplicación de ejemplo, que detallaremos más en profundidad en el capítulo correspondiente.

7.2. Tests unitarios

Entendemos por tests unitarios aquellos que prueban de forma aislada una parte del código. En el proyecto se han implementado tests unitarios usando el framework `PlUnit`. Los tests unitarios de `PlUnit` tienen la siguiente estructura. En este ejemplo se prueba el predicado `process_parameters`, con datos de prueba que simulan una petición real. Se comprueba que el predicado devuelve correctamente un diccionario con los datos de parámetros de entrada y las tripletas especificadas.

```
:- begin_tests(parameters).  
  
test(process_parameters_empty) :-  
    Controller = 'http://example.com/Controller',  
    Parameter = 'http://example.com/ControllerParameter',  
    ParamName = "id",  
    rdf_assert(Controller, lyncex:parameter, Parameter),  
    rdf_assert(Parameter, lyncex:param_name, ParamName),  
    FormData = [id="42"];
```

```
process_parameters(FormData, Controller, OutDict),  
OutDict = _{id:"42"}.  
  
:- end_tests(parameters).
```

Se definen zonas de test delimitadas por `begin_tests` y `end_tests`. En su interior definimos reglas test, que deben ser cumplirse. En este ejemplo, queríamos probar la regla `map_bnode` y si todo va bien, todos los términos deben ser verdaderos. La ejecución de los tests se detalla en el manual de instalación.

Para ejecutar los tests unitarios, el método recomendado es usar Docker Compose y ejecutar:

```
docker-compose run test-unit
```

7.3. Test E2E

Los tests E2E prueban la validez del sistema en su conjunto (end-to-end, punto a punto). Esto se consigue levantando un servidor de Lyncex e insertando datos de aplicaciones de prueba. Posteriormente, vamos probando que todas las características funcionan tal y como se especificó en los tests. Estos tests se encuentran definidos en la carpeta `features`.

Estos tests son muy importantes ya que durante el desarrollo se definía cada tarea nueva como un test E2E. Si el código pasa el test creado con anterioridad, significa que la tarea está completada y pasa a formar parte del incremento.

Los test E2E se implementan con Behave, que es una herramienta específica para realizar Behaviour Driven Development (BDD). La idea es definir, en lenguaje natural, las condiciones de aceptación de una característica. Para ello usa las palabras `Given/When/Then` (y los conectores `And/But`), más conocido como lenguaje Gherkin. Las principales ventajas de este modelo es que podemos trasladar todas las condiciones de aceptación a código y que los tests los pueden escribir personas con escaso conocimiento técnico. El corolario de la primera consecuencia es que siempre vamos a estar atentos ante regresiones en el código.

Las frases con `Given` son precondiciones, acciones que se realizan para dejar el sistema en un estado listo para ejecutar la prueba. Las frases con `When` (idealmente solo una frase), representan el sujeto a prueba, el estímulo que tenemos que hacer para desencadenar aquello que queremos probar. Finalmente, con `Then` deberíamos comprobar que el test arroja los resultados esperados. Deben ser principalmente aserciones.

Behave permite implementar los pasos en Python, en un fichero paralelo (carpeta `steps`).

Veamos un ejemplo de un test E2E. En este ejemplo se vacía el contenido de Lyncex y se carga el fichero `test3.ttl` como preparación. Se prueba que la visita a la URL `/person3?id=42` devuelve un código 200 y una respuesta HTML válida tanto en declaración (`response type`) como en contenido.

```
Scenario: Parameters (GET, valid)
    Given I have an empty Lyncex instance
    And I do a POST request with 'features/test3.ttl' data
    When I visit '/person3?id=42'
    Then I get a 200 status code
    And I get a '<b>ID: </b>42' response
    And I get a 'text/html' response type
```

Los steps son parecidos a esto:

```
from behave import step
import requests
import hashlib

@step("I visit '{url}'")
def step\_visit\_url(context, url):
    context.request = requests.get(f"http://lyncex:11011{url}")

@step("I get the photo '{photo}'")
def step\_get\_photo(context, photo):
    m = hashlib.sha1()
    m.update(context.request.content)
    n = hashlib.sha1()
    with open(photo, "rb") as f:
        n.update(f.read())
    assert m.digest() == n.digest()
```

los steps admiten un templatizado inverso (sirven como patrón para sacar datos) y además siempre comparten una variable llamada context, que podemos usar para conservar estado entre steps.

Para ejecutar los tests E2E, el método recomendado es usar Docker Compose y ejecutar:

```
docker-compose run test-e2e
```

7.4. GitHub Actions

GitHub Actions es un servicio de integración continua/despliegue continuo (CI/CD por sus siglas). Se encuentra integrado dentro del servicio web GitHub y nos permite definir tareas que se ejecutan ante ciertas acciones de entrada. Uno de los usos más habituales es el de comprobar, cuando llegan commits nuevos al repositorio Git, que los tests siguen pasando.

El proyecto cuenta con un fichero que activa GitHub Actions. Este fichero simplemente ejecuta los dos tipos de tests automatizados y si alguno de ellos finaliza incorrectamente, marca el commit en rojo y envía un correo para avisar de que hay código que no pasa los tests en el repositorio.

```
name: Test
on: [push]
jobs:
  test-e2e:
    runs-on: ubuntu-18.04
    steps:
      - name: Checkout
        uses: actions/checkout@v1
      - name: Build Docker images
        run: docker-compose build
      - name: Execute E2E test
        run: docker-compose run test-e2e
  test-unit:
    runs-on: ubuntu-18.04
    steps:
      - name: Checkout
        uses: actions/checkout@v1
      - name: Build Docker images
        run: docker-compose build
      - name: Execute unit tests
        run: docker-compose run test-unit
```

7.5. Conclusiones

El desarrollo de test ha sido fundamental para llevar a buen puerto la aplicación. Se ha comprobado que ningún tipo de test sirve para todo. Un uso de varias técnicas es la única forma eficaz de detectar errores y regresiones.

En particular, en el desarrollo de Lyncex han sido de gran importancia los test E2E, ya que definían tareas con condiciones de aceptación de forma precisa. Al programarse estos tests antes que el propio código, servían para dar una especificación detallada de ciertas cosas que se habían pasado por alto en el proceso de diseño: ¿qué debe responder la API en caso de error? ¿exactamente como tienen que ser los formularios autogenerados?

Además, han sido de gran importancia para detectar regresiones. Cualquier cambio en un componente, podía influir en otros componentes dada la naturaleza de Prolog. En Prolog, si un término falla, se va a buscar otro término que pueda cumplir las condiciones, pero a veces, puede desembocar en código que no queríamos que se hubiese llamado. Estos caminos se pueden generar por accidente, en componentes totalmente separados. Este tipo de test ha resultado especialmente útil para comprobar que las rutas de ejecución que existen son siempre las deseadas.

Un miedo que existía al empezar a realizar los tests, es que estos no fuesen lo suficientemente robustos, es decir, variase su resultado con demasiada facilidad. Los tests unitarios no han tenido este problema, pero los tests E2E, han fallado a veces por variaciones menores. En múltiples ocasiones se han tenido que crear steps especiales en Behave para controlar casos especiales que la sintaxis Given/When/Then no manejaba bien, como los saltos de línea en el propio test.

El tiempo dedicado a los tests también ha sido mayor del que esperábamos. Si bien el tiempo dedicado al test está asociado a su implementación, muchas veces nos hemos encontrado con problemas para replicar exactamente el comportamiento que queríamos desde Behave/Python. A veces, se ha tenido que refactorizar parte de los tests ya hechos por haber encontrado algún fallo a posteriori.

El hecho de mantener un sistema de integración continua, no ha sido excesivamente complicado y ha ayudado a encontrar alguna regresión cuando no se habían ejecutado los tests en local. Sin embargo, los tests en local apenas llevan tiempo de ejecución, y en ese sentido, un sistema de CI tampoco supone un ahorro de tiempo. Si los tests durasen más tiempo (media hora), sí hubiese sido un componente esencial.

En general, podemos decir que los tests han servido para mantener un nivel de calidad constante en todo el software mientras iba creciendo y ha sido un tiempo bien invertido.

Capítulo 8

Aplicación de ejemplo

Con el fin de demostrar que Lyncex es un framework útil, se ha decidido escribir una aplicación de ejemplo. Esta aplicación será una prueba manual de que el sistema funciona y servirá para validar conceptos.

