

UNIVERSIDAD POLITÉCNICA DE CATALUÑA

FACULTAD DE INFORMÁTICA DE BARCELONA

INGENIERÍA DE SOFTWARE

---

# Repositorio de árboles genealógicos en BD NoSQL

---

*Autor:*

Daniel ALBARRAL NUÑEZ

*Director:*

Enric MAYOL



UNIVERSITAT POLITÈCNICA  
DE CATALUNYA  
BARCELONATECH



## 1. Resumen/Abstract

fsdfadf

# Índice

<b>1. Resumen/Abstract</b>	<b>1</b>
<b>2. Introducción y estado del arte</b>	<b>5</b>
2.1. Motivación . . . . .	5
2.2. Contextualización del proyecto . . . . .	6
2.2.1. Los árboles genealógicos . . . . .	6
2.2.2. Bases de datos orientadas a grafos (BDOG) . . . . .	6
2.2.3. REST-ful (REpresentational State Transfer) . . . . .	8
2.3. Perspectiva general del software genealógico actual. . . . .	8
2.4. Análisis previo de tecnologías . . . . .	9
2.4.1. Base de datos . . . . .	9
2.4.2. Lenguaje de programación . . . . .	10
2.5. Integración de leyes y regulaciones . . . . .	13
2.5.1. Derecho de información . . . . .	13
2.5.2. Consentimiento del afectado . . . . .	13
<b>3. Alcance</b>	<b>14</b>
3.1. Objetivos . . . . .	14
3.2. Objetivos académicos . . . . .	14
3.3. Metodología y rigor . . . . .	15
3.3.1. SCRUM . . . . .	15
3.4. Metodo de validación . . . . .	17
3.5. Fases de desarrollo . . . . .	17
3.5.1. Análisis de requisitos y estudio previo . . . . .	17
3.5.2. Especificación . . . . .	17
3.5.3. Diseño . . . . .	17
3.5.4. Implementación y pruebas . . . . .	18
3.6. Posibles obstáculos . . . . .	18
<b>4. Planificación</b>	<b>19</b>
4.1. Gantt . . . . .	19
4.1.1. Tareas y Gráficos gantt . . . . .	20
4.2. Recursos . . . . .	22
4.3. Alternativas y plan de acción . . . . .	22
<b>5. Presupuesto y impacto social</b>	<b>23</b>
5.1. Identificación y estimación de costes . . . . .	23
5.1.1. Costes de personal . . . . .	23
5.1.2. Coste del espacio de trabajo . . . . .	25
5.1.3. Coste del los servidores . . . . .	25

<b>6. Sostenibilidad y compromiso social</b>	<b>26</b>
6.1. Dimensión económica - 7 . . . . .	26
6.2. Dimensión social - 9 . . . . .	26
6.3. Dimensión ambiental - 6 . . . . .	26
<b>7. Análisis de requisitos</b>	<b>27</b>
7.1. Visión global del sistema . . . . .	27
7.2. Actores . . . . .	27
7.3. Requisitos funcionales . . . . .	27
7.3.1. Usuario final . . . . .	27
7.3.2. Aplicación . . . . .	28
7.4. Requisitos no funcionales . . . . .	28
<b>8. Especificación</b>	<b>30</b>
8.1. Casos de uso . . . . .	30
8.1.1. Usuario final . . . . .	30
8.2. Modelo conceptual de datos . . . . .	42
8.2.1. Restricciones de integridad . . . . .	43
<b>9. Diseño</b>	<b>44</b>
9.1. Patrones de arquitectónicos . . . . .	44
9.1.1. Modelo vista controlador . . . . .	44
9.1.2. Message Broker . . . . .	45
9.1.3. REST . . . . .	47
9.2. Patrones de mensajería . . . . .	47
9.2.1. <i>Publish-subscribe</i> . . . . .	47
9.3. Patrones Estructurales . . . . .	48
9.3.1. Mixins . . . . .	48
9.4. Protocolos . . . . .	48
9.4.1. OAuth 2 . . . . .	48
9.5. Diseño del modelo de datos . . . . .	51
9.5.1. Árboles temporales. . . . .	54
9.5.2. Geocomponentes . . . . .	55
9.6. Diseño de la capa de dominio . . . . .	56
9.7. Diseño del modelo de comportamiento . . . . .	59
<b>10. Implementación</b>	<b>60</b>
10.1. Usuarios . . . . .	60
10.1.1. SignUp . . . . .	60
10.1.2. LogIn . . . . .	63
10.2. Gestión de aplicaciones . . . . .	64
10.3. Endpoints . . . . .	65

10.3.1. CRUD . . . . .	66
10.3.2. Asíncronos . . . . .	67
10.3.3. UploadGedcom . . . . .	68
10.3.4. Encontrar similitudes . . . . .	70
<b>11.Pruebas</b>	<b>77</b>
11.1. Django rest framework test suit . . . . .	77
11.2. POSTMAN . . . . .	79
<b>12.Conclusiones</b>	<b>82</b>
12.1. Cambios en la planificación . . . . .	82
12.2. Objetivos académicos . . . . .	82
12.2.1. CES1.1 . . . . .	82
12.2.2. CES1.2 . . . . .	82
12.2.3. CES1.4 . . . . .	83
12.2.4. CES1.5 . . . . .	83
12.2.5. CES1.6 . . . . .	83
12.2.6. CES1.9 . . . . .	83
12.2.7. CES2.2 . . . . .	83
12.3. Evoluciones futuras . . . . .	84
<b>13.Referencias</b>	<b>85</b>

## 2. Introducción y estado del arte

Este apartado se estructurara de la siguiente forma: Primero en la motivación, se explican los motivos por los cuales se ha decidido dedicar el proyecto de final de grado a construir una aplicación *backend* para la gestión de árboles genealógicos usando bases de datos NoSQL, más concretamente basadas en grafos, también como se ha decidido ofrecer el servicio a la aplicación para gestionar estos árboles. Después se expondrá el estado del arte donde se contextualizara el proyecto. A continuación, se analizará el estado en el que se encuentra la tecnología con la cual realizaremos el desarrollo de este proyecto. Y por último, se definirán los objetivos que se han perseguido alcanzar con la consecución de este proyecto.

### 2.1. Motivación

La rápida evolución de las tecnologías constantemente nos acerca nuevas formas de aproximarnos a problemas que ya estaban resueltos, ya sea para resolver el problema de una forma más eficiente, elegante y/o versátil, entendamos por elegante que conceptualmente la solución propuesta nos resulta sencilla de comprender en contraposición a otra más compleja y por versátil que la solución parte de una abstracción que puede ser aplicada para solucionar diferentes problemas. Bajo esta perspectiva se ha buscado desarrollar una plataforma *backend* que nos permita gestionar un repositorio de árboles genealógicos almacenado en una base de datos orientada a grafos, mediante una API rest-ful. Durante el desarrollo de la introducción veremos como la decisión de resolver el problema de "realizar un repositorio de árboles genealógicos en bases de datos orientadas a grafos" y su gestión mediante una API rest-ful nos ayuda a dar una solución eficiente, elegante y versátil.

## 2.2. Contextualización del proyecto

Se empezara introduciendo los diferentes conceptos sobre los que se sustenta el proyecto y explicando su relación con el proyecto:

### 2.2.1. Los árboles genealógicos

Un arbole genealógico, también llamado genorama, es la representación gráfica de los antepasados y descendientes de un individuo. Para su representación se suelen usar tablas o árboles, siendo esta última la forma más común.

#### Uso y aplicación de los árboles genealógicos

Los árboles genealógicos se usan como herramienta en la genealogía, que se encarga de estudiar y seguir la ascendencia y descendencia de una persona o familia. La genealogía es una ciencia auxiliar de la Historia y es trabajada por un genealogista. Uno de los objetivos del software a desarrollar es dar soporte a los genealogistas.

Por otro lado hay varias comunidades de aficionados que llevan sus propios árboles genealógicos, el software creado también les podrá dar servicio a esta tipología de usuarios.

### GEDCOM [2] (GEnealogical Data COMmunication)

Es un formato de archivo de datos, proporciona un formato flexible y uniforme para el intercambio de datos genealógicos computarizados. El modelo de datos que usa GEDCOM esta basado en los núcleos familiares y individuos. En el siguiente apartado donde abalaremos de las tecnologías usadas para desarrollar el proyecto entramos en detalle de las cuestiones técnicas del formato, profundizando en su modelo de datos y su estructura.

### 2.2.2. Bases de datos orientadas a grafos (BDOG)

Las bases de datos orientadas a grafos se investigan desde hacer mucho, su poca popularidad es debido a la imposición generalizada del modelo relacional, este fenómeno pudo ser producido por la simplicidad y fácil manejo del modelo relacional.

Con la aparición de las nuevas metodologías ágiles y el requerimiento de modelos que permitan mayor flexibilidad en el desarrollo (e.g. SCURM) y las exigencias de velocidad de nuevas plataformas con modelos donde los datos están altamente asociativos y las consultas requieren usar gran cantidad de estas asociaciones o relaciones (e.g. redes sociales) las BDOG han ganado popularidad. Los motivos por los que estas bases de datos satisfacen los requerimientos mencionados se explican en los siguientes apartados. A continuación se explica las entidades que conforman una BDOG.



**Estructura .**

Una DBOG representa la información como nodos de un grafo y sus relaciones mediante sus aristas. Lo que resulta realmente diferencial del modelo relacional es que en este modelo las relaciones son ciudadanos de primer orden de la estructura del grafo (i.e. A diferencia de una base de datos relacional, donde las claves foráneas que relacionan los objetos del modelo son atributos de estos.) [6], lo que repercute en que las consultas con un alto nivel de acoplamiento sean más rápidas, los motivos de este fenómeno se explicaran más adelante.

**Propiedades .**

Hay dos propiedades principales de las BDOG que tenemos que considerar a la hora de entender sus capacidades:

**Forma de almacenado subyacente .**

Solo algunas BDOG almacenan la información de forma nativa en una estructura de grafo, otras por lo contrario serializan los datos a un modelo relacional o algún otro modelo de almacenamiento de datos de propósito general.

**Motor de procesamiento .**

Una propiedad de las BDOG es la adyacencia libre de índices, esto quiere decir que los nodos están conectados físicamente entre ellos en la base de datos. Esto mejora la eficiencia en las consultas que han de recorrer nuestro modelo, lo que provocaría que en una base de datos relacional tuviésemos que hacer una gran cantidad de *joins*. La contrapartida de la adyacencia libre de índice es que aumenta el coste de las consultas que no requieran recorrer nuestro modelo. Existen técnicas en el diseño de nuestros modelos que nos permiten esquivar esta contra-prestación convirtiendo atributos de los nodos en nuevos nodos y simulando índices sobre estos atributos [7].

**Expresividad y simplicidad .**

La propiedad de las BDOG donde las relaciones son ciudadanos de primer orden, a diferencia de otras bases de datos como las SQL que usan el modelo relacional donde las relaciones se definen mediante claves foráneas entre los datos o otras aproximaciones como las bases de datos basadas en clave-valor donde las relaciones tampoco son ciudadanos de primer orden, nos permite definir modelos simples y expresivos sobre nuestro dominio.

La figura 1 nos da una visión de algunas de las tecnologías de BDOG más importantes que podemos encontrar hoy en día, el valor de *Graph Processing* se mide según si la tecnología hace uso de la adyacencia libre de índices en su motor de procesamiento y el *Graph Storage* se evalúa como nativo o no nativo si la base de datos usa serializadores para acabar almacenando los datos en una estructura no grafoide (i.e SQL o clave-valor).

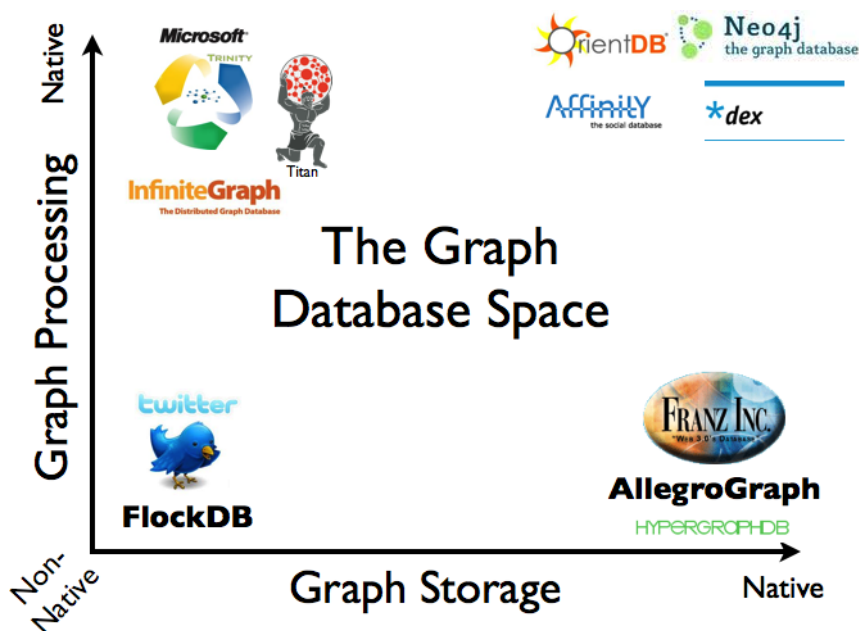


Figura 1: Visión del conjunto de tecnologías

### 2.2.3. REST-ful (REpresentational State Transfer)

REST nace como un estilo de arquitectura del software que consiste en coordinar un conjunto de restricciones arquitectónicas a los componentes, conectores y recursos, en un sistema distribuido de *hypermedia* (e.g. imágenes, audio, vídeo, texto plano y hiperenlaces). Estas restricciones se efectúan con el objetivo de mejorar la escalabilidad, simplicidad, modificabilidad, visibilidad, portabilidad, rendimiento y confiabilidad.