8.1. Objetivo y alcance

La aplicación tendrá como objetivo poder editar el dataset de bibliotecas de Castilla y León. Este dataset, ya está expresado en formato RDF en el portal Datos Abiertos de la Junta de Castilla y León, usando dos ontologías comunes: vCard y FOAF.

La duración del proyecto no será mayor de una semana. No se necesita una interfaz de usuario refinada. El proyecto debe tratar de usar, de forma idiomática, el framework Lyncex.

Definimos las siguientes historias de usuario:

1. Visualizar datos de las bibliotecas. Mostrar tanto en listado como en individual los datos de las bibliotecas.
2. Editar datos de las bibliotecas. Crear, eliminar y editar datos de las bibliotecas. Restringir el acceso de edición a la cuenta del administrador.
3. Mostrar bibliotecas en un mapa. Poner cada biblioteca en un mapa interactivo.

8.2. Análisis y Diseño

Los requisitos que podemos obtener de las historias de usuario son los siguientes:

Requisitos funcionales

1. El sistema deberá permitir visualizar los datos de una biblioteca individualmente

2. El sistema deberá permitir visualizar un listado de las bibliotecas
3. El sistema deberá permitir crear nuevas bibliotecas
4. El sistema deberá permitir eliminar bibliotecas
5. El sistema deberá permitir editar datos de las bibliotecas
6. El sistema deberá bloquear el acceso a las secciones de edición, dejando acceder al administrador

Requisitos no funcionales

1. El sistema deberá ejecutarse como una aplicación dentro de Lyncex

8.3. Arquitectura

La arquitectura es básicamente un recubrimiento por encima de la de Lyncex. Se trata de una aplicación web con tres capas: presentación, negocio y datos. En las tres capas se delega parte del trabajo a Lyncex.

Los datos ya tienen definida dos ontologías que especifican cómo son los datos. En este caso son FOAF[4] y vCard[15]. La primera sirve para modelar interacciones sociales (personas y agentes) y la segunda modela una agenda de contactos. No obstante, vamos a ignorar las relaciones con la ontología FOAF en nuestra aplicación, ya que son secundarias.

8.4. Componentes

La aplicación se compone de dos componentes: un script de transformación de la entrada y por otro lado, la propia aplicación, que consideramos suficientemente pequeña como para ser un componente por sí mismo.

Aun así, podemos dividir la aplicación en diferentes páginas:

- HomePage. La página de inicio
- BookPage. Vista individual de una biblioteca.
- BookListPage. Listado de todas las bibliotecas.
- LoginPage. Permite acceder al sistema como usuario administrador.
- BookForm. Formulario protegido para añadir/editar/borrar bibliotecas.
- GeoForm. Formulario protegido para añadir/editar/borrar puntos geográficos.

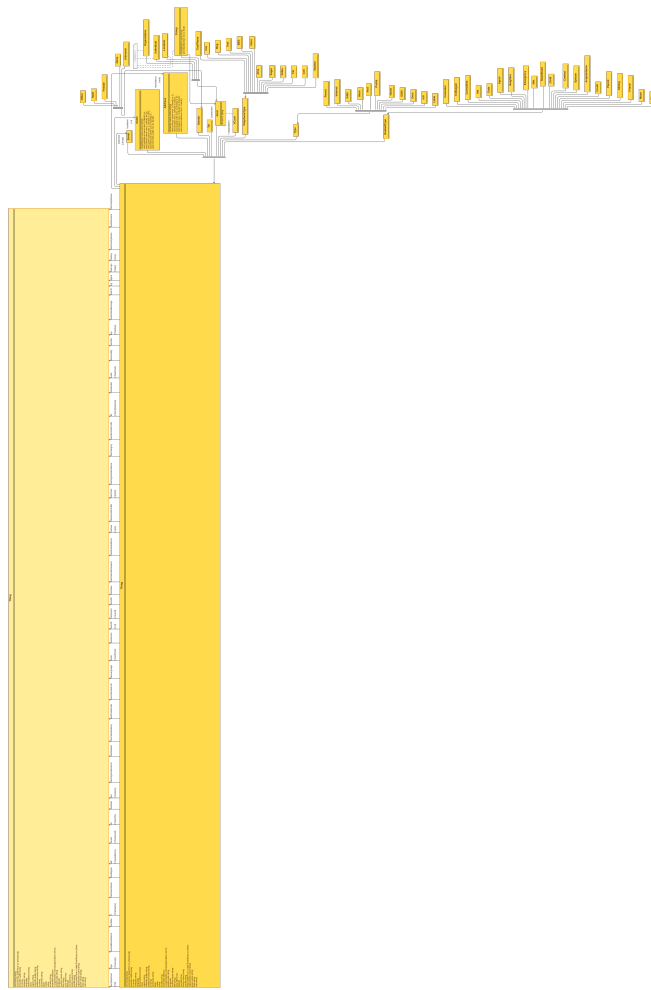


Figura 8.1: Diagrama de la ontología vCard

http://owlged.lumii.lv/online_visualization/2pt2

El script surge como consecuencia del primer requisito no funcional. Es necesario realizar una transformación de los datos a Turtle si no lo estuviesen. Los datos en la página oficial de datos abiertos se encuentran en RDF, pero en su sintaxis XML. Es por ello necesario realizar un pequeño script que realice la transformación inicial.

8.5. Implementación

Para la implementación se han usado las mismas herramientas que para el desarrollo de Lyncex, es decir, Visual Studio Code, Git, y, evidentemente, RDF. El script se realizó con Python usando la librería `rdflib`.

La aplicación intenta usar todas las características presentes en Lyncex.

Inicialmente define un prefijo para `vCard`. Posteriormente va definiendo las páginas como controladores de Lyncex. `HomePage` es un `ContentController`. `BookPage` y `BookListPage` son `TemplateControllers`, ambos usando `handler`, `queries` y `parámetros`. `LoginPage` es un `LoginController`, y `BookForm` y `GeoForm` son `FormController` con control de acceso.

8.6. Validación y pruebas

La aplicación de ejemplo no dispone de pruebas automatizadas, pero sí se han ejecutado pruebas manuales. Cabe destacar que la implementación de esta aplicación sacó a la luz bugs originales de Lyncex. Gracias a ello se implementó la función `exists` dentro del sistema de plantillas.

8.7. Tour

Para comprobar el funcionamiento de esta aplicación de ejemplo, vamos a hacer un tour de alto nivel. Suponemos que tenemos una instancia de Lyncex arrancada en `localhost`, en el puerto 11011, y está vacío el `triplestore`.

Lo primero será añadir las tripletas necesarias. Hay tres archivos:

- `vcard.ttl` - Una miniontología RDF Schema que simula a la de `vCard` (la cual es OWL)
- `bibliocyl.ttl` - Datos de fuentes abiertas, previamente transformados a Turtle desde RDF/XML
- `bibliocylapp.ttl` - La app en Lyncex

La manera de cargarlas, mediante `cURL`, es la siguiente:

```
curl -X POST -H "Content-Type: text/turtle; charset=utf-8" \
  --data-binary "@vcard.ttl" http://localhost:11011/_api
```

```
curl -X POST -H "Content-Type: text/turtle; charset=utf-8" \  
  --data-binary "@bibliocyl.ttl" http://localhost:11011/_api  
curl -X POST -H "Content-Type: text/turtle; charset=utf-8" \  
  --data-binary "@bibliocylapp.ttl" http://localhost:11011/_api
```

Una vez hecho esto, podemos acceder a diferentes páginas desde el navegador. La página de inicio estará disponible escribiendo localhost:11011 en el navegador, tal y como se ve en la figura 8.2.

Esto corresponde a las siguientes tripletas:

```
<HomePage>  
  a lcx:ContentController ;  
  lcx:url "" ;  
  lcx:method "get" ;  
  
  lcx:content [  
    a cnt:ContentAsText ;  
    cnt:chars ""  
<h1>Bienvenido a BiblioCyL</h1>  
<p>Aplicación creada por Adrián Arroyo</p>  
<p><a href=/books>Ver listado bibliotecas</a></p>  
""  
  ] ;  
  lcx:mime "text/html" .
```

Hacemos click en el enlace. Esto nos lleva al BookListPage, el listado de todas las bibliotecas del dataset, que podemos ver en la figura 8.3.

Esto se corresponde a estas tripletas:

```
<BookListPage>  
  a lcx:TemplateController ;  
  lcx:url "books" ;  
  lcx:method "get" ;  
  lcx:template [  
    a cnt:ContentAsText ;  
    cnt:chars ""  
    <h1>Listado de bibliotecas de Castilla y León</h1>  
    <ul>  
      {% each cards , card %}  
        <li>  
          <a href=/book?id={{ card.id }}>{{ card.name }}</a>  
        </li>  
      {% end %}  
    </ul>
```

```
    """
] ;

lcx:handler [
  a lcx:Handler ;
  lcx:handler_name "cards" ;
  lcx:code """
  handler(_Param, Output) :-
    findall(Card, (
      rdf(Biblio, rdf:type, 'http://www.w3.org/2006/vcard/ns#
        VCard'),
      db(Biblio, v:fn, Name),
      atom_concat('http://book.lyncex.com/aarroyoc/dev/lyncex/
        apps/', Id, Biblio),
      Card = _{id:Id, name:Name}
    ), Output).
  """
] .
```

Hacemos click en una biblioteca cualquiera, en este caso, la de Berlanga de Duero. Esto nos mostrará la vista con detalles de la biblioteca que son la URL y el correo, si existieran, mostrando además un mapa de su localización. Esto es el BookPage. Se muestra en la figura 8.4.

El BookPage se implementa con estas tripletas:

```
<BookPage>
  a lcx:TemplateController ;
  lcx:url "book" ;
  lcx:method "get" ;
  lcx:parameter [
    a lcx:Parameter ;
    lcx:param_name "id"
  ] ;

  lcx:template [
    a cnt:ContentAsText ;
    cnt:chars """
    <h1>{{ card.fn }}</h1>
    {% if exists(atom(url), card) %}
    <p>URL: <a href={{ card.url }}>{{ card.url }}</a></p>
    {% end %}
    {% if exists(atom(email), card) %}
    <p><a href={{ card.email }}>Email</a></p>
    {% end %}
    <iframe width=425 height=350 frameborder=0
      scrolling=no marginheight=0 marginwidth=0
      src=https://www.openstreetmap.org/export/
```

```
embed.html?bbox=-10.206298828125002%2C39.
232253141714914%2C1.3842773437500002%2C44.
33956524809713&marker={{ geo.lat }}%2C{{ geo.lon }}
style=border: 1px solid black></iframe>
"""

] ;

lcx:query [
  a lcx:Query ;
  lcx:query_name "card" ;
  lcx:template_subject "http://book.lyncex.com/aarroyoc/dev/
    lyncex/apps/{{ id }}"
] ;

lcx:handler [
  a lcx:Handler ;
  lcx:handler_name "geo" ;
  lcx:code """
  handler(Param, Output) :-
    get_dict(id, Param, Id),
    atom_concat('http://book.lyncex.com/aarroyoc/dev/lyncex/
      apps/', Id, Biblio),
    rdf(Biblio, v:geo, Geo),
    db(Geo, v:latitude, Lat),
    db(Geo, v:longitude, Lon),
    Output = _{lat:Lat, lon:Lon}.
  """
] .
```

Vamos a añadir una biblioteca nueva. Si tratamos de acceder a `/admin/book` tendremos un mensaje de *Forbidden* y no podremos ver el formulario. Accedemos primero a `/login`, donde se nos mostrará el `LoginPage`, como se aprecia en la figura 8.5:

```
<LoginPage>
  a lcx:LoginController ;
  lcx:url "login" ;
  lcx:username "aarroyoc" ;
  lcx:password "8
    d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92
  " ;
  lcx:template [
    a cnt:ContentAsText ;
    cnt:chars "{% unescape login %}"
  ] .
```

Una vez hayamos ingresado los credenciales correctos, ya podremos entrar a `/admin/book` (el `BookForm`, figura 8.6). Se nos mostrará un formulario con diversos campos a rellenar. El primero

es la IRI del sujeto sobre el que vamos a crear tripletas nuevas. Es necesario modificarlo, el valor que hay es solo una plantilla. Además para el campo geo deberemos indicar una IRI del sujeto geo. Este sujeto lo podemos crear entrando a /admin/geo (es el GeoForm).

Si queremos editar el contenido podemos acceder a la URL /admin/book?_id=IRI (figura 8.7), una vez allí veremos los campos rellenos, podremos editar el contenido y podremos borrar el sujeto del triplestore.

```
<BookForm>
  a lcx:FormController ;
  lcx:url "admin/book" ;
  lcx:access "private" ;
  lcx:base_subject "http://book.lyncex.com/aarroyoc/dev/lyncex/apps
/" ;
  lcx:class <http://www.w3.org/2006/vcard/ns#VCard> ;

  lcx:template [
    a cnt:ContentAsText ;
    cnt:chars """
    <h1>Bibliotecas de Castilla y León</h1>
    <style>
    input{
      display:block;
      width:500px;
    }
    </style>
    {% unescape form %}
    """
  ] .
```

Con esto, hemos visto las características principales de Lyncex aplicadas sobre un conjunto de datos reales.

8.8. Conclusiones

El desarrollo de esta aplicación ha servido para encontrar fallos y extraer algunas conclusiones.

Por un lado se ha visto la necesidad de permitir desactivar la validación RDF Schema, ya que en ciertos datasets no se cumplen las restricciones de forma exacta, y menos en estos datos, que realmente vienen definidos por una ontología OWL. También queda para el futuro ser compatibles con OWL, al menos al mismo nivel que con RDF Schema.

También hemos visto el problema de las interfaces en los formularios pregenerados. ¿Qué idioma debe mostrar el botón? ¿Las IRI son suficientemente explicativas para mostrar directamente?

Se han detectado errores en la gestión de propiedades opcionales que requirieron de soluciones no planteadas antes.



Figura 8.2: HomePage de BiblioCyL

Listado de bibliotecas de Castilla y León

- [Biblioteca Pública Municipal "Miguel de Unamuno" de Palencia](#)
- [Biblioteca Pública Municipal de Aranda de Duero](#)
- [Biblioteca Pública Municipal de Ayllón](#)
- [Biblioteca Pública Municipal Centro Cultural de Benavente](#)
- [Biblioteca Pública Municipal de Cantalejo](#)
- [Biblioteca Pública Municipal de Cantimpalos](#)
- [Biblioteca Pública Municipal de Carbonero el Mayor](#)
- [Biblioteca Pública Municipal de Coca](#)
- [Biblioteca Pública Municipal de El Espinar](#)
- [Biblioteca Pública Municipal de Mozoncillo](#)
- [Biblioteca Pública Municipal de Nava de la Asunción](#)
- [Biblioteca Pública Municipal de Navas de Oro](#)
- [Biblioteca Pública Municipal de Riaza](#)
- [Biblioteca Pública Municipal de San Cristóbal de Segovia](#)
- [Biblioteca Pública Municipal Carlos Parrondo. San Ildefonso](#)
- [Biblioteca Pública Municipal Ramón Menéndez Pidal. San Rafael](#)
- [Biblioteca Pública Municipal de Turégano](#)
- [Biblioteca Pública Municipal "Cronista Herrera" de Cuéllar](#)
- [Biblioteca de Caja Rural. Fuentepelayo](#)
- [Biblioteca Popular "San Miguel" de Caja Segovia. Sacramenia](#)
- [Biblioteca Pública Municipal de Candeleda](#)
- [Biblioteca Pública Municipal de Arenas de San Pedro](#)
- [Biblioteca Pública Municipal de Ólvega](#)
- [Biblioteca Pública Municipal de El Barco de Ávila](#)
- [Biblioteca Pública Municipal de El Barraco](#)
- [Biblioteca Pública Municipal de Arévalo](#)
- [Biblioteca Pública Municipal de Toro](#)
- [Biblioteca Pública Municipal "San José Obrero"](#)
- [Biblioteca Pública Municipal Candelaria-Bloques de Zamora](#)
- [Biblioteca Pública Municipal "Antonio Gamoneda" de La Pola de Gordón](#)
- [Biblioteca Pública Municipal de Casavieja](#)