Los sistemas que se adecuan a una arquitectura rest son llamados rest-ful. Estos sistemas se comunican a través de HTTP con el uso de los verbos que esté implementa, normalmente los verbos GET, POST, PUT, DELETE suelen estar asimilados a las operaciones CRUD de una base de datos. Las interfaces que usan estos sistemas son llamados *endpoints* (i.e URIS). En la actualidad existen varios *frameworks* orientados al desarrollo de APIs REST, más adelante haremos un estudio de algunas de las diferentes tecnologías que podemos encontrar.

## 2.3. Perspectiva general del software genealógico actual.

Todo software genealógico, como mínimo permite almacenar la siguiente información de un individuo: datos personales (nombre, apellidos, etc), información de los eventos vitales de la misma (fechas y lugares de estos eventos) y las relaciones (familiares) entre personas. Contra más flexible es el programa más información te permite introducir acerca de un individuo. También proporcionan diferentes maneras de representar la información y permiten exportar a GEDCOM

la información representada y importar archivos GEDCOM.

La mayor parte del software genealógico actual esta basado en soluciones de escritorio, pero en los últimos años han proliferado diferentes soluciones web como myheritage o familysearch, que no solo sirven como plataforma de edición sino que también son grandes plataformas *cloud* en las que se almacenan los árboles genealógicos.

Las soluciones más avanzadas, aparte de la gestión de árboles también ofrecen herramientas más orientadas a la investigación, como podrían ser sistemas de búsqueda de individuos basados en sus relaciones o herramientas estadísticas.

## 2.4. Análisis previo de tecnologías

### 2.4.1. Base de datos

Para la selección de las bases de datos han valorado las siguientes bases de datos orientadas a grafos, estas bases de datos han estado seleccionadas bajo criterios subjetivos buscando que aportasen rasgos diferenciales entre ellas, pero a la vez se pueda acceder a documentación o soporte para problemas específicos un una cierta facilidad:

**HyperGraphDB** (<http://hypergraphdb.org/>) :

Es un base de datos orientada a grafos de propósito general, extensible, potable, distribuida y integrable. Diseñada especialmente para proyectos sobre inteligencia artificial y web semántica. También se puede usar como base de datos orientada a objetos.

**InfoGrid** (<http://infogrid.org>) :

Es una base de datos orientada a grafos especialmente enfocada al desarrollo de aplicaciones *backend* REST-ful. Hace especial énfasis en la orientación a objetos. De este proyecto destaca su modularidad.

**Neo4j** (<http://neo4j.com/>) :

Esta base de datos basada en grafos esta diseñada como una base de datos de propósito general, cumple los requisitos de las dos anteriores a pesar que no se pueden crear *hyperedges* como tal a diferencia de HyperGraphDB, por otro lado se pueden usar técnicas en la representación de los grafos que los simulan. También aporta mejoras en el almacenamiento de los datos respecto a InfoGrid, ya que el almacenamiento esta implementado en grafos de forma nativa.

#### **Decisión:**

Finalmente se ha escogido Neo4j las principales razones son:

- Neo4j almacena los datos en un grafo de forma nativa.
- El motor de procesamiento es orientado a grafos.

- Es una de las bases de datos orientadas a grafos más usadas por ello con más comunidad. (fuente DB Engines Ranking)
- A pesar de usar un lenguaje específico, Cypher es muy intuitivo ya que usa una representación grafoide para realizar las consultas.

#### 2.4.2. Lenguaje de programación

Los lenguajes escogidos para el análisis han estado seleccionados con criterios subjetivos, seleccionándolos de tal forma que aporten alternativas a la hora tanto de dar agilidad a la implementación como a ser eficientes en el rendimiento. También se ha tendido en cuenta los *frameworks* que se pueden usar en estos lenguajes para la consecución de una API rest.

Para desarrollar este proyecto se ha optado entre los siguientes lenguajes de programación.

**Scala** ([www.scala-lang.org/](http://www.scala-lang.org/)) :

Scala es un lenguaje de propósito general multi-paradigma, da soporte completo para programación funcional y orientada a objetos, también dispone de un sistema de inferencia de tipo estático muy potente. Se compila a Java byte code y se ejecuta sobre la máquina virtual de Java. Es totalmente interoperable con Java por lo que se pueden usar librerías Java y viceversa.

Frameworks:

**Scalatra**([www.scalatra.org/](http://www.scalatra.org/)) :

Es un *Micro-framework* para scala, consta de librerías para la creación de una API rest completa, como *micro-framework* no fuerza ningún tipo de arquitectura. Se usa para desarrollar bajo la siguiente filosofía:

**Empezar desde lo pequeño, desarrolla hacia arriba :**

Empieza construyendo un pequeño núcleo y integra módulos simples para solucionar las tareas.

**Libertad :**

Permite al programador usar la estructura y las librerías que crea convenientes para su aplicación.

**Sólido pero no terco :**

Usa componentes sólidos para la creación de la API, por ejemplo, los serverlets no son una solución muy moderna a la gestión de la comunicación cliente servidor, pero tienen una gran comunidad detrás y son extremadamente estables. Por otro lado se anima a usar la flexibilidad del framework para usar nuevas técnicas y aproximaciones en diferentes ámbitos del servicio.

**Ama HTTP :**

HTTP es un protocolo de naturaleza carente de estado, no se ha de pretender usar mecanismos complejos que den la apariencia de estado.

**Lift(liftweb.net/) :**

Es un *framework* para scala, esta diseñado para seguir la arquitectura 'View First'. Clama ser uno de los *frameworks* más seguros y destaca por los siguientes puntos:

**Seguridad :**

Lift es resistente a casi todas las vulnerabilidades comunes incluyendo la mayoría de la lista "OWASP *top 10* de".

**Amigable para el desarrollador :**

Lisft esta diseñado para facilitar la creación de aplicaciones de forma rápida y sencilla.

**Escalable :**

Esta diseñado para dar soporte a grandes cantidades de trafico.

**Modular :**

El programa se compone de módulos, esto da mucha agilidad a la hora ampliar las funcionalidades de los proyectos y da una arquitectura muy limpia y fácil de mantener.

**Python(python.org/) :**

Python es un lenguaje de propósito general, interpretad, y de tipado dinámico. Es un lenguaje multi-paradigma, combina la orientación a objetos con la programación funcional. A diferencia de scala la parte funcional no esta totalmente desarrollada ya que no permite valores inmutables o técnicas como curring. Python esta diseñado para agilizar el proceso de desarrollo y facilitar la legibilidad y comprensión del código.

Frameworks:

**Falsk(flask.pocoo.org/) :**

Es un *micro-framework*, esta diseñado permitir el desarrollo de aplicaciones web de forma flexible. Consta de los siguientes herramientas:

- Incorpora servidor y debugger
- Integra soporte para unit test
- Gestor de peticiones RESTful
- Usa Jinja2 templating
- Soporte para "Cookies" con seguridad
- Cumple al 100 % el WSGI 1.0 (Estándar para la comunicación de las aplicaciones con el servidor)

- Basado en unicode
- Extensamente documentado

**Django**([www.djangoproject.com/](http://www.djangoproject.com/)) :

Es un *framework*, esta diseñado permitir el desarrollo de aplicaciones rápidamente y con una estructura limpia. Consta de diferentes capas que aportan seguridad la aplicación. Esta basado en la arquitectura modelo vista controlador. Entre sus virtudes destacan la gran facilidad para modular las aplicaciones y facilitar su integración y una comunidad muy extensa, también dispone de una documentación clara, bien estructurada y muy completa. Django destaca en su web estos tres puntos:

- Ridiculaemntne rapido de desarrollar
- Altamente seguro
- Escalabilidad asombrosa

**Decisión:** Finalmente se ha escogido Django. Los motivos han sido:

**Velocidad de desarrollo a coste de velocidad de ejecución** : Python es conocido por la rapidez que aporta al desarrollar, dado que es un lenguaje de muy alto nivel con multitud de librerías, por otro lado el usar Django nos aporta gran parte de las librerías necesarias para desarrollar una aplicación de servidor que de servicio REST. Por contra esta decisión repercute en que la velocidad de ejecución de escala en comparación con python es mucho mayor, se asumirá esta desventaja ya que la finalidad del proyecto no es llevar el producto a un entorno de producción real donde la velocidad de ejecución sea capital.

A destacar:

**Django Rest Framework** :

Framework que se integra con Django y proporciona herramientas para facilitar la serialización, enrutamiento teniendo en cuenta el protocolo REST-ful, autorización y autenticación, creación de vistas para las operaciones CRUD, etc.

**Django OAuth Toolkit** :

Conjunto de herramientas para crear autenticación y autorización con permisos, basado en el protocolo OAuth2.

**Celerys** :

Librería de python que permite combinado con una tecnología “*message broker*” permite crear tareas que realicen funcionalidades de la aplicación o crear tareas asíncronas.

**neomodel OGM(Object graph mapper)** :

Permite crear modelos en la base de datos neo4j basándose en los modelos de la aplicación.

## 2.5. Integración de leyes y regulaciones

Al desarrollar una aplicación *backend* donde se almacenaran datos de usuarios y personas se tendrá que respetar la **Ley Orgánica 15/1999 de Protección de Datos de Carácter Personal (Lopd)**. Por ello se tendrá que tener en cuenta:

### 2.5.1. Derecho de información

El derecho de información esta regulado en la **Ley Orgánica 15/1999 de Protección de Datos de Carácter Personal (Lopd)**. Regula las condiciones en que se ha de recoger, tratar y ceder los datos personales, su fin es proteger la intimidad y demás derechos ciudadanos. Para adecuarse a esta ley, si un usuario introduce datos personales de otra persona se tendrá que se notificar de la posesión de estos datos al afectado y pedir su aprobación para su almacenamiento.

### 2.5.2. Consentimiento del afectado

EL artículo 6 de la Lopd, regula que para tratar datos personales se ha de tener el consentimiento expreso del damnificado, por ello al registrarse los usuarios tendrán que aceptar una licencia de acuerdo con el usuario final (EULA). Por el momento al ser una plataforma en fase beta donde no se registraran los usuarios no se redactara dicho acuerdo ni se realizara ninguna confirmación a la hora del registro.





### 3. Alcance

#### 3.1. Objetivos

Este proyecto busca desarrollar un sistema desde el que se pueda gestionar un repositorio de árboles genealógicos y los usuarios del mismo sistema. El sistema tendrá que ser accesible a través de una API rest que permita a aplicaciones de terceros usar los datos del repositorio. Se ha establecido que la persistencia de los árboles genealógicos tendrá que realizarse sobre una base de datos orientada a grafos, esta decisión se ha tomado para poder probar como una base de datos con estas características nos permite modelar un árbol genealógico aprovechando sus propiedades.

El modelo diseñado para la base de datos orientada a grafos para el almacenamiento de árboles genealógicos deberá ser flexible (i.e. orientado a poder incluir nuevos atributos o conceptos manteniendo la consistencia del modelo) almacenar información genealógica y que este orientada a maximizar la eficiencia de las consultas más complejas que necesite ejecutar el sistema.

El objetivo de la API rest es cubrir todas las operaciones CRUD del repositorio y añadir funcionalidades extra sobre el sistema, las 2 funcionalidades estudiadas son la carga de archivos GEDCOM al repositorio y la búsqueda de personas similares entre los diferentes árboles almacenados.

La gestión de usuarios que realice la API rest se tendrá que realizar de forma segura para que los datos almacenados por los usuarios estén protegidos.

#### 3.2. Objetivos académicos

Bajo los términos en los que se inscribió el proyecto los objetivos que este tendrá que alcanzar serán:

**CES1.1 :**

Desarrollar, mantener y evaluar sistemas y servicios software complejos o críticos.

**CES1.2 :**

Dar solución a problemas de integración en función de las estrategias, de los estándares y tecnologías disponibles.

**CES1.4 :**

Desarrollar, mantener y evaluar servicios y aplicaciones distribuidas con soporte de red.

**CES1.5 :**

Especificar, diseñar y implementar bases de datos.

**CES1.6 :**

Administrar bases de datos (CIS4.3)

**CES1.9 :**

Demostrar comprensión en la gestión y gobierno de sistemas software.

**CES2.2 :**

Diseñar soluciones apropiadas en uno o más dominós de la aplicación, usando métodos de ingeniería del software que integren aspectos éticos, sociales, legales y económicos.

En las conclusiones de explicara como se han cumplido estos objetivos de forma detallada.

### 3.3. Metodologia y rigor

Las metodologías usadas son las siguientes:

#### 3.3.1. SCRUM

Para el desarrollo del proyecto se ha escogido como metodología de trabajo SCRUM, los *sprints* dado la naturaleza del proyecto se han adaptado esta metodología ágil para ser usada por una sola persona. La metodología de trabajo se ha definido de la siguiente forma:

**Product backlog :**

Para crear el *product backlog* se han definido un seguido de historias de usuario, de acuerdo con el tutor del proyecto, que asume el rol de *product owner*. Estas historias de usuario se revisaran constantemente en los *sprint backlogs*.