Figura 8.3: BookListPage

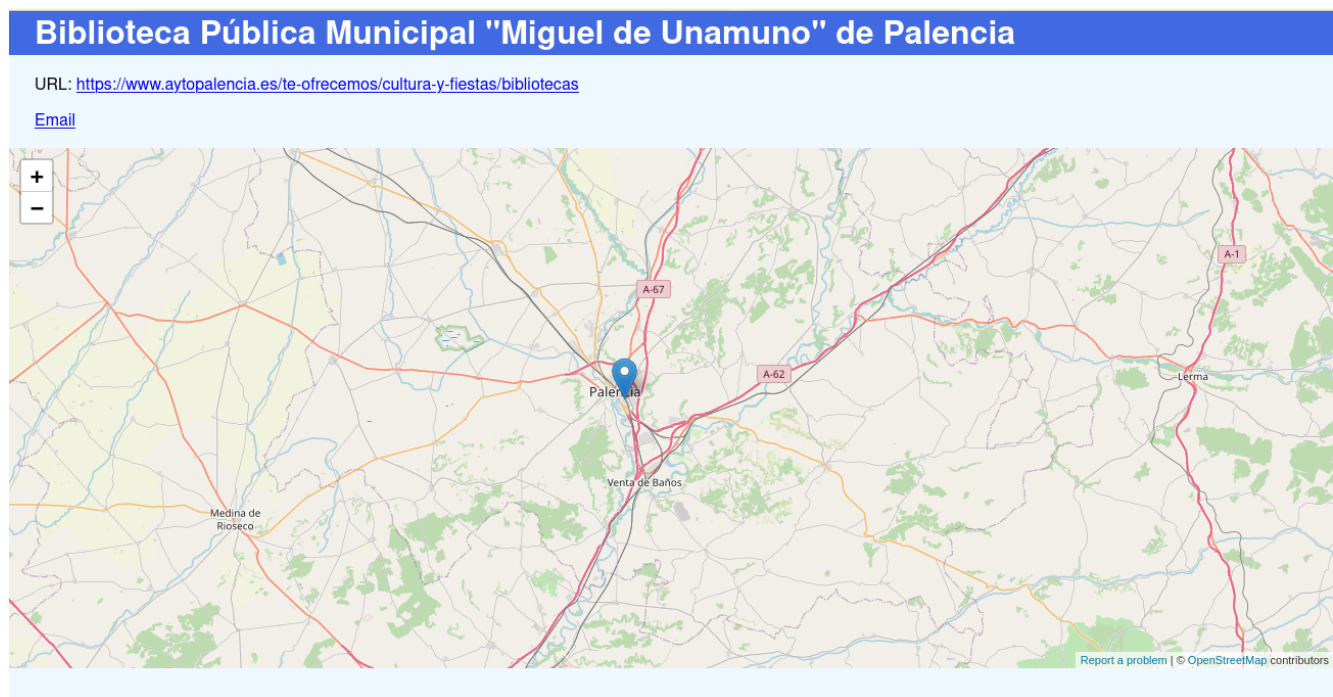
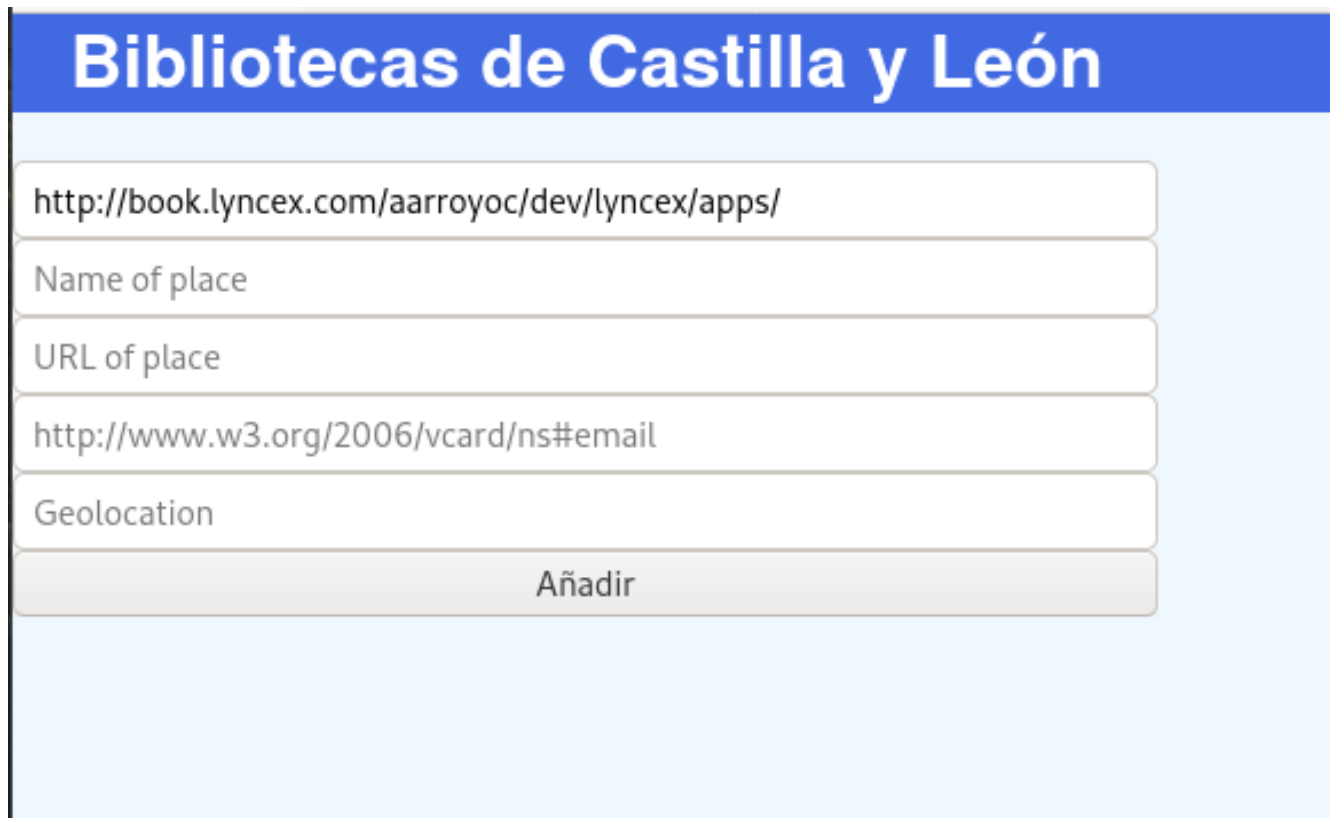


Figura 8.4: BookPage

Login

[Añadir biblioteca](#)[Añadir geolocalización](#)

Figura 8.5: LoginPage



Bibliotecas de Castilla y León

`http://book.lyncex.com/aarroyoc/dev/lyncex/apps/`

Name of place

URL of place

`http://www.w3.org/2006/vcard/ns#email`

Geolocation

Añadir

Figura 8.6: BookForm

Sobre aspectos positivos se ha encontrado que, aunque hay todavía muchos detalles sin pulir, en general la aplicación es fácil de programar. El programador que usa Lyncex necesita poco tiempo para realizar la aplicación entera.

Como aspectos negativos, hemos visto que es difícil actualizar la aplicación una vez está ejecutándose, ya que, aunque es posible, es difícil seleccionar todo lo que queremos reemplazar. Además, el contenido estático complementario (como el CSS) no se adapta muy bien al formato de tener que introducirlo como un string más.

Bibliotecas de Castilla y León

http://book.lyncex.com/aarroyoc/dev/lyncex/apps/1191563299709

Biblioteca Pública Municipal

https://www.aytopalencia.es/te-ofrecemos/cultura-y-fiestas/bibliotecas

mailto: bibliotecaunamuno@aytopalencia.es

_:genid1

Editar

Borrar

Figura 8.7: BookForm edición y borrado

Capítulo 9

Manuales

En este capítulo se incluyen dos manuales para el uso de Lyncex.

9.1. Manual de instalación

El objetivo de este manual es disponer de un servidor de Lyncex funcionando correctamente en un equipo informático.

Lyncex es una aplicación implementada completamente en Prolog, sin embargo, no usa Prolog estándar sino que necesita específicamente el intérprete SWI Prolog. Este intérprete es open source y se encuentra disponible en gran cantidad de sistemas operativos y arquitecturas diferentes. Sin embargo, solo se ha probado su funcionamiento correcto en sistemas Linux con arquitectura x86 de 64 bits. Es necesario tener conexión a Internet durante la instalación.

9.1.1. Docker

Para disponer de un entorno lo más controlado y aislado posible del resto del sistema se usan tecnologías de contenedores, concretamente Docker. Una de las ventajas de Docker es que también se puede usar en MacOS y Windows, mediante una virtualización de Linux automática que realiza Docker.

El primer paso será tener instalado Docker y su utilidad Docker Compose. Este software suele estar en los repositorios de las principales distribuciones Linux. En Debian o Ubuntu sería así:

```
sudo apt update
sudo apt install docker.io docker-compose
```

Una vez instalado, podemos obtener el código de la aplicación. Para esto hará falta tener Git, aunque desde GitHub también se pueden generar comprimidos tar.gz o zip que no necesitan tener instalado Git:

```
git clone https://github.com/aarroyoc/lyncex/ lyncex
cd lyncex
```

Una vez hemos descargado el código y tenemos Docker, podemos ejecutar el comando de Docker Compose para construir todas las imágenes de la aplicación:

```
docker-compose build
```

Esto comenzará la descarga de Prolog, así como de Python y numerosas dependencias. Guardará las imágenes en el almacén local de Docker.

Ahora podemos realizar la configuración de los contenedores. Edita el fichero `docker-compose.yml` con un editor de textos cualquiera. El campo más importante es el de ports. Aquí podemos configurar el puerto sobre el que va a estar escuchando Lyncex. Para realizar el cambio hay que modificar “11011:11011.” a “PUERTO_DESEADO:11011”. Otra configuración de interés es la activación o no del validador de RDF Schema. La variable de entorno `LYNCEX_RDF_SCHEMA_VALIDATION` controla su activación. Por defecto está a `true`, necesario además para pasar los tests con éxito, pero en ciertas ocasiones puede tener sentido desactivar la validación ya que se trabaja con datos que no siguen la ontología de forma estricta.

Una vez configurado, ya podemos ejecutar Lyncex:

```
docker-compose up lyncex
```

Y Lyncex estará ya disponible.

9.1.2. Tests

Una vez instalado, puede ser interesante ejecutar los tests automatizados para comprobar que efectivamente la versión instalada tiene un cierto nivel de confiabilidad.

Para ello lo primero es suspender la ejecución del contenedor web, con Control-C o mediante el comando siguiente:

```
docker-compose down
```

Para ejecutar los tests unitarios:

```
docker-compose run test-unit
```

Para ejecutar los tests E2E:

```
docker-compose run test-e2e
```

Una vez finalizada la ejecución de los tests, hay que destruir todos los contenedores. Para ello volvemos a ejecutar:

```
docker-compose down
```


9.2. Manual de usuario

En este manual veremos cómo usar Lyncex una vez instalado.

Para el manual usaremos la herramienta cURL, para realizar peticiones HTTP mediante comandos sencillos. También se pueden usar otras herramientas similares como Postman, o incluso realizar las llamadas HTTP desde un programa personalizado.

Vamos a suponer que Lyncex se encuentra ejecutándose en localhost en el puerto 11011.

La primera tarea que debe realizarse es subir tripletas. Estas deben estar guardadas en un fichero Turtle. Para subir tripletas usaremos el verbo POST de HTTP y el endpoint de la API es `/_api`.

Veamos un ejemplo, si tenemos este fichero Turtle, con el nombre `author.ttl`:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ex: <http://example.org/stuff/1.0/> .

<http://www.w3.org/TR/rdf-syntax-grammar>
  dc:title "RDF/XML Syntax Specification (Revised)" ;
  ex:editor [
    ex:fullname "Dave Beckett";
    ex:homePage <http://purl.org/net/dajobe/>
  ] .
```

Podemos guardar las tripletas de la siguiente forma:

```
curl -X POST -H "Content-Type: text/turtle; charset=utf-8" \
  --data-binary "@author.ttl" http://localhost:11011/_api
```

Si el fichero es correcto y la validación de ontologías también lo es, responderá OK.

La siguiente operación que podemos hacer es obtener de vuelta los datos insertados en el triplestore. Para ello usamos el verbo GET y el endpoint `/_api/query`.

```
curl -X GET http://localhost:11011/_api/query
```

El resultado seguramente sea diferente al fichero original, pero equivalente en cuanto a información:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix stuff: <http://example.org/stuff/1.0/> .
```

```
# Named toplevel resources (1)
```

```
<http://www.w3.org/TR/rdf-syntax-grammar>
  stuff:editor [ stuff:fullname "Dave Beckett" ;
                 stuff:homePage <http://purl.org/net/dajobe/>
               ] ;
  dc:title "RDF/XML Syntax Specification (Revised)" .
```

La API GET soporta filtros, para obtener grafos parciales. Para ello se fija el valor del sujeto, objeto o predicado a un valor fijo y se sacan todas las tripletas que cumplen con ese valor fijo en la posición correspondiente. Esto se hace mediante los parámetros GET llamados subject, predicate y object respectivamente. Por ejemplo, si queremos filtrar por las tripletas que tienen dc:title como predicado:

```
curl -X GET \
"http://localhost:11011/_api/query?predicate=http://purl.org/dc/
elements/1.1/title"
```

Obtendremos como resultado:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
```

```
# Named toplevel resources (1)
```

```
<http://www.w3.org/TR/rdf-syntax-grammar>
  dc:title "RDF/XML Syntax Specification (Revised)" .
```

También podemos borrar las tripletas almacenadas mediante el verbo DELETE y el endpoint `/_api/delete`. Este endpoint también soporta parámetros similar al GET con filtros. El uso es sencillo:

```
curl -X DELETE "http://localhost:11011/_api/delete"
```

Capítulo 10

Conclusiones y trabajo futuro

Para acabar la memoria, en este último capítulo vamos a repasar algunas de las conclusiones obtenidas, así como vías de trabajo futuras.

10.1. Conclusiones

Por un lado, la metodología Scrum ha resultado ser efectiva. Creemos que en este caso, donde podía haber requisitos cambiantes, ha sido una elección correcta. La planificación sin embargo, se ha desviado mucho respecto a la realidad, aunque esto es atribuible a la falta de experiencia.

Las librerías de Prolog han sorprendido por su madurez. Son perfectamente válidas en entornos de producción a pesar de su rareza. No son extremadamente complejas, pero optimizan los casos de uso principales de forma muy ergonómica y adaptada al lenguaje. Además, la capacidad de debugging y testing nos ha sorprendido gratamente. También merece destacar la ayuda que han proporcionado los desarrolladores de SWI-Prolog ante cualquier problema detectado.

El lenguaje en sí, ha demostrado ser a la vez un desafío y una profunda satisfacción. Porque es un lenguaje idóneo para expresar ciertas relaciones, para trabajar con datos, pero debemos recordar que no podemos usar técnicas de otros lenguajes que ya están interiorizadas. Este problema de rareza también se ha notado al tratar de elaborar diagramas UML.

RDF por su parte ha demostrado ser un sistema potente, pero con lagunas. Si bien es una forma esencial de representación de la información, falla a la hora de ser manipulado para realizar ediciones. No existe forma de modificar tripletas, siendo lo más común borrar y volver a crear, sin embargo, esto requiere saber la tripleta exacta que queremos borrar, qu en el caso de las propiedades con multiplicidad, se complica.

El testing mediante BDD ha sido extremadamente útil y práctico. Ha permitido detectar regresiones y en prácticamente todo el ciclo de desarrollo. Además permitía saber fácilmente cuando una tarea había sido completada con éxito.

Se ha intentado en todo momento que la experiencia de desarrollo sea ideal. El framework ha demostrado ser flexible, permitiendo realizar prácticamente lo que se quiera hacer, aunque convenientemente optimizado solamente para las tareas más comunes, como son los CRUD y los

formularios. Es sencillo de instalar y configurar en cualquier distribución Linux, gracias a Docker y Docker Compose.

Por último, el hecho de tener una ontología por separado, definida de forma independiente pero abierta, permite que se pueda extender por otros autores, dotándole de mayor funcionalidad. E implementaciones independientes pueden tratar de implementar la funcionalidad semántica de igual forma que lo hace Lyncex, mejorando en aspectos que se consideren peores de Lyncex.

10.2. Trabajo futuro

En cuanto al diseño, se han quedado muchas cosas interesantes en el tintero. Una propuesta de futuro sería desacoplar el Triplestore del propio proceso web. Esto se eligió por simplicidad, pero de cara a un producto más profesional, sería necesario que la base de datos tuviese más potencia y escalabilidad. Un triplestore externo podría lograrlo de forma sencilla.

También es importante un aspecto aquí parcialmente ignorado, como es la seguridad. En futuras iteraciones se debería implementar un sistema de control de acceso más potente, así como una gestión de secretos más ergonómica. Existen diversas formas de diseñar este sistema. Un control de acceso fino a nivel de tripleta puede ser excesivo pero sería muy flexible.

El contenido estático también debería poder ser más ergonómico, ya que inyectarlo dentro de ficheros Turtle es un proceso incómodo y propenso a errores. En contenido binario este problema se magnifica.

Respecto a la usabilidad de cara al desarrollador, opinamos que permitir solo la comunicación mediante una interfaz Turtle no es óptimo de cara al desarrollador. Quizá una herramienta o un IDE para interactuar específicamente con este sistema arreglaría muchos problemas de ergonomía encontrados.

10.3. Palabras finales

A nivel personal considero que la aplicación cumple con los objetivos marcados, y que si bien, no cumple los requisitos para ser usado en producción, es suficiente para proponer este concepto de plataforma de desarrollo.

Durante este tiempo he podido aprender y profundizar sobre ciertos aspectos de la informática que durante el resto de asignaturas no pude hacer. Esto me ha servido para desmitificar ciertas tecnologías e ideas y poder tener una opinión más formada sobre este tipo de tecnologías.

Apéndice

Apéndice A

Ontología