**Sprints :**

Se desarrollaran a lo largo de dos semanas, en este tiempo se desarrollaran las historias de usuario escogidas.

**Sprint backlog :**

Se realizaran al final de cada *sprint*, dado que esta tarea estará limitada por la disponibilidad del tutor y el alumno, se dara flexibilidad pudiendo realizar el *sprint backlog* de varios *sprints* en una reunión. Durante estas reuniones se revisaran las historias de usuario, reorganizando su prioridad, modificando su contenido, añadiendo nuevas o eliminando historias que no se crean necesarias.

**Product owner :**

El *product owner* es el responsable de de mantener el *product backlog*, entre sus tareas asegurar el valor del trabajo realizado por el equipo de desarrollo, este rol lo asumirá el director del proyecto.

**Scrum master :**

Es la persona responsable de asegurarse que la metodologia se sigue y es usada correctamente. Este rol se asumirá por el estudiante.

**Equipo de desarrollo :**

Se encarga de desarrollar las funcionalidades del producto durante los *sprints*, para este proyecto no se usara un equipo, el único desarrollador sera el estudiante.

En la figura 2 se puede ver un esquema de las fases de la metodología.

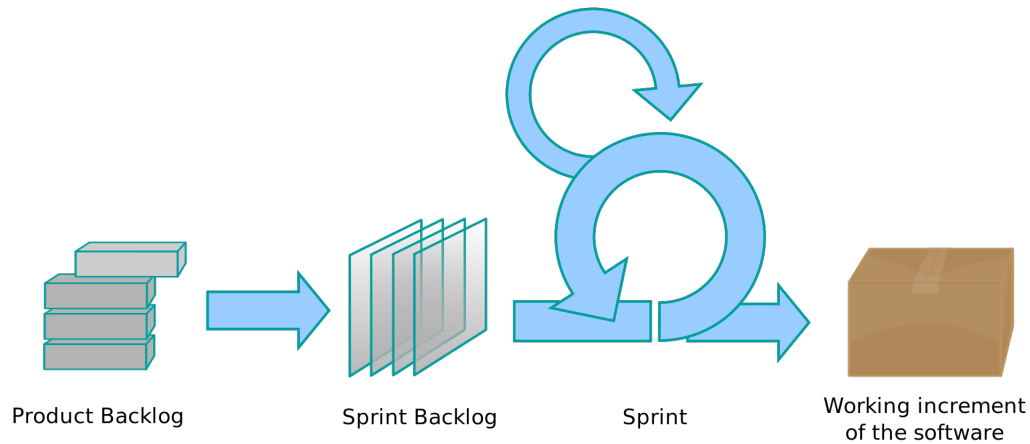


Figura 2: Metodología SCRUM

Finalmente si bien no podemos decir que la metodología usada sea SCRUM, más bien una adaptación de esta metodología a las necesidades del proyecto, para simplificar en el trabajo se hablara de esta adaptación como SCRUM, si bien sabiendo que es una adaptación de este.

### 3.4. Metodo de validación

Al seguir la metodología Scrum, no se necesita definir criterios de aceptación de todo el proyecto entero, puesto que la propia dinámica ofrece un fuerte control del producto por parte del “product owner”. Uno de los mejores aspectos de esta metodología es la gran cohesión que crea entre todos los miembros involucrados al proyecto. En general siempre hay una misma visión del producto que se está creando, y el “product owner” puede ir perfilando el producto durante el transcurso de los sprints. No se definen criterios de aceptación de todo el producto, pero si que se definen criterios de aceptación de las diferentes tareas. Gracias a que las tareas tienen una granularidad muy pequeña, los criterios de aceptación pueden ser muy precisos y concisos. Cómo se ha visto anteriormente, los encargados de validar los criterios de aceptación de las tareas es el *product owner*, durante los *sprint backlogs*.

### 3.5. Fases de desarrollo

El desarrollo del sistema constara de cuatro fases, que se realizaran de manera secuencial de la siguiente manera.

#### 3.5.1. Análisis de requisitos y estudio previo

Primero, durante el transcurso de GEP se realizara un estudio previo donde se definirá el estado del arte, alcance, planificación, se explicara la metodologia que se seguirá, también se realizara también una estimación de costes.

Después, En la primera etapa del proyecto se realizara el análisis de requisitos, mediante este análisis se describirán tanto los requisitos funcionales como no funcionales que tendrá que cubrir la solución que se desarrolle, por supuesto estos requisitos se revisaran y priorizaran según se vayan desarrollando *sprint*.

#### 3.5.2. Especificación

Una vez realizado el análisis de requisitos se definirá la especificación del proyecto. Con la especificación se definirán los casos de uso y un diagrama UML con nuestro modelo conceptual de datos.

#### 3.5.3. Diseño

En la fase de diseño decidiremos que patrones aplicaremos en nuestro proyecto y estudiaremos como las tecnologías que se ha decidido usar ayudan a seguir ciertas arquitecturas. También se estudiaran los protocolos que se vayan a usar.

Por ultimo se definirá el diseño del modelo de datos establecido a partir de diseño del modelo

conceptual de datos y se decidirán diferentes técnicas de optimización. También se estudiara el diseño de capa de domino y se definirán los diagramas de secuencia más importantes.

#### 3.5.4. Implementación y pruebas

Durante la implementación se realizara el desarrollo del sistema, teniendo en cuenta las decisiones que se hayan tomado durante las etapas anteriores. A medida que se vaya avanzando en la implementación se diseñaran test de integración par ir validando el desarrollo.

### 3.6. Posibles obstáculos

**Tiempo:** La gestión del tiempo sera uno de los obstáculos más notables a la hora de desarrollar el proyecto, dado que se pretende plantear un proyecto que no necesariamente ha de concluir su desarrollo con la entrega final, para ello se tendrán que plantear claramente las iteraciones necesarias para tener una versión funcional del software y el tiempo de desarrollo que requerirán.

**Integración de tecnologías y desarrollo:** Uno de los objetivos de este proyecto es combinar ciertas tecnologías con las que no se ha trabajado anteriormente, esto puede repercutir en que se encuentren dificultades al integrarlas. Por ello este hecho se tendrá que tener en cuenta en la planificación temporal.



## 4. Planificación

Como se indica en el alcance del proyecto, el trabajo consta de cuatro etapas, análisis de requisitos y estudio previo, especificación, diseño e implementación y pruebas. Si tenemos en cuenta la fase de GEP, donde se realizó el análisis de requisitos y el estudio previo, que va del 14 de noviembre al 14 de septiembre de 2015, en la que se han estimado 20 horas de trabajo semanales lo que da un total de 80 horas. La fase de especificación y diseño, que va del 15 de octubre al 14 de noviembre de 2015, en la que se ha estimado de la misma forma 20 horas de trabajo semanal, con lo que se estiman también 80 horas de trabajo. En la fase de implementación donde se empieza a aplicar la metodología ágil se ha dividido en 11 sprints de dos semanas, a cada sprint se ha calculado un trabajo de 30 horas, lo que da un total de 330 horas. En total se calcula que el trabajo tendrá un total de 490 horas de trabajo.

### 4.1. Gantt

Dado que en el proyecto se desarrolla íntegramente por una sola persona, la secuencialidad del diagrama es absoluta.

Durante la fase de desarrollo simplemente se especifican los *Sprints* que se llevarán a cabo hasta conseguir la consecución del proyecto, a medida que las historias de usuario se vayan redactando y priorizando se irán asignando a los *sprints*.

## 4.1.1. Tareas y Gráficos gantt

Name	Begin date	End date
GEP	9/14/15	10/14/15
Estado del arte y alcance	9/14/15	9/22/15
Planificación temporal	9/23/15	9/28/15
Gestión económica y sostenibilidad	9/29/15	10/5/15
Preparación de la presentación final de GEP	10/6/15	10/14/15
Especificación	10/15/15	11/14/15
Redacción de las historias de usuario	10/15/15	10/23/15
Requisitos no funcionales	10/24/15	10/28/15
Especificación de la arquitectura	10/29/15	11/8/15
Análisis de la selección de tecnologías	11/9/15	11/14/15
Implementación	11/15/15	4/16/16
Sprint 1 (Configuración del entorno de pre y pro)	11/15/15	11/28/15
Sprint 2 (Configuración del entorno de pre y pro)	11/29/15	12/12/15
Sprint 3 (Persistencia)	12/13/15	12/26/15
Sprint 4 (Persistencia)	12/27/15	1/9/16
Sprint 5 (API - CRUD)	1/10/16	1/23/16
Sprint 6 (AP - Usuarios)	1/24/16	2/6/16
Sprint 7 (API - End-points ext)	2/7/16	2/20/16
Sprint 8 (API - Async)	2/21/16	3/5/16
Sprint 9 (API - Similitudes)	3/6/16	3/19/16
Sprint 10 (API - Similitudes)	3/20/16	4/2/16
Sprint 11 (Revisión)	4/3/16	4/16/16
Revisión documentación	4/17/16	6/30/16

Figura 3: Tareas del diagrama de gantt.



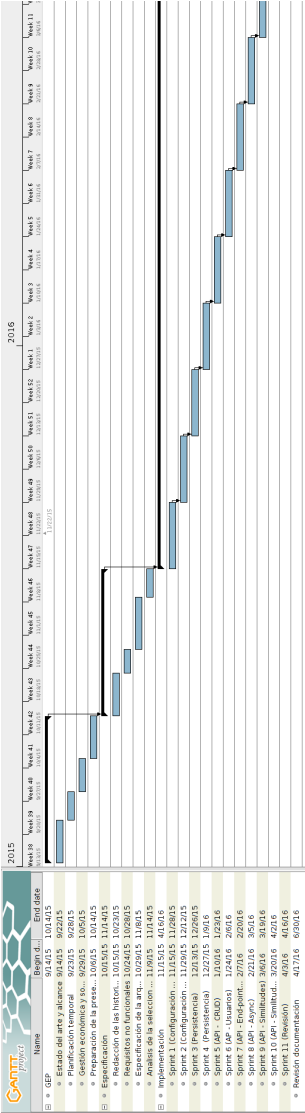


Figura 4: Diagrama de gantt.

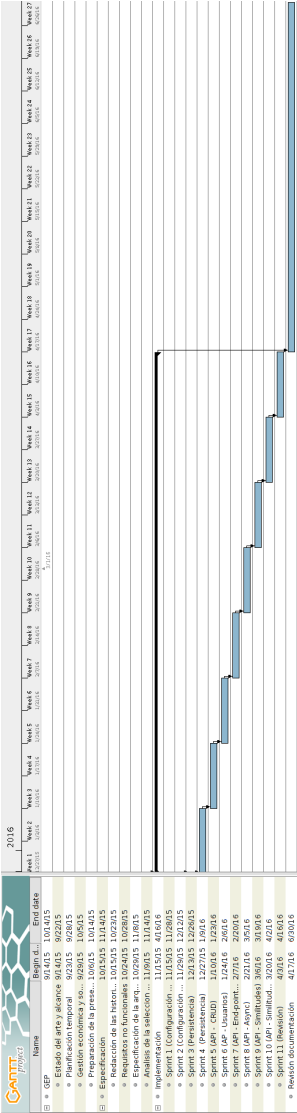


Figura 5: Diagrama de gantt.

## 4.2. Recursos

Las necesidades en recursos serán muy escasas dado que todo el software que se usara es libre y las maquinas en las que se trabaja en pre-producción serán virtuales. La única necesidad de tener unos recursos que se necesiten planificar, ya que requieren ser contratados, son los servidores en los que se hará el *deploy*, se tendrán que contratar durante las semanas que duren los *sprints* en los que se haga el *deploy*. Por otro lado en cuanto a recursos humanos el proyecto se llevara a cabo por un solo desarrollador que se encargará tanto de la programación como la documentación.

## 4.3. Alternativas y plan de acción

Dado que el proyecto se desarrollara usando *Scrum*, al final de cada *sprint*, durante el *sprint backlog* se determinara si se han cumplido las expectativas del *sprint*. Por otro lado en el *sprint planning* se tendrá en cuenta si han habido carencias o *bugs* en los anteriores *sprints* para incorporarlos como historias de usuario al siguiente *sprint*.

Como se ha comentado en el alcance, se diseñara un software abierto orientado al desarrollo continuado. Por lo que en el *sprint* 11 se dará por concluido el proyecto, y se documentara el estado en el que se encuentre y las historias de usuario pendientes, así como las desviaciones de la planificación inicial.

## 5. Presupuesto y impacto social

La siguiente planificación económica tiene en cuenta los siguientes puntos:

- El proyecto no está orientado a dar rentabilidad económica.
- Todas las tareas se asignarán a la misma persona, el sueldo se supondrá según el rol que asuma en cada momento.

### 5.1. Identificación y estimación de costes

#### 5.1.1. Costes de personal

A pesar que el proyecto solo será desarrollado por una única persona, durante el transcurso del proyecto asumirá diferentes roles. Los costes de personal se estimarán calculando la carga de trabajo que tendrá que asumir cada rol, se establecerá el coste por hora a cada rol.

Ateniéndonos a la planificación temporal:

- La parte que corresponde a planificación, que va del día 09/14/2015 al 11/14/15, será asignada al jefe del proyecto.
- La primera fase de desarrollo, donde se configura el entorno de trabajo, que se comprende del 11/15/15 al 12/12/15, se asignará al de técnico de sistemas.