```
@prefix : <https://lyncex.com/lyncex#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix cnt: <http://www.w3.org/2011/content#> .
```

```
# CNT
```

```
cnt:Content
  a rdfs:Class .
```

```
cnt:ContentAsText
  a rdfs:Class ;
  rdfs:subClassOf cnt:Content .
```

```
cnt:ContentAsBase64
  a rdfs:Class ;
  rdfs:subClassOf cnt:Content .
```

```
cnt:bytes
  a rdf:Property ;
  rdfs:domain cnt:ContentAsBase64 ;
  rdfs:range xsd:string .
```

```
cnt:chars
  a rdf:Property ;
  rdfs:domain cnt:ContentAsText ;
  rdfs:range xsd:string .
```

```
# GENERAL
```

```
:Application
  rdf:type rdfs:Class .
```

```
:controller
  rdf:type rdf:Property ;
  rdfs:domain :Application ;
  rdfs:range :Controller .

:prefix
  rdf:type rdf:Property ;
  rdfs:domain :Application ;
  rdfs:range :Prefix .

:Controller
  rdf:type rdfs:Class .

:url
  rdf:type rdf:Property ;
  rdfs:domain :Controller ;
  rdfs:range xsd:string .

:method
  rdf:type rdf:Property ;
  rdfs:domain :Controller ;
  rdfs:range xsd:string .

:access
  rdf:type rdf:Property ;
  rdfs:domain :Controller ;
  rdfs:range xsd:string .

# ContentController

:ContentController
  rdf:type rdfs:Class ;
  rdfs:subClassOf :Controller .

:content
  rdf:type rdf:Property ;
  rdfs:domain :ContentController ;
  rdfs:range cnt:Content .

:mime
  rdf:type rdf:Property ;
  rdfs:domain :ContentController ;
  rdfs:range xsd:string .

# TemplateController

:TemplateController
  rdf:type rdfs:Class ;
```



```
    rdf:subClassOf :Controller .
```

```
:template
```

```
    rdf:type rdf:Property ;
    rdfs:domain :TemplateController ;
    rdfs:range cnt:ContentAsText .
```

```
:query
```

```
    rdf:type rdf:Property ;
    rdfs:domain :TemplateController ;
    rdfs:range :Query .
```

```
:handler
```

```
    rdf:type rdf:Property ;
    rdfs:domain :TemplateController ;
    rdfs:range :Handler .
```

```
:parameter
```

```
    rdf:type rdf:Property ;
    rdfs:domain :TemplateController ;
    rdfs:range :Parameter .
```

```
## Query
```

```
:Query
```

```
    rdf:type rdfs:Class .
```

```
:query_name
```

```
    rdf:type rdf:Property ;
    rdfs:domain :Query ;
    rdfs:range xsd:string .
```

```
:subject
```

```
    rdf:type rdf:Property ;
    rdfs:domain :Query ;
    rdfs:range xsd:string .
```

```
:template_subject
```

```
    rdf:type rdf:Property ;
    rdfs:domain :Query ;
    rdfs:range xsd:string .
```

```
## Handler
```

```
:Handler
```

```
    rdf:type rdfs:Class .
```

```
:handler_name
```

```

    rdf:type rdf:Property ;
    rdfs:domain :Handler ;
    rdfs:range xsd:string .

:code
    rdf:type rdf:Property ;
    rdfs:domain :Handler ;
    rdfs:range xsd:string .

## Parameter

:Parameter
    a rdfs:Class .

:param_name
    a rdf:Property ;
    rdfs:domain :Parameter ;
    rdfs:range xsd:string .

:code
    a rdf:Property ;
    rdfs:domain :Parameter ;
    rdfs:range xsd:string .

:validation
    a rdf:Property ;
    rdfs:domain :Parameter ;
    rdfs:range xsd:string .

# FormController

:FormController
    a rdfs:Class ;
    rdfs:subClassOf :TemplateController .

:base_subject
    a rdf:Property ;
    rdfs:domain :FormController ;
    rdfs:range xsd:string .

:class
    a rdf:Property ;
    rdfs:domain :FormController ;
    rdfs:range rdfs:Class .

# LoginController
:LoginController
    a rdfs:Class ;

```

```
    rdfs:subClassOf :TemplateController .
```

```
:username
```

```
    a rdf:Property ;  
    rdfs:domain :LoginController ;  
    rdfs:range xsd:string .
```

```
:password
```

```
    a rdf:Property ;  
    rdfs:domain :LoginController ;  
    rdfs:range xsd:string .
```

```
## Prefix
```

```
:Prefix
```

```
    rdf:type rdfs:Class .
```

```
:namespace
```

```
    rdf:type rdf:Property ;  
    rdfs:domain :Prefix ;  
    rdfs:range xsd:string .
```

```
:prefix_name
```

```
    rdf:type rdf:Property ;  
    rdfs:domain :Prefix ;  
    rdfs:range xsd:string .
```


Apéndice B

Crterios de aceptación en Gherkin

En este apéndice se presentan los criterios de aceptación reales, expresados en lenguaje Gherkin, que se han usado en Behave para validar que se iban cumpliendo los requisitos funcionales.

Feature: HTTP API to work with Lyncex

Scenario: GET with an empty database

Given I have an empty Lyncex instance
When I do a GET request
Then I get a 200 status code
And I get an empty response

Scenario: POST with an empty database

Given I have an empty Lyncex instance
When I do a POST request with 'features/test1.ttl' data
Then I get a 200 status code
And I get a 'OK' response

Scenario: GET with a non-empty database

Given I have an empty Lyncex instance
And I do a POST request with 'features/test1.ttl' data
When I do a GET request
Then I get a 200 status code
And I get the contents of 'features/test1.ttl'

Scenario: GET with filter

Given I have an empty Lyncex instance
And I do a POST request with 'features/test1.ttl' data
When I do a filtered (subject='https://lyncex.com/lyncex#quijote') GET request
Then I get a 200 status code
And I get the contents of 'features/test1_filtered2.ttl'

Scenario: DELETE a non-empty database

Given I have an empty Lyncex instance

And I do a POST request with 'features/test1.ttl' data
When I do a DELETE request
Then I get a 200 status code
And I get a 'OK' response

Scenario: DELETE with filter

Given I have an empty Lyncex instance
And I do a POST request with 'features/test1.ttl' data
When I do a filtered (subject='https://lyncex.com/lyncex#quijote
') DELETE request
Then I get a 200 status code
And I get a 'OK' response
And I do a GET request
And I get the contents of 'features/test1_filtered.ttl'

Scenario: RDF Schema validation

Given I have an empty Lyncex instance
And I do a POST request with 'features/test1.ttl' data
When I do a POST request with 'features/test1_bad.ttl' data
Then I get a 'NOT VALID' response
#And I get a 401 status code
And I do a GET request
And I get the contents of 'features/test1.ttl'

Feature: Forms in Lyncex

Scenario: Save content of a form

Given I have an empty Lyncex instance
And I do a POST request with 'features/test4.ttl' data
When I submit the form '/form1' with data '_id=https://app.lyncex
.com/book/Quijote' and 'name=Don Quijote'
Then I visit '/book?id=Quijote'
And I get a 'Name: Don Quijote' response
And I get a 'text/html' response type
And I get a 200 status code

Scenario: Save content of a form (valid)

Given I have an empty Lyncex instance
And I do a POST request with 'features/test4.ttl' data
When I submit the form '/form2' with data '_id=https://app.lyncex
.com/book/Quijote' and 'xname=Quijote'
Then I visit '/book2?id=Quijote'
And I get a 'Name: Quijote' response
And I get a 200 status code
And I get a 'text/html' response type

Scenario: Save content of a form (invalid)

Given I have an empty Lyncex instance
And I do a POST request with 'features/test4.ttl' data

When I submit the form '/form2' with data '_id=https://app.lyncex.com/book/Quijote' and 'xname=Don Quijote'
 Then I visit '/book2?id=Quijote'
 And I get a 500 status code

Scenario: Autogenerate form

Given I have an empty Lyncex instance
 And I do a POST request with 'features/test4.ttl' data
 When I visit '/form1'
 And I get a '<form method="POST"><input type="url" name="_id" value="https://app.lyncex.com/book/"><input type="text" placeholder="https://app.lyncex.com/name" name="https://app.lyncex.com/name"><input type="submit" value="Añadir"></form>' response
 And I get a 'text/html' response type
 And I get a 200 status code

Scenario: Multiple triples (generated form)

Given I have an empty Lyncex instance
 And I do a POST request with 'features/test4.ttl' data
 When I visit '/form3'
 And I get a '<form method="POST"><input type="url" name="_id" value="https://app.lyncex.com/book/"><textarea placeholder="https://app.lyncex.com/author" name="https://app.lyncex.com/author"></textarea><input type="submit" value="Añadir"></form>' response
 And I get a 'text/html' response type
 And I get a 200 status code

Scenario: Multiple triples (save)

Given I have an empty Lyncex instance
 And I do a POST request with 'features/test4.ttl' data
 When I submit the form '/form3' with data '_id=https://app.lyncex.com/book/SuperLibro' and 'author=small'
 Then I visit '/book3?id=SuperLibro'
 And I get a 'Authors: Cervantes, Lope de Vega,' response
 And I get a 200 status code
 And I get a 'text/html' response type

Scenario: Visualize saved triples

Given I have an empty Lyncex instance
 And I do a POST request with 'features/test4.ttl' data
 And I submit the form '/form3' with data '_id=https://app.lyncex.com/book/SuperLibro' and 'author=small'
 When I visit '/form3?id=https://app.lyncex.com/book/SuperLibro'
 Then I get the following response

"""

<form action="/form3" method="POST"><input readonly type="url

```

    " name="_id" value="https://app.lyncex.com/book/SuperLibro
"><textarea placeholder="https://app.lyncex.com/author"
name="https://app.lyncex.com/author">Cervantes
Lope de Vega
</textarea><input type="submit" value="Editar"></form><form
method="GET"><input type="hidden" name="_delete" value="
yes"><input type="hidden" name="_id" value="https://app.
lyncex.com/book/SuperLibro"><input type="submit" value="
Borrar"></form>
"""

```

And I get a 200 status code
And I get a 'text/html' response type

Scenario: Delete triples

Given I have an empty Lyncex instance
And I do a POST request with 'features/test4.ttl' data
And I submit the form '/form3' with data '_id=https://app.lyncex.com/book/SuperLibro' and 'author=small'
When I visit '/form3?_delete=yes&_id=https://app.lyncex.com/book/SuperLibro'
Then I get a 'OK' response
And I get a 200 status code
And I get a 'text/html' response type

Scenario: Update triples (add more)

Given I have an empty Lyncex instance
And I do a POST request with 'features/test4.ttl' data
And I submit the form '/form3' with data '_id=https://app.lyncex.com/book/SuperLibro' and 'author=small'
When I submit the form '/form3' with data '_id=https://app.lyncex.com/book/SuperLibro' and 'author=large'
Then I visit '/book3?id=SuperLibro'
And I get a 'Authors: Cervantes, Lope de Vega, Pepito,' response
And I get a 200 status code
And I get a 'text/html' response type

Scenario: Update triples (delete)

Given I have an empty Lyncex instance
And I do a POST request with 'features/test4.ttl' data
And I submit the form '/form3' with data '_id=https://app.lyncex.com/book/SuperLibro' and 'author=large'
When I submit the form '/form3' with data '_id=https://app.lyncex.com/book/SuperLibro' and 'author=small'
Then I visit '/book3?id=SuperLibro'
And I get a 'Authors: Cervantes, Lope de Vega,' response
And I get a 200 status code
And I get a 'text/html' response type

Scenario: Form also work for relationships

Given I have an empty Lyncex instance

And I do a POST request with 'features/test4.ttl' data

And I submit the form '/form4' with data '_id=https://app.lyncex.com/person/Mario' and 'friend=https://app.lyncex.com/person/Jaime'

Then I visit '/form4?_id=https://app.lyncex.com/person/Mario'

And I get the following response

"""