- Durante las siguientes fases del desarrollo, del 12/12/15 al 08/14/16, la carga de trabajo se distribuirá entre todos los roles. Estas fases se comprenden entre el 12/12/15 y el 05/14/16.

**Jefe del proyecto** 5 %

**Programador** 70 %

**Sistemas** 10 %

**Tester** 15 %

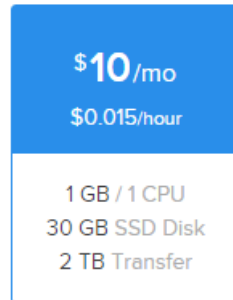
- La ultima fase donde se preparara la presentación del proyecto estará asignada al jefe del proyecto, del 05/15/16 al 06/14/16.

Contando una dedicación media de tres horas al día:

Costes de personal			
ROL	Horas	€/hora	Coste total
Jefe del proyecto	274.8	40	10992
Programador	431.2	30	12936
Tester	92.4	25	2310
Sistemas	169.6	25	4240
			30478

### 5.1.2. Coste del espacio de trabajo

Bajo el supuesto que se desarrollase el proyecto en un área de *coworking*, concretamente coworking fontanella, el coste sería de 180€ al mes, dado que el proyecto dura aproximadamente 9 mes, tal como podemos ver en la planificación temporal, implicaría un coste de 1620€.



### 5.1.3. Coste del los servidores

Durante la fase de *deploy* se alquilaran servidores de digital ocean, el precio es de 10€ al mes, dado que la fase de deploy dura un mes el coste será de 10€.

## 6. Sostenibilidad y compromiso social

Cada una de las tres secciones siguientes están evaluadas sobre 10.

### 6.1. Dimensión económica - 7

El proyecto a desarrollar no busca la rentabilidad económica, por otro lado sus costes de desarrollo son muy bajos, siendo un proyecto muy viable económicamente. Si bien, en el caso de querer comercializar el software, como hemos podido ver en el estado del arte, es difícil que en el tiempo de duración del proyecto con los recursos que se disponen dar una solución del nivel de las soluciones actuales.

### 6.2. Dimensión social - 9

Al ser un software en el que todo su código sera abierto y integrara varias tecnologías que actualmente siguen en desarrollo, podrá ayudar a varias comunidades de desarrolladores. Por otro lado un resultado final del proyecto podría ser interesante para diferentes sectores dedicados a la genealogía que busquen una solución abierta para guardar la información de la que disponen.

### 6.3. Dimensión ambiental - 6

El proyecto tiene un mínimo impacto medio ambiental, no se espera que mediante el uso del software se ahorre ningún recurso material, pero el coste de mantenimiento y creación no implica consumir recursos materiales.





## 7. Análisis de requisitos

### 7.1. Visión global del sistema

Como ya se ha comentado en los capítulos anteriores, el proyecto consiste en el desarrollo de API rest-ful desde la que poder gestionar un repositorio de árboles genealógicos.

Para ello el *backend* implementado deberá permitir que diferentes aplicaciones se den de alta para poder acceder a los datos almacenados por los usuarios y así mismo permitir a estos usuarios realizar las operaciones necesarias para gestionar sus árboles genealógicos.

### 7.2. Actores

Los actores implicados en el uso del sistema son dos:

#### Usuario final :

Este actor sera el que se dará de alta en nuestra aplicación y ha de ser capaz de almacenar y gestionar sus árboles genealógicos.

#### Aplicación :

El *backend* ha de permitir a diferentes aplicaciones (*fontends*) darse de alta. Estas aplicaciones recibirán las credenciales necesarias para poder conseguir los permisos de acceso de nuestros usuarios finales y manipular sus árboles.

### 7.3. Requisitos funcionales

Los requisitos funcionales describen todas las funcionalidades que proporcionara el *backend*. A continuación se listan todos los requisitos funcionales desglosados por actor:

#### 7.3.1. Usuario final

##### Gestión de cuenta :

**Registro** : El usuario deberá poderse registrar en el sistema introduciendo los datos necesarios.

**Modificar sus datos** : El usuario deberá poder modificar sus datos.

**Login** : El usuario deberá de poder acceder al una interfaz desde la que gestionar sus datos de usuario.

**Activación de cuenta** : El usuario una vez registrado tendrá que poder activar la cuenta vía email.

### 7.3.2. Aplicación

#### Gestión de cuenta :

**Registro** : El usuario deberá poderse registrar en la aplicación introduciendo los datos necesarios.

**Login** : El usuario deberá de poder acceder al una interfaz desde la que gestionar sus datos de usuario.

**Modificar sus datos** : El usuario deberá poder modificar sus datos.

#### Gestión de aplicaciones :

**Dar de alta aplicación** : El usuario ha de poder dar de alta aplicaciones que posteriormente accederán al contenido del repositorio mediante la API.

**Conseguir token** : Las aplicaciones una vez dadas de alta podrán pedir *tokens* asignados a los usuarios finales con su nivel autorización y que permita su autenticación.

#### Gestión de los árboles :

**Creación, modificación y consulta de árboles genealógicos** : El usuario deberá poder crear, modificar y consultar los árboles genealógicos.

**Creación, modificación y consulta de personas** : El usuario deberá de poder crear, modificar y consultar las personas almacenadas en los árboles genealógicos.

**Creación, modificación y consulta de eventos** : El usuario deberá de poder crear, modificar y consultar los eventos de las personas almacenadas en los árboles genealógicos.

**Subir un archivo GEDCOM** : El usuario deberá de poder cargar archivos GEDCOM en su cuenta.

**Encontrar personas similares** : El usuario deberá de poder seleccionar una persona de uno de sus árboles genealógicos y buscar personas similares en otros árboles de otros usuarios.

## 7.4. Requisitos no funcionales

A continuación se muestra una lista de los requisitos no funcionales del sistema:

1. La API ha de permitir un uso completo de todas las funcionalidades independientemente de la tecnología que la use.
2. Se ha de proporcionar una interfaz gráfica para gestionar los usuarios de la aplicación.I.

3. El tiempo de respuesta de todos los endpoints ha de ser similar sea cuan sea la funcionalidad que implementen.
4. La plataforma ha de constar de sistemas que aseguren la seguridad de los datos de usuario.
5. Todos los *endpoints* de la API tendrán que respetar los estandartes de una aplicación rest-ful



## 8. Especificación

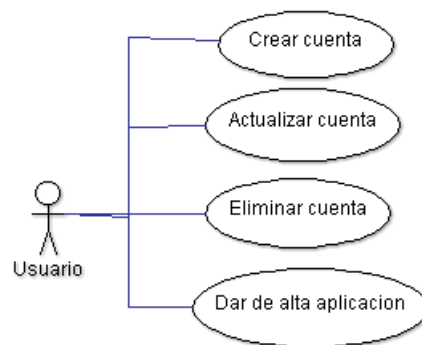
En la especificación se concretarán todos los artefactos, donde se desglosará conceptualmente las características, elementos y servicios necesarios para desarrollar el proyecto.

### 8.1. Casos de uso

Los casos de uso explican los pasos o actividades que los actores tendrán que realizar para llevar a cabo algún proceso, los actores son los que hemos visto en el análisis de requisitos. A continuación se desglosan los casos de uso divididos por actor.

#### 8.1.1. Usuario final

El siguiente diagrama explica de forma visual las interacciones del usuario final con el sistema, a continuación se encuentran las tablas que detallan cada caso de uso.



---

**Crear cuenta**

Actor	Usuario final
-------	---------------

---

Precondición	-
--------------	---

Postcondición	El usuario dispone de una cuenta con la que acceder al sistema.
---------------	---

Escenario principal	<ol style="list-style-type: none"> <li>1. El usuario introduce los datos requeridos por el sistema y los confirma.</li> <li>2. El sistema da de alta al usuario</li> </ol>
---------------------	--

Escenario alternativo	<ol style="list-style-type: none"> <li>1. El usuario introduce los datos requeridos por el sistema y los confirma.</li> <li>2. El sistema avisa al usuario que los datos introducidos no son correctos.</li> <li>3. Se vuelve a ejecutar el caso de uso crear cuenta.</li> </ol>
-----------------------	--

---

**Actualizar cuenta**

Actor	Usuario final
-------	---------------

---

Precondición	El usuario esta dado de alta en el sistema
--------------	--

Postcondición	El usuario a actualizado sus datos de usuario en el sistema.
---------------	--

Escenario principal	<ol style="list-style-type: none"> <li>1. El usuario introduce los datos requeridos por el sistema y los confirma.</li> <li>2. El sistema actualiza los datos del usuario</li> </ol>
---------------------	--

Escenario alternativo	<ol style="list-style-type: none"> <li>1. El usuario introduce los datos requeridos por el sistema y los confirma.</li> <li>2. El sistema avisa al usuario que los datos introducidos no son correctos.</li> <li>3. Se vuelve a ejecutar el caso de uso actualizar cuenta.</li> </ol>
-----------------------	---

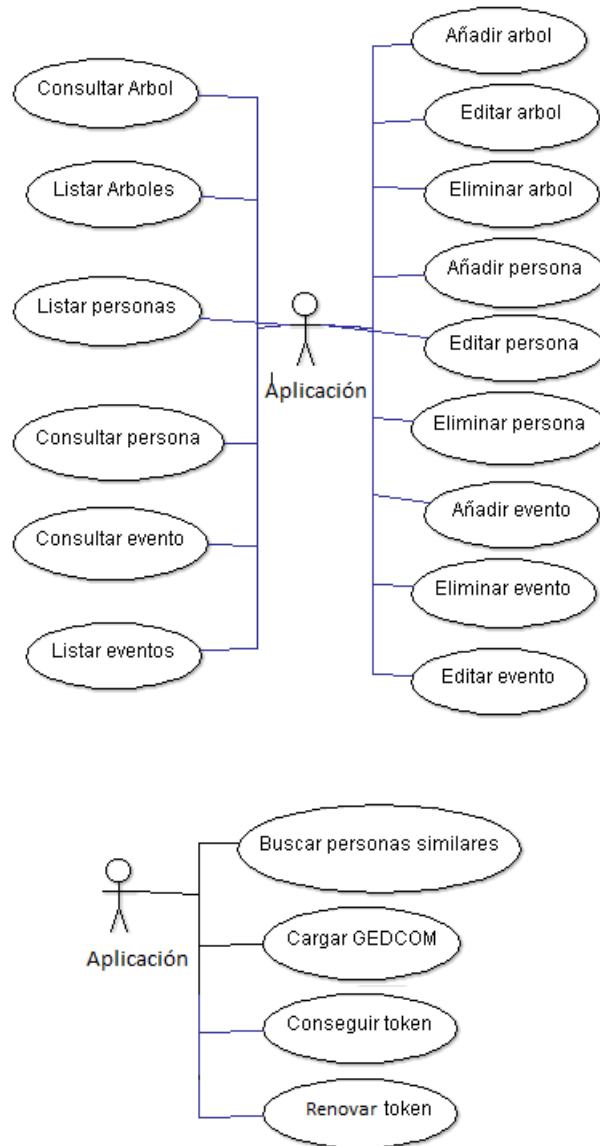
---

**Eliminar cuenta**

Actor	Usuario final
Precondición	El usuario esta dado de alta en el sistema
Postcondición	El usuario es dado de baja del sistema.
Escenario principal	<ol style="list-style-type: none"> <li>1. El usuario indica al sistema que se quiere dar de baja.</li> <li>2. El sistema da de baja al usuario del sistema.</li> </ol>

**Dar de alta aplicación**

Actor	Usuario final
Precondición	El usuario esta dado en alta en el sistema
Postcondición	El usuario proporciona los datos necesarios para dar de alta la aplicación.
Escenario principal	<ol style="list-style-type: none"> <li>1. El usuario introduce los datos requeridos por el sistema y los confirma.</li> <li>2. El sistema da de alta la nueva aplicación.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. El usuario introduce los datos requeridos por el sistema y los confirma.</li> <li>2. El sistema avisa al usuario que los datos introducidos no son correctos.</li> <li>3. Se vuelve a ejecutar el caso de uso dar de alta aplicación.</li> </ol>





---

**Conseguir token**

Actor	Aplicación
Precondición	La aplicación esta dada en alta en el sistema
Postcondición	El sistema devuelve un token que valida un usuario contra el sistema.
Escenario principal	<ol style="list-style-type: none"><li>1. La aplicación pide un token al sistema dando sus credenciales y las de un usuario.</li><li>2. El sistema retorna a la aplicación un token que contiene la información para validar un usuario a través suyo.</li></ol>
Escenario alternativo	<ol style="list-style-type: none"><li>1. La aplicación pide un token al sistema dando sus credenciales y las de un usuario.</li><li>2. Retorna un mensaje de error indicando que credenciales no son correctos.</li><li>3. Se vuelve a ejecutar el caso de uso conseguir token.</li></ol>