```
<form action="/form4" method="POST"><input readonly type="url"
  name="_id" value="https://app.lyncex.com/person/Mario"><
  textarea placeholder="https://app.lyncex.com/friend" name="
  https://app.lyncex.com/friend">https://app.lyncex.com/person/
  Jaime
</textarea><input type="submit" value="Editar"></form><form
  method="GET"><input type="hidden" name="_delete" value="yes"><
  input type="hidden" name="_id" value="https://app.lyncex.com/
  person/Mario"><input type="submit" value="Borrar"></form>
"""
```

And I get a 200 status code

And I get a 'text/html' response type

And I visit '/person4'

And I get a 'Friend: https://app.lyncex.com/person/Jaime' response

And I get a 200 status code

And I get a 'text/html' response type

Feature: Sessions in Lyncex – Forms 2

Scenario: Login form

Given I have an empty Lyncex instance

And I do a POST request with 'features/test5.ttl' data

When I visit '/login'

And I get a '<form method="POST"><input type="text" name="user"><input type="password" name="password"><input type="submit" value="Login"></form>' response

And I get a 'text/html' response type

And I get a 200 status code

Scenario: Do a login

Given I have an empty Lyncex instance

And I do a POST request with 'features/test5.ttl' data

When I login at '/login'

Then I get a '<p>¡Sesión iniciada!</p>' response

And I get a 'text/html' response type

And I get a 200 status code

Scenario: Deny access

Given I have an empty Lyncex instance
And I do a POST request with 'features/test5.ttl' data
When I visit '/private'
Then I get a 403 status code
And I get a 'text/html' response type

Scenario: Grant access

Given I have an empty Lyncex instance
And I do a POST request with 'features/test5.ttl' data
And I login at '/login'
When I visit with cookies '/private'
Then I get a 'Monty Python' response
And I get a 'text/plain' response type
And I get a 200 status code

Feature: Static content on Lyncex

Scenario: Get a non-existing page

Given I have an empty Lyncex instance
When I visit '/'
Then I get a 404 status code

Scenario: Load a static website

Given I have an empty Lyncex instance
And I do a POST request with 'features/test2.ttl' data
When I visit '/'
Then I get a 200 status code
And I get a 'About' response
And I get a 'text/html' response type

Scenario: Load a static asset

Given I have an empty Lyncex instance
And I do a POST request with 'features/test2.ttl' data
When I visit '/comuneros.jpg'
Then I get a 200 status code
And I get a 'image/jpeg' response type
And I get the photo 'features/ComunerosMini.jpg'

Feature: Show information with HTML-based templating

Scenario: No variable templates

Given I have an empty Lyncex instance
And I do a POST request with 'features/test3.ttl' data
When I visit '/'
Then I get a 200 status code
And I get a '<h1>Welcome to Lyncex</h1>' response
And I get a 'text/html' response type

Scenario: Mirror templates in Turtle

Given I have an empty Lyncex instance
And I do a POST request with 'features/test3.ttl' data
When I visit '/person'
Then I get a 200 status code
And I get a 'Name: Adrián Arroyo
Age: 21
Other person is: Mario Arroyo</p>' response
And I get a 'text/html' response type

Scenario: Code templates

Given I have an empty Lyncex instance
And I do a POST request with 'features/test3.ttl' data
When I visit '/person2'
Then I get a 200 status code
And I get a 'Name: Adrián Arroyo
Nombre 2: Mario Arroyo' response
And I get a 'text/html' response type

Scenario: Parameters (GET, valid)

Given I have an empty Lyncex instance
And I do a POST request with 'features/test3.ttl' data
When I visit '/person3?id=42'
Then I get a 200 status code
And I get a 'ID: 42' response
And I get a 'text/html' response type

Scenario: Parameters (GET, invalid)

Given I have an empty Lyncex instance
And I do a POST request with 'features/test3.ttl' data
When I visit '/person3?id=jojo'
Then I get a 500 status code

Scenario: Parameters (GET, no validation)

Given I have an empty Lyncex instance
And I do a POST request with 'features/test3.ttl' data
When I visit '/person4?id=pepo'
Then I get a 200 status code
And I get a 'ID: pepo' response
And I get a 'text/html' response type

Scenario: Parameters (POST, valid)

Given I have an empty Lyncex instance
And I do a POST request with 'features/test3.ttl' data
When I submit form '/person5' with data 'name=Adrián'
Then I get a 200 status code
And I get a 'ID: Adrián' response
And I get a 'text/html' response type

Scenario: Register db prefix

Given I have an empty Lyncex instance
And I do a POST request with 'features/test3.ttl' data
When I visit '/person6'
Then I get a 200 status code
And I get a 'Name: Adrián Arroyo
Nombre 2: Mario
Arroyo' response
And I get a 'text/html' response type

Scenario: Validate parameter with Prolog (valid)

Given I have an empty Lyncex instance
And I do a POST request with 'features/test3.ttl' data
When I visit '/person7?age=14'
Then I get a 200 status code
And I get a 'ID: 14' response
And I get a 'text/html' response type

Scenario: Validate parameter with Prolog (invalid)

Given I have an empty Lyncex instance
And I do a POST request with 'features/test3.ttl' data
When I visit '/person7?age=Pepe'
Then I get a 500 status code

Scenario: Query with template

Given I have an empty Lyncex instance
And I do a POST request with 'features/test3.ttl' data
When I visit '/person8?person=Mario'
Then I get a 200 status code
And I get a 'Name: Adrián Arroyo
Age: 21

Other person is: Mario Arroyo</p>' response
And I get a 'text/html' response type

Bibliografía

- [1] ADIDA, B., BIRBECK, M., SPORNY, M., AND HERMAN, I. RDFa 1.1 primer - third edition. W3C note, W3C, Mar. 2015. <http://www.w3.org/TR/2015/NOTE-rdfa-primer-20150317/>.
- [2] APACHE SOFTWARE FOUNDATION. Apache CouchDB . <https://couchdb.apache.org>. Versión: 3.0.0 Accedido: 2020-03-22.
- [3] APACHE SOFTWARE FOUNDATION. Apache Jena . <https://jena.apache.org/>. Versión: 3.14.0 Accedido: 2020-03-22.
- [4] BRICKLEY, D. Foaf vocabulary specification 0.9. <http://xmlns.com/foaf/spec/20070524.html> (2007).
- [5] CAROTHERS, G., AND SEABORNE, A. RDF 1.1 trig. W3C recommendation, W3C, Feb. 2014. <http://www.w3.org/TR/2014/REC-trig-20140225/>.
- [6] COHN, M. Why the fibonacci sequence works well for estimating. <https://www.mountaingoatsoftware.com/blog/why-the-fibonacci-sequence-works-well-for-estimating>. Accedido: 2020-06-21.
- [7] CONTRIBUTORS, D. C. Dublin core metadata initiative. <https://dublincore.org/>. Accedido: 2020-06-25.
- [8] CONTRIBUTORS, V. C. Couchapp: Web application hosted in apache couchdb. <https://couchapp.readthedocs.io/en/latest/>. Accedido: 2020-06-24.
- [9] DATACHEMIST. TerminusDB . <https://terminusdb.com/>. Versión: 1.1.2 Accedido: 2020-03-22.
- [10] DJANGO SOFTWARE FOUNDATION. Django . <https://www.djangoproject.com>. Versión: 3.0.4 Accedido: 2020-03-22.
- [11] FEENEY, K. Graph fundamentals — part 1: Rdf. <https://medium.com/terminusdb/graph-fundamentals-part-1-rdf-60dcf8d0c459>. Accedido: 2020-06-24.
- [12] FERNÁNDEZ, J., MARTÍNEZ-PRIETO, M. A., GUTIERREZ, C., POLLERES, A., AND ARIAS, M. Binary rdf representation for publication and exchange (hdt). *Journal of Web Semantics* 19 (03 2013), 22–41.
- [13] GAO, S., BEECH, D., MALONEY, M., THOMPSON, H., SPERBERG-McQUEEN, M., AND MENDELSON, N. W3C xml schema definition language (XSD) 1.1 part 1: Structures. W3C recommendation, W3C, Apr. 2012. <http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/>.

- [14] GUHA, R., AND BRICKLEY, D. RDF schema 1.1. W3C recommendation, W3C, Feb. 2014. <http://www.w3.org/TR/2014/REC-rdf-schema-20140225/>.
- [15] IANNELLA, R., AND MCKINNEY, J. vcard ontology - for describing people and organizations. W3C note, W3C, May 2014. <http://www.w3.org/TR/2014/NOTE-vcard-rdf-20140522/>.
- [16] INC., S. Stackoverflow 2020 developer survey. <https://insights.stackoverflow.com/survey/2020>. Accedido: 2020-06-24.
- [17] KELLOGG, G., LANTHALER, M., AND SPORNY, M. JSON-ld 1.0. W3C recommendation, W3C, Jan. 2014. <http://www.w3.org/TR/2014/REC-json-ld-20140116/>.
- [18] LANTHALER, M., CYGANIAK, R., AND WOOD, D. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, Feb. 2014. <http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>.
- [19] MARCO, F. J. G. Schema. org: la catalogación revisitada. *Anuario ThinkEPI* 7, 1 (2013), 169–172.
- [20] MARTÍNEZ-PRIETO, M. A., SILVESTRE, J., BREGÓN, A., GATÓN, V., ET AL. ¿ puede ser agile la docencia universitaria?(uvagile).
- [21] MICROSOFT. Visual Studio Code . <https://code.visualstudio.com/>. Versión: 1.44.2 Accedido: 2020-04-22.
- [22] OPENLINK SOFTWARE. Virtuoso . <https://virtuoso.openlinksw.com/>. Versión: 8.3 Accedido: 2020-03-22.
- [23] PRUD'HOMMEAUX, E., AND CAROTHERS, G. RDF 1.1 turtle. W3C recommendation, W3C, Feb. 2014. <http://www.w3.org/TR/2014/REC-turtle-20140225/>.
- [24] SCHREIBER, G., AND GANDON, F. RDF 1.1 XML syntax. W3C recommendation, W3C, Feb. 2014. <http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/>.
- [25] SCHWABER, K., AND SUTHERLAND, J. The scrum guide. november 2017, 2017.
- [26] SEABORNE, A., AND CAROTHERS, G. RDF 1.1 n-triples. W3C recommendation, W3C, Feb. 2014. <http://www.w3.org/TR/2014/REC-n-triples-20140225/>.
- [27] VELASCO, C. A., KOCH, J., AND ACKERMANN, P. Representing content in RDF 1.0. W3C note, W3C, Feb. 2017. <https://www.w3.org/TR/2017/NOTE-Content-in-RDF10-20170202/>.
- [28] W3C. W3c semantic web. https://www.w3.org/2001/sw/wiki/Main_Page. Accedido: 2020-06-24.
- [29] W3C. OWL 2 web ontology language document overview (second edition). W3C recommendation, W3C, Dec. 2012. <http://www.w3.org/TR/2012/REC-owl2-overview-20121211/>.
- [30] W3C. SPARQL 1.1 overview. W3C recommendation, W3C, Mar. 2013. <http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/>.
- [31] WIELEMAKER, J. Rdf save turtle expand and rdf literals. <https://swi-prolog.discourse.group/t/rdf-save-turtle-expand-and-rdf-literals/1968>. Accedido: 2020-06-25.

- [32] WIELEMAKER, J., BEEK, W., HILDEBRAND, M., AND VAN OSSENBRUGGEN, J. Cliopatria: a swi-prolog infrastructure for the semantic web. *Semantic Web* 7, 5 (2016), 529–541.
- [33] WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. Swi-prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96.
- [34] WIKIPEDIA. Arquitectura de von neumann — wikipedia, la enciclopedia libre, 2020. [Internet; descargado 24-junio-2020].
- [35] WIKIPEDIA CONTRIBUTORS. Convention over configuration — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Convention_over_configuration&oldid=956632496, 2020. [Online; accessed 25-June-2020].
- [36] WIKIPEDIA CONTRIBUTORS. Don't repeat yourself — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Don%27t_repeat_yourself&oldid=963600275, 2020. [Online; accessed 25-June-2020].
- [37] WIKIPEDIA CONTRIBUTORS. Notation3 — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=Notation3&oldid=958373033>, 2020. [Online; accessed 24-June-2020].
- [38] ÁNGELA MARTÍNEZ LABRADOR. El salario mínimo interprofesional 2020 entra en vigor con alguna sorpresa. <https://www.grupo2000.es/asi-sera-tu-nomina-en-2020-tras-congelarse-el-salario-minimo/>. Accedido: 2020-06-21.