---

---

**Renovar token**

Actor	Aplicación
Precondición	La aplicación esta dada en alta en el sistema
Postcondición	El sistema devuelve un token que valida un usuario contra el sistema.
Escenario principal	<ol style="list-style-type: none"><li>1. La aplicación pide la renovación de un token al sistema dando sus credenciales y las credenciales que permiten renovar un token concreto.</li><li>2. El sistema retorna a la aplicación un token que contiene la información para validar un usuario a través suyo.</li></ol>
Escenario alternativo	<ol style="list-style-type: none"><li>1. La aplicación pide la renovación de un token al sistema dando sus credenciales y las credenciales que permiten renovar un token concreto.</li><li>2. Retorna un mensaje de error indicando que credenciales no son correctos.</li><li>3. Se vuelve a ejecutar el caso de uso renovar token.</li></ol>

---

**Añadir árbol**

Actor	Aplicación
Precondición	La aplicación esta dada en alta en el sistema
Postcondición	El sistema añade el árbol con los datos proporcionados.
Escenario principal	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al servidor con los datos necesarios para dar de alta un árbol y el token de un usuario.</li> <li>2. El sistema confirma la creación del árbol a la aplicación.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para dar de alta un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token no es valido.</li> <li>3. Se va a ejecutar el caso de uso conseguir token.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al servidor con los datos necesarios para dar de alta un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token esta caducado.</li> <li>3. La aplicación ejecuta el caso de uso renovar token.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al servidor con los datos necesarios para dar de alta un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el árbol no es valido.</li> </ol>

**Editar árbol**

Actor	Aplicación
Precondición	La aplicación esta dada en alta en el sistema
Postcondición	El sistema actualiza el árbol con los datos proporcionados.
Escenario principal	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al servidor con los datos necesarios para dar de actualizar los datos de un árbol y el token de un usuario.</li> <li>2. El sistema confirma que los datos del árbol han modificado a la aplicación.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para actualizar los datos de un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token no es valido.</li> <li>3. Se vuelve a ejecutar el caso de uso conseguir token.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para actualizar los datos de un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token esta caducado.</li> <li>3. La aplicación ejecuta el caso de uso renovar token.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para actualizar los datos de un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el árbol no existe.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para actualizar los datos de un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el árbol no es valido.</li> </ol>

**Eliminar árbol**

Actor	Aplicación
Precondición	La aplicación esta dada en alta en el sistema
Postcondición	El sistema elimina el árbol con los datos proporcionados.
Escenario principal	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para eliminar un árbol y el token de un usuario.</li> <li>2. El sistema confirma el árbol ha sido eliminado a la aplicación.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para eliminar un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token no es valido.</li> <li>3. Se vuelve a ejecutar el caso de uso conseguir token.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para eliminar un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token esta caducado.</li> <li>3. La aplicación ejecuta el caso de uso renovar token.</li> </ol>

---

**Consultar árbol**

Actor	Aplicación
Precondición	La aplicación esta dada en alta en el sistema
Postcondición	El sistema retorna la información de árbol consultado.
Escenario principal	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para consultar un árbol y el token de un usuario.</li> <li>2. El sistema retorna los datos del árbol consultado</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para consultar un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token no es valido.</li> <li>3. Se ejecuta el caso de uso conseguir token.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para consultar un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token esta caducado.</li> <li>3. La aplicación ejecuta el caso de uso renovar token.</li> </ol>

---

---

**Cargar árbol**

Actor	Aplicación
Precondición	La aplicación esta dada en alta en el sistema
Postcondición	El sistema almacena el árbol.
Escenario principal	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para cargar un árbol y el token de un usuario.</li> <li>2. El sistema carga todos los datos del árbol enviado por la aplicación.</li> <li>3. El sistema confirma la operación</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para cargar un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token no es valido.</li> <li>3. Se ejecuta el caso de uso conseguir token.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para consultar un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token esta caducado.</li> <li>3. La aplicación ejecuta el caso de uso renovar token.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos necesarios para cargar un árbol y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el los datos del árbol a cargar no son validos.</li> </ol>

---

**Buscar personas similares**

Actor	Aplicación
Precondición	La aplicación esta dada de alta en el sistema
Postcondición	El sistema confirma que se ha aceptado la petición y crea conexiones entre los usuarios similares.
Escenario principal	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos que identifican a una persona y el token de un usuario.</li> <li>2. El sistema crea las conexiones entre los usuarios similares.</li> <li>3. El sistema confirma la operación al usuario</li> <li>4. El sistema confirma por correo electrónico que la operación ha finalizado y el estado.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos que identifican a una persona y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token no es valido.</li> <li>3. Se ejecuta el caso de uso conseguir token.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos que identifican a una persona y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el token esta caducado.</li> <li>3. La aplicación ejecuta el caso de uso renovar token.</li> </ol>
Escenario alternativo	<ol style="list-style-type: none"> <li>1. La aplicación envía una petición al sistema con los datos que identifican a una persona y el token de un usuario.</li> <li>2. Retorna un mensaje de error indicando que el los datos del de la persona no son validos.</li> </ol>

\*Los casos de uso referentes a las operaciones CRUD de los otros objetos del modelo funcionan con la misma lógica que las operaciones CRUD del árbol.

Por lo tanto no se especifican.



## 8.2. Modelo conceptual de datos

El modelo conceptual de datos, es la representación de los conceptos significativos en el dominio del problema, es decir, son las principales entidades que forman el sistema a partir de ahora estas entidad las llamaremos clases. El siguiente diagrama explica como se relacionan las clases de nuestro modelo de datos entre ellos en UML (Unified Modeling Language).

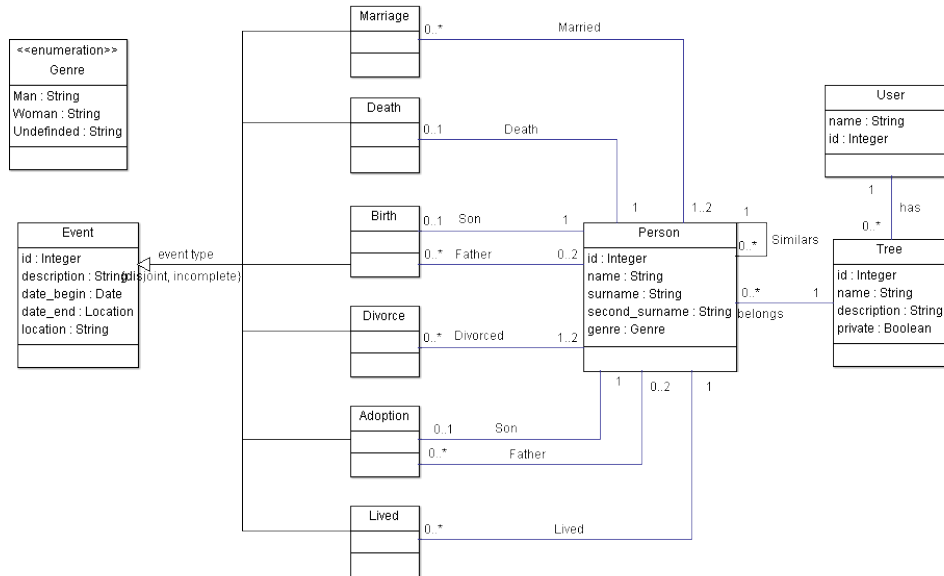


Figura 6: Diagrama del modelo conceptual de datos

A continuación se explica que representa cada clase de nuestro modelo.

**User** : La clase *User* constan de atributos descriptivos y tienen asociados los árboles genealógicos que han creado. Esta clase representa los usuarios de nuestro sistema.

**Tree** : La clase *Tree* consta de los atributos descriptivos de un árbol genealógico y tiene asociadas las personas que pertenecen a este árbol.

**Person** : Esta clase contiene la información que describe a un individuo, tiene asociados diferentes *Events* que represan eventos de su vida. También contiene relaciones con personas del sistema que potencialmente pueden ser la misma persona.

**Event** : La clase *Event* es una clase polimórfica y contiene todos los atributos que describen a cualquier evento independientemente del tipo, se ha diseñado de tal forma que permita la incorporación de nuevos tipos de eventos, con el fin de poder ampliar las funcionalidades

en un futuro. Una característica que vale mencionar es que la fecha de un evento se dará mediante un intervalo, permitiendo establecer fechas indeterminadas (i.e. podemos establecer que un evento fue a partir de una fecha o que como muy tarde sucedió en una fecha), para ello hay dos atributos `date_begin` y `date_end`, en el caso que sepamos la fecha exacta los dos tendrán el mismo valor, en el caso que sepamos que fue a partir de una fecha solo definiremos `date_begin` y por ultimo en caso que sepamos que como muy tarde sucedio en una fecha solo estableceremos `date_end`.

Los tipos de evento que se tendrán en cuenta para el desarrollo este proyecto son:

**Birth** : Esta clase representa un nacimiento, las relaciones contienen los padres y el hijo.

**Lived** : Esta clase representa donde ha estado viviendo una persona.

**Death** : Esta clase representa la defunción de una persona.

**Marriage** : Esta clase representa un matrimonio entre dos personas.

**Divorce** : Esta clase representa un divorcio entre dos personas.

**Adoption** : Esta clase representa una adopción, las relaciones contienen los padres y el hijo.

### 8.2.1. Restricciones de integridad

Dada la naturaliza del modelo, aparte de las restricciones que establecen las multiplicidades se definirán restricciones que conciernen a los eventos, estas són:

#### **date\_begin menor que date\_end :**

Las fecha de inicio de un evento siempre tendrá que ser menor o igual que su fecha de finalización.

#### **Evento antes de un nacimiento :**

No se podrá establecer un intervalo de fechas en un evento que entre en conflicto con un intervalo de fechas de un nacimiento de una misma persona.

#### **Evento despues de una defunción :**

No se podrá establecer un intervalo de fechas en un evento que entre en conflicto con un intervalo de fechas de una defunción de la misma persona.



## 9. Diseño

En este apartado se presentara el diseño del software derivado de la especificación, este diseño busca representar la estructura que tendrá que tener el software y los patrones que se aplicaran, todo con el objetivo de mejorar la escalabilidad, portabilidad y reusabilidad. Con este objetivo se desplegara el diagrama de modelo conceptual de datos (ver apartado ??) concretando la estructura con la que se tendrá que desarrollar el sistema y se explicara la arquitectura que se usara para llevarlo a cabo.

Para poder entender los diagramas de diseño primero introduciremos los patrones usados en el proyecto, gran parte de estos patrones están extraídos del libro Design patterns [10].

### 9.1. Patrones de arquitectónicos

#### 9.1.1. Modelo vista controlador

Para el desarrollo del proyecto se usara el framework de desarrollo Django, este framework esta diseñado para seguir la arquitectura modelo , vista, controlador. Con la puntualidad que se ha modificado la nomenclatura a modelo, vista, template.

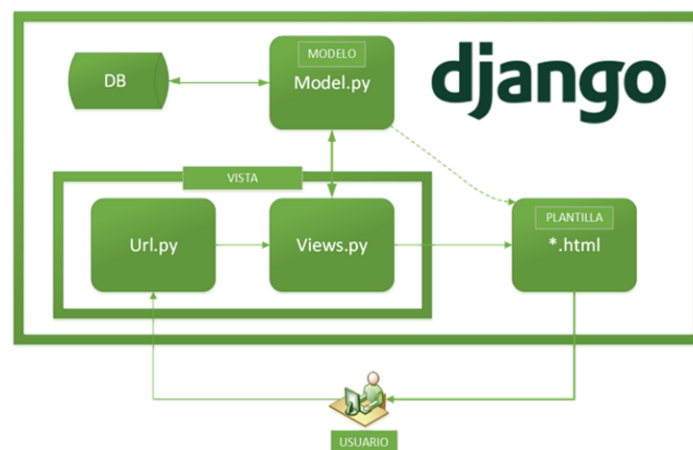


Figura 7: Arquitectura propuesta por el framework Django

Los modelos son los encargados de contener las clases que representan nuestro modelo de datos, en las vistas se encuentra la lógica de la aplicación y en las plantillas se renderiza el resultado para presentarlo al usuario. Para este proyecto se ha usado otro framework que trabaja sobre Django, Django API rest framework, este nos proporciona herramientas para llevar a cabo APIs rest con Django, una de estas herramientas son los serializadores, encargados de transformar nuestras respuestas a json o XML, de tal forma que nuestra arquitectura finalmente quedaría de

la siguiente forma.

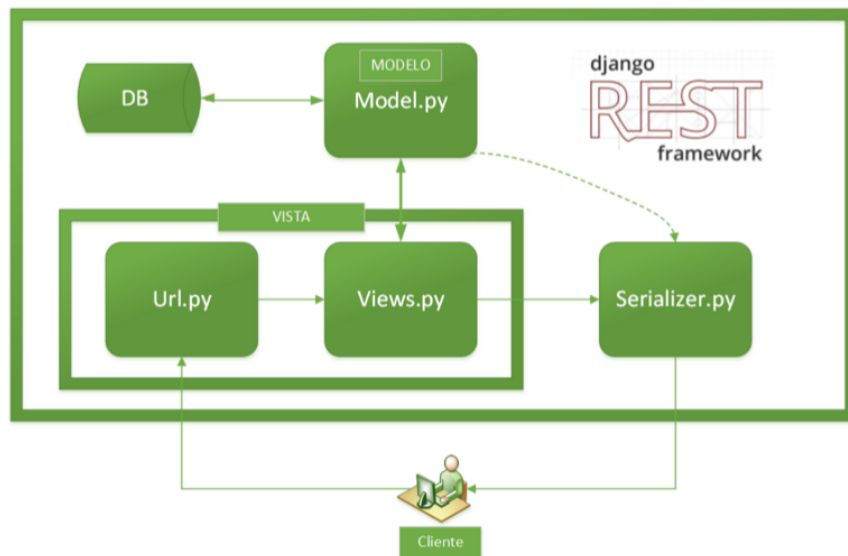


Figura 8: Arquitectura propuesta por Django Rest Framework

El objetivo final de este patrón es separar la representación interna de la información de la manera en que la información es presentada o aceptada. De esta forma, en el proyecto, la persistencia solo es tratada en los modelos, la lógica del sistema en las vistas y la presentación en los serializadores.

### 9.1.2. Message Broker

Es un patrón usado para la validación de mensajes, transformación y enrutamiento. El “message broker” hace de mediador entre aplicaciones permitiendo el paso de mensajes entre ellas sin necesidad de tener en cuenta el estado del sistema emisor o receptor.

En este proyecto se ha usará Celery y RabbitMQ para desplegar este patrón arquitectónico. Celery se encarga de encolar los mensajes y configura nuestros workers que consumirán del “message broker” en este caso RabbitMQ.

Con este patrón arquitectónico se se basa en el patrón de mensajes “publish–subscribe”, en el apartado destinado a patrones de mensajes se explicara en detalle este patrón.

La figura 9 podemos ver un diagrama de la arquitectura de este patrón con las tecnologías que se han decidido usar.

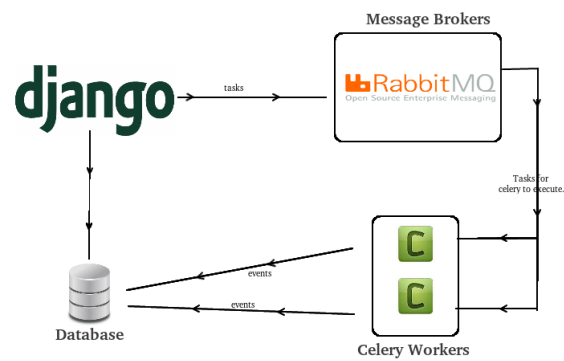


Figura 9: Arquitectura para usar message broker

Se ha decidido usar este patrón arquitectónico para hacer que ciertas tareas muy pesadas del sistema se hagan de forma distribuida, concurrente y asíncrona, esto hará que el usuario si requiere una de las tareas que siguen este patrón pueda recibir un mensaje de aceptación y ser notificado cuando la tarea termine. Las tareas son la subida de archivos Gedcom y la búsqueda de usuarios similares.

### 9.1.3. REST

REST es el acrónimo de a Transferencia de Estado Representacional (Representational State Transfer), es un estilo de arquitectura para sistemas de hipermedia(i.e. término con el que se designa al conjunto de métodos o procedimientos para escribir, diseñar o componer contenidos) distribuidos. Las arquitecturas REST usan el protocolo HTTP para dar acceso a los recursos del sistema. Los fundamentos clave de REST son:

**Protocolo cliente/servidor sin estado** : Cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes.

**Operaciones bien definidas** : HTTP en sí define un conjunto pequeño de operaciones, las más importantes son POST, GET, PUT y DELETE. Con frecuencia estas operaciones se equiparan a las operaciones CRUD en bases de datos.

**Sintaxis universal** : En un sistema REST cada recurso es accesible únicamente a través de su URI.

**Uso de hipermedios** : Las respuestas llevan referencias a los recursos con los que está relacionado el recurso consultado, permitiendo navegar entre ellos.

Se ha optado por el uso de este patrón ya que permite a nuestro sistema ser utilizado desde cualquier plataforma (aplicaciones móviles, *front-ends* web), dado que toda la comunicación REST se realiza por protocolo HTTP.

## 9.2. Patrones de mensajería

### 9.2.1. *Publish-subscribe*

El patrón *publish-subscribe* consiste en plantear un sistema de envío de mensajes entre dos entidades, una encargada de enviar los mensajes, llamada *publisher* y otra encargada de tratarlos, llamada *subscriber*. El patrón establece que tanto el *publisher* como el *subscriber* tienen que funcionar de manera independiente, es decir que no necesiten saber de la existencia de estos. Esto se consigue mediante una cola de mensajes, en este proyecto será RabbitMQ como se ha explicado en el apartado en el que se explica el patrón *message broker* 9.1.2. Las ventajas de este patrón son:

Mayor escalabilidad de la infraestructura. Esta escalabilidad la conseguiríamos creando nuevos *subscribers* que escuchen a la cola de tareas.

Reducción del acoplamiento, dado que tanto el *publisher* como el *subscriber* pueden funcionar de forma independiente uno del otro.

## 9.3. Patrones Estructurales

### 9.3.1. Mixins

Un mixin es una clase que ofrece funcionalidades y puede ser heredada por una subclase, estas clases no están pensadas para funcionar de forma autónoma. Se puede entender como una inyección de código. Este patrón refuerza el principio de inversión de dependencias (i.e. una forma de reducir el grado de interdependencia entre los módulos). Este patrón estructural es el que usa Django rest framework para facilitar la creación de vistas. El framework nos proporciona varios *mixins* que contienen las llamadas a los serializadores necesarias para efectuar las operaciones CRUD sobre un modelo.

## 9.4. Protocolos

Como ya se ha comentado en los apartados anteriores la API a desarrollar sera REST, en consecuencia al autenticar y autorizar de los usuarios que hagan uso de nuestra API se tendrá que hacer de una forma que el sistema no necesite conservar el estado. Para ello se usara OAuth 2, un protocolo que permitirá autenticar y autorizar a nuestros usuarios en el sistema.

### 9.4.1. OAuth 2

La funcionalidad básica de OAuth 2 es permitir a terceros autenticar usuarios contra un sistema, mediante el uso de tokens. Para ello se vale de diferentes mecanismos. para este proyecto se usara el mecanismo *password based*, que no permite aislar el control de las credenciales de usuario a las clientes que usan nuestra API pero por contrapartida su mecanismo es más simple. Primero definiremos la nomenclatura que se usara.

**User** : Es el usuario final que almacenara los datos en nuestra API.

**Client** : Es una aplicación de terceros que da servicio a los usuarios haciendo de interfaz entre la API y el usuario.

**Provider** : API contra la que los clientes se pueden autenticar mediante OAuth.

**Client secret y client id** : Estos dos elementos identifican a un cliente, son las credenciales que este usa para conseguir los tokens que permiten a los usuarios validarse.

**Resource** : Un *resource* es un recurso almacenado por un *user* que nuestra API almacena.

**Acces token** : Un *acces token* es el token que una vez obtenido por el cliente le permite validar al usuario que lo esta usando contra nuestra API. Para mejorar la seguridad, los tokens tendrán un tiempo de vida, con lo que se tendrán que ir renovando.

A continuación se adjunta un diagrama genérico de los pasos que tiene que seguir un cliente, para adquirir un recurso usando las credenciales de un usuario.



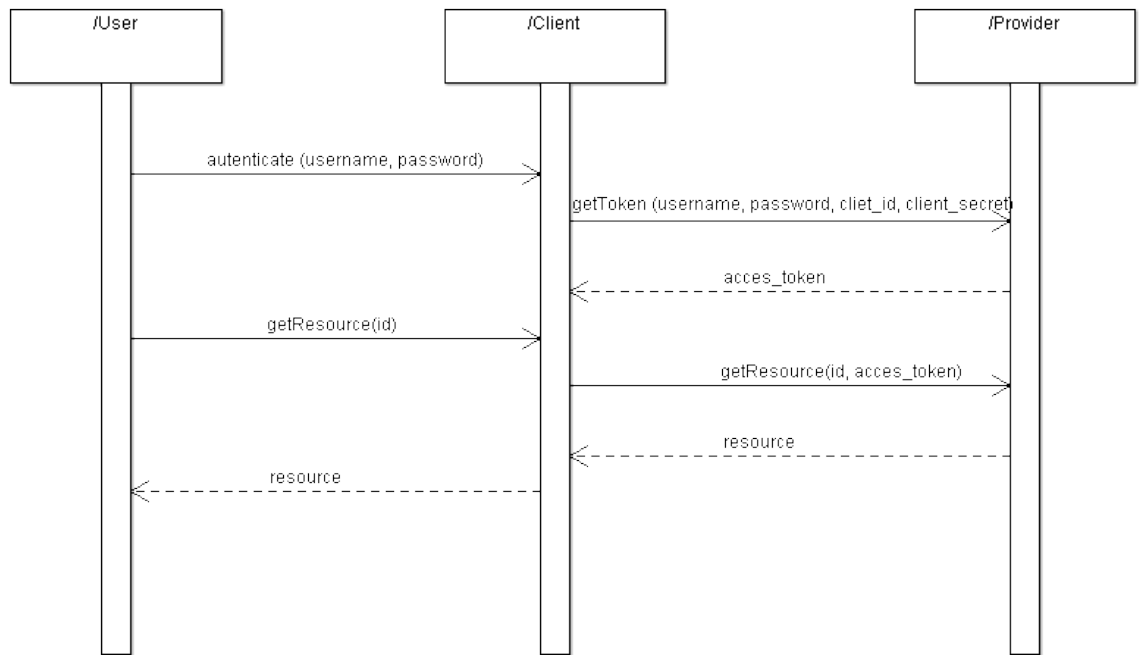


Figura 10: Diagrama para de autenticación con OAuth, password based

El siguiente diagrama representa el flujo para pedir un token de renovación.

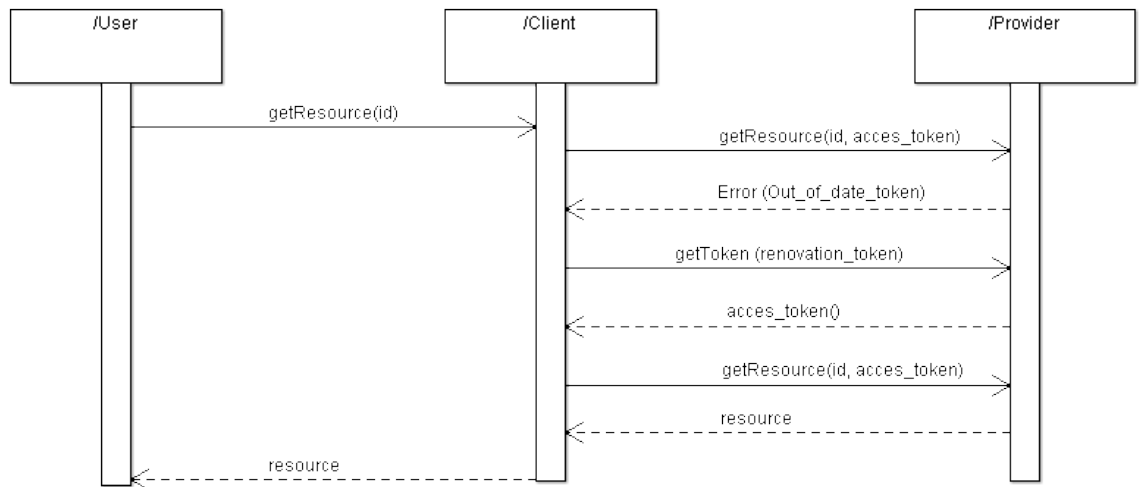


Figura 11: Diagrama para renovar acces token con OAuth

Como podemos ver en la figura 10 la interacción empieza con el intento de autenticación de un usuario mediante una aplicación. Seguidamente la aplicación se identifica contra el *provider* proporcionando sus credenciales y las del usuario. Como resultado el *provider* retorna un *acces token* que permitirá autenticar al usuario con los permisos que este disponga. Con este *acces token* el cliente podrá pedir recursos al *provider* y este podrá usar las credenciales del token para autenticar al usuario en cada petición y saber sus permisos.

En la figura 11 vemos como se realiza la renovación de un token una vez el cliente se da cuenta que este ha caducado. La secuencia empieza con una llamada del cliente para conseguir un recurso, a lo que el *provider* responde con un mensaje de error notificando que el token ya no es válido. Seguidamente el cliente hace una llamada a la uri usada para conseguir el token, con el *renovation token* y el *provider* retorna un nuevo token.

El funcionamiento de todo el protocolo está definido en el RFC 6750 titulado The OAuth 2.0 Authorization Framework [1].

En la fase de implementación se explicará con que endpoints concretos se usarán estas funcionalidades y que parámetros se tendrán que proporcionar.

## 9.5. Diseño del modelo de datos

El sistema a desarrollar es una API rest-ful que permita la interacción de aplicaciones de terceros sobre nuestros datos, con ese objetivo nuestro modelo ha de constar de mecanismos para que estos terceros se autenticuen contra nuestro sistema. Para ello se implementara un sistema para autenticar y autorización basado en oAuth 2.0 como se ha explicado en el apartado 9.4.1 , el diagrama especificado a continuación amplia el del modelo conceptual de datos que podemos ver en la figura 6 añadiendo todas las clases necesarias para su futura implementación con los patrones y protocolos explicados en los apartados anteriores.

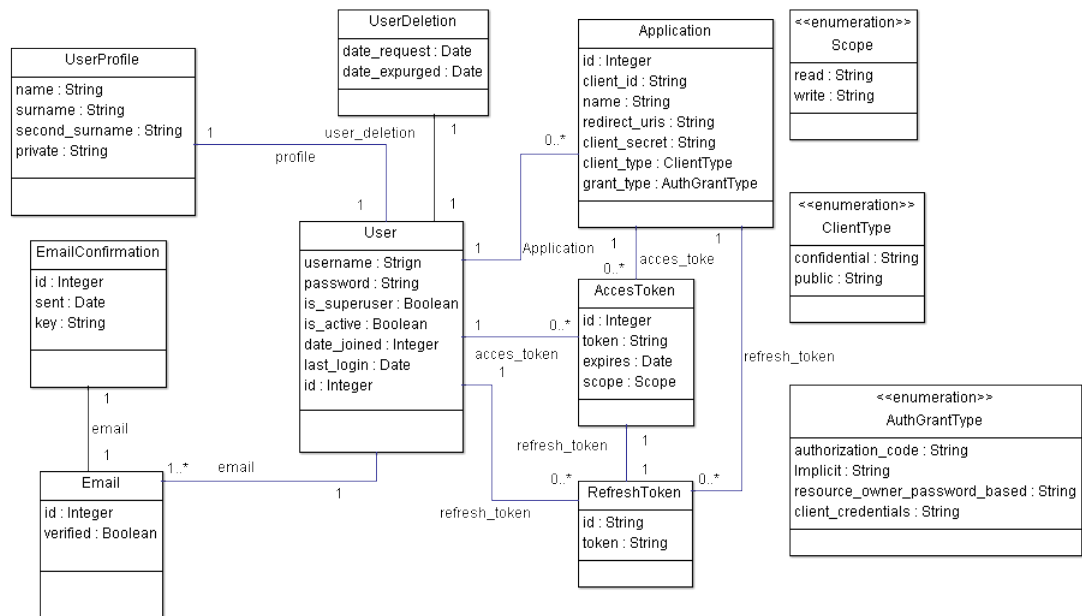


Figura 12: Diagrama modelo de datos relacional

Se ha decidido que la gestión de usuarios se hará usando modelo relacional, esta decisión se a tomado teniendo en cuenta que nuestro sistema no realiza consultas donde se tenga que recorrer el modelo, por lo que no tiene sentido usar una base de datos orientada a grafos, en lo que se refiere a al gestión de usuarios. De esta forma también se puede aprovechar las librerías que proporciona Django para la gestión de usuarios, que usan el modelo relacional.

A continuación se explican las clases definidas en el diagrama:

**User** : Representa un usuario del sistema, sus atributos son los datos esenciales para que este pueda hacer uso del sistema.

**UserProfile** : Representa los datos personales de un usuario.

**Email** : Representa un correo electrónico que esta asociado a un usuario.

**EmailConfirmation** : Contiene los datos referentes a la validación de un correo electrónico de un usuario. La key es la clave que el usuario tiene que usar para validar el correo y sent la fecha en la que se ha enviado el correo pidiendo la validación.

**UserDeletion** : Contiene las peticiones de borrado de cuenta de los usuarios, `date_request` indica la fecha en la que se ha realizado la petición de eliminación y `date_expurged` la fecha en que la cuenta debe ser borrada definitivamente del sistema.

**Application** : Esta clase se refiere a los clientes que dan de alta los usuarios. Estos clientes son las aplicaciones de terceros que están dadas de alta para usar nuestro sistema. Se incluyen todos los atributos necesarios establecidos por el RFC que describe OAuth [1] para describir un cliente. A pesar que solo se usaran los mencionados en el apartado de diseño 9.4.1.

**AccessToken** : Contiene los AccessToken que ha solicitado un cliente. Contiene todos los atributos especificados en el RFC que describe OAuth2, de la misma forma que en *Application*.

**RefreshToken**: Esta clase contiene el *refresh token* necesario para renovar un *access token* que ya ha expirado, tal como se explica en el apartado 9.4.1.

A continuación, en la figura 13 se encuentra el diagrama de diseño del árbol genealógico:

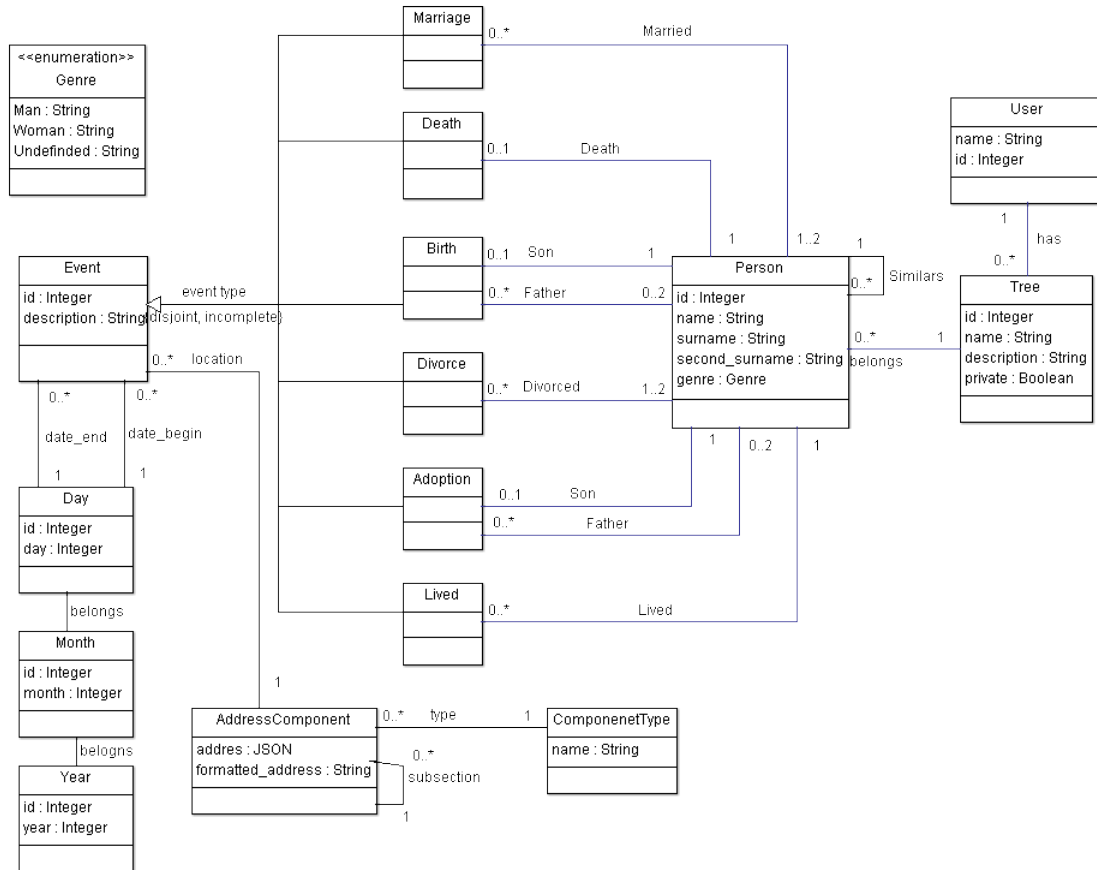


Figura 13: Diagrama modelo de datos orientado a grafos

Esta parte del modelo de datos se implementará usando un modelo orientado a grafos, ya que es sobre los datos que nuestro sistema tendrá que realizar gran parte de consultas que requieren recorrer el dominio. El diseño está hecho de tal forma que se pueda aprovechar la adyacencia libre de índice explicada en el apartado 2.2.2.

La clase *User*, será un nodo que compartirá el id con el de la clase *User* de la base de datos relacional, para poderse referenciar de forma cruzada.

En las siguientes selecciones se explica cómo se ha diseñado más concertadamente las fechas y las localizaciones en el modelo orientado a grafos y los beneficios de este diseño, las otras clases siguen la misma definición que las del modelo conceptual definido en la sección 8.2.

### 9.5.1. Árboles temporales.

Con el objetivo de aprovechar la adyacencia libre de índice los las DBOG se implementara una estructura arbórea donde se representaran las fechas, esta estructura se propone en el libro GrapDB 1, esto nos permitirá gran rapidez a la hora de consultar todos los eventos que hayan sucedido en un año, mes o día. Podemos ver un ejemplo de su estrucutra en la figura 14

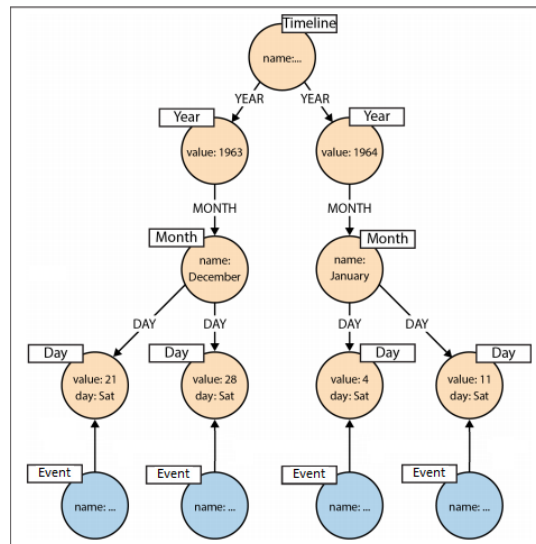


Figura 14: Ejemplo de un *timeline tree*

En esta estructura llamada *Timeline tree* 14 cada año tiene su propio conjunto de meses y cada mes tiene su propio conjunto de días. Para mantener la estructura solo tenemos que añadir nodos cuando estos sean necesarios.

Los nodos Year, Month y Day representan las clases con el mismo nombre del diagrama de la figura 13.

### 9.5.2. Geocomponentes

Aplicando la misma lógica que para las fechas también se creará una estructura arbórea para las localizaciones. La peculiaridad de esta estructura es que cualquier elemento del árbol podrá tener una relación con el evento, en caso que no se pueda concretar la localización con más detalle. Y de la misma forma que en el *timeline tree* los nodos serán únicos, por lo que solo se tendrán que crear en el caso que no existan ya en la base de datos. Podemos ver un ejemplo de esta estructura en la figura 15.

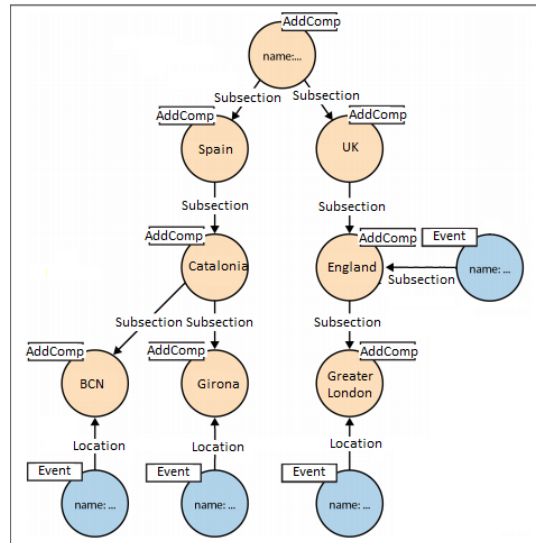


Figura 15: Ejemplo de un árbol de geocomponentes

## 9.6. Diseño de la capa de dominio

Se harán dos diagramas con el fin de mejorar la comprensibilidad de estos, uno correspondiente al *publisher* y otro al *subscriber*, siguiendo la nomenclatura definida en la explicación del patrón *publish-subscriber* 9.2.1.

A continuación explicaremos el diagrama del *publisher*, que podemos ver en la figura 16. Este diagrama esta simplificado y no contiene las clases referentes a los eventos, si bien siguen la misma estructura que las clases referentes a las personas y arboles.

Podemos diferenciar 3 componentes principales siguiendo la adaptación del patrón MVC explicado en la sección 9.1.1, estos componentes son:

**Models** : Estos contienen los atributos especificados en el apartado 8.2, más concretamente en la figura 13. Y todas las operaciones necesarias para tratar estos datos. Dado que Neo4j no permite definir restricciones de integridad en el SGBD se ha creado una clase *restriction*, donde se implementan todas y son llamadas desde el modelo, ejecutándose antes de realizar las operaciones que necesiten comprobar alguna restricción. En el diagrama salen representados como las clases *Person* y *Tree*

**Views** : Se han implementado usando los *mixins* que proporciona API rest framework, estas clases añaden las operaciones equivalentes a las operaciones CRUD de nuestra base de datos llamando a los serializadores que definimos en nuestras vistas, en este caso *TreeViewSet* y *PersonViewSet*. Por otro lado la clase *APIView* nos proporciona herramientas para tratar las request, que contienen la información de la petición a tratar. Por ultimo *Response* es una clase de Django nos da las operaciones que nos permiten enviar una respuesta a la *request* solicitada.

**Serializers** : Estas clases se encargan de serializar y deserializar la información que llega en la *request*. Para implementar los serializadores se ha usado la clase *BaseSerializer* de API rest framework, esta clase nos proporciona las funciones que las clases *mixins* de las vistas usaran para realizar sus operaciones, el atributo *many* nos permite serializar y deserializar conjuntos de datos en caso que este sea *True*. Para poder usar esta clase tenemos que implementar los métodos definidos en nuestros serializadores, *TreeSerializer* y *PersonSerializer*. *to\_internal\_value* se encarga de transformar los datos que llegan de la *request* en forma de string los tipos de datos que se usen en el sistema, estos datos se guardaran en el atributo *valid\_data*, *to\_representation* recibe un objeto de nuestra base de datos y lo serializa a un JSON con los datos que retorna a una petición, por ultimo *create* y *update* realizan dichas operaciones con la información que *to\_internal\_value* a convertido a tipos de datos que usa nuestro sistema.

A continuación se explican las otras clases y se explica su función.

**permissions** : Esta clase es un ejemplo de un permiso al que esta restringida una *view*. Antes de ejecutar ninguna operación, las *views* comprueban que todos los permisos se cumplan.



Las *views* del sistema usan los siguientes permisos:

- `IsAuthenticated`: Comprueba que el usuario que realiza la *request* esta autenticado
- `TokenHasReadWriteScope`: Comprueba que el usuario que realiza la *request* tenga permisos de lectura y escritura sobre los recursos accedidos.

**router** : Esta clase es la encargada de enturar las peticiones según el *endpoint* por el que han llegado a las *views* correspondientes

**celery** : Esta clase es la encargada de enviar las tareas al *message broker*, tal y como se explica en el apartado 9.1.2. Para enviar la tarea de cargar un gedcom, lo que hace es serializar la función que realiza esta tarea y la envía mediante la función *async* al *message broker*.

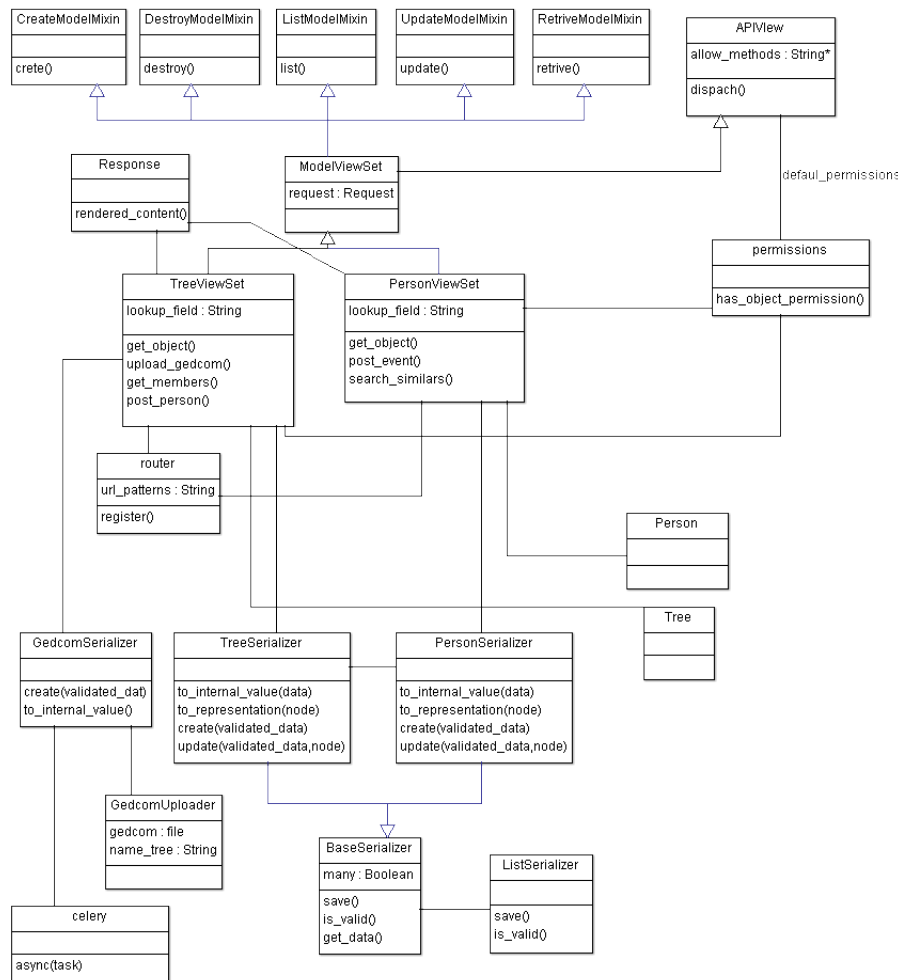


Figura 16: Diagrama de la capa de domino

Ahora se explicara el diagrama del *subscriber* encargado de tratar con las tareas de cargar un GEDCOM. El diagrama lo podemos ver en la figura 17 .

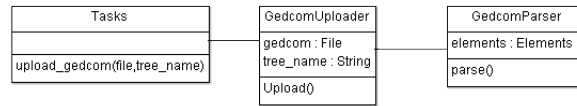


Figura 17: Diagrama de la capa de domino

**pendiente**

## 9.7. Diseño del modelo de comportamiento

En este apartado se incluyen los diagramas del modelo de comportamiento para las funcionalidades del sistema. Para la comunicación entre los modelos vistas y serializadores se hará un diagrama que generalice el funcionamiento. Para las operaciones más concretas se añadirá un diagrama para cada una.

Como ejemplo se usara la obtención de una persona.

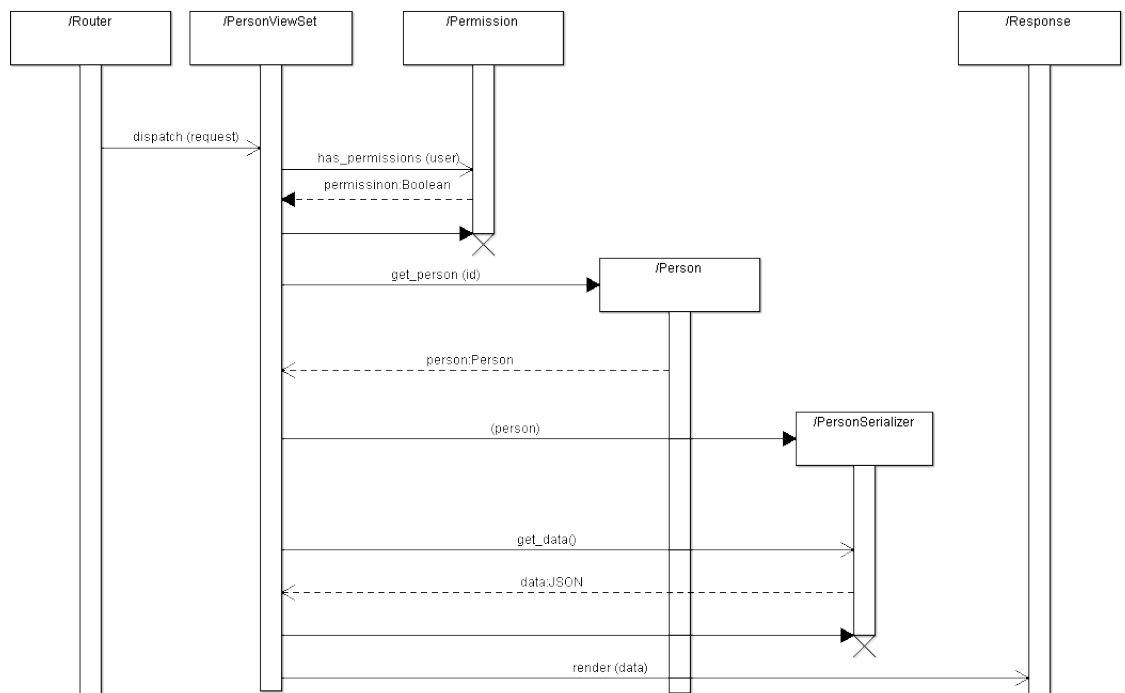


Figura 18: Diagrama de secuencia para obtener una persona

Aquí se adjunta un diagrama de una operación más compleja, pero que lógicamente sigue la misma lógica que la anterior.

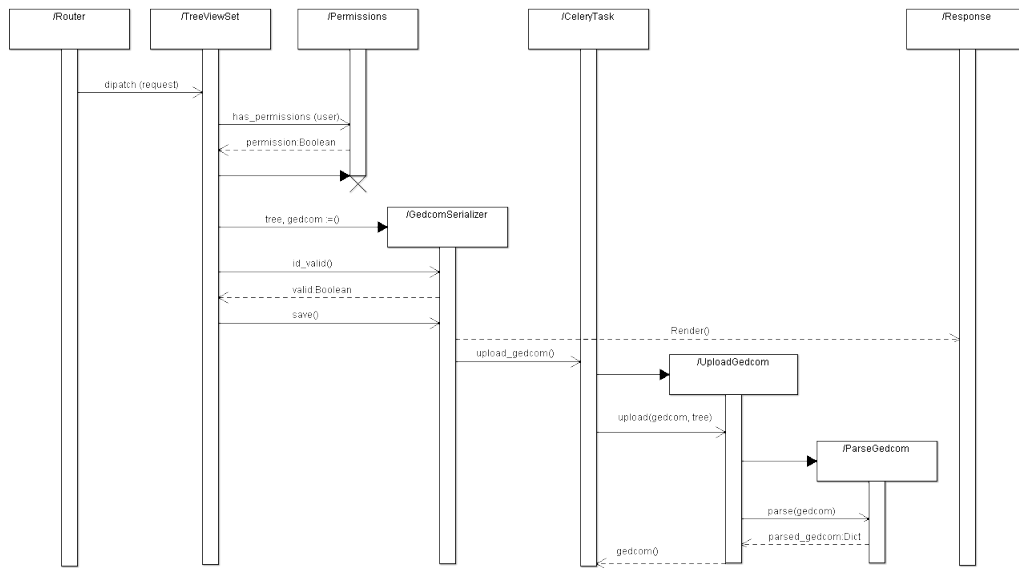


Figura 19: Diagrama de secuencia para subir un archivo GEDCOM

## 10. Implementación

En esta sección se explica el desarrollo del proyecto y el análisis de la implementación. Para facilitar la comprensión se ha separado en diferentes etapas.

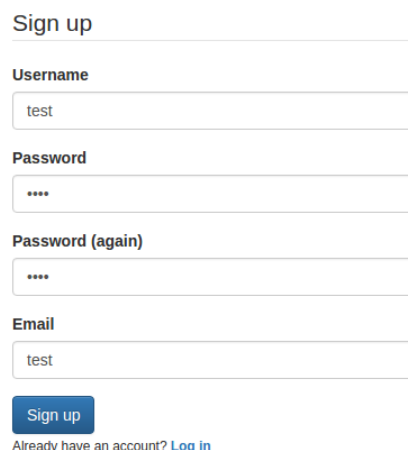
### 10.1. Usuarios

Para el desarrollo de la gestión de usuarios se han usado diferentes librerías auth y contrib principalmente. Estas librerías permiten guardar los datos de usuarios más básicos y herramientas para validarlos.

Para la gestión de usuarios se ha añadido el modulo llamado account que mediante diferentes endpoints permite la ejecución de views que realizan gran parte del trabajo.

#### 10.1.1. SignUp

La creación de las cuentas de usuario se tienen que realizar mediante un template(i.e. no se ha habilitado ningún endpoint de la API). El objetivo es que usuarios puedan crear aplicaciones que usen nuestra API. El formulario que podemos ver en la figura 20



The image shows a web form for user registration. It has a title 'Sign up' followed by a horizontal line. Below the line are four input fields: 'Username' with the value 'test', 'Password' with four asterisks, 'Password (again)' with four asterisks, and 'Email' with the value 'test'. At the bottom of the form is a blue button labeled 'Sign up'. Below the button is a link that says 'Already have an account? Log in'.

Figura 20: Registro de un usuario

Como podemos ver en el formulario se pide la confirmación del password. Para ellos se ha añadido validación de campos. En la figura 21 podemos ver el formulario.

La validación de campos se hace de la siguiente forma.

The screenshot shows a 'Sign up' form with the following elements:

- A red error message at the top: "You must type the same password each time." with a close button (x).
- A 'Username' field containing the text 'test'.
- A 'Password' field (empty).
- A 'Password (again)' field (empty).
- An 'Email' field containing the text 'test', with a red error message below it: "Enter a valid email address."
- A blue 'Sign up' button.
- A link below the button: "Already have an account? [Log in](#)".

Figura 21: Mensajes de error en el registro

Aquí podemos ver el código del template:

```
<form id="signup_form" method="post" action="{% url 'account_signup' %}"
autocapitalize="off" {% if form.is_multipart %}
enctype="multipart/form-data"{% endif %}>
    <legend>{% trans "Sign up" %}</legend>
    {% csrf_token %}
    {{ form|bootstrap }}
    {% if redirect_field_value %}
        <input type="hidden" name="{% _redirect_field_name _}"
        value="{% _redirect_field_value _}" />
    {% endif %}
    <button type="submit" class="btn btn-primary">
        {% trans "Sign up" %}
    </button>
</form>
```

Aprovecharemos la oportunidad para explicar como funciona el sistema de templates de django, el código que se encuentra entre los tags `{% codigo %}` es código python que modifica el comportamiento de forma dinámica. Por otro lado el comando `csrf_token` añade seguridad contra los ataques Cross Site Request Forgery a los formularios.

Viendo el código nos preguntamos como se añaden los campos al formulario, esto se hace de la siguiente forma, el tag `{{ form }}` lo que hace es inyectar los campos que se encuentran en la clase especificada en la vista que "action" dispara, para ello la vista tiene que disponer del atributo

from\_class y se tiene asignar la clase que hereda de django.forms.Form donde están especificados los atributos. Para más clarificación se adjunta el código del caso concreto que estamos viendo.

```
class SignupForm(forms.Form):

    username = forms.CharField(
        label=_("Username"),
        max_length=30,
        widget=forms.TextInput(),
        required=True
    )
    password = forms.CharField(
        label=_("Password"),
        widget=forms.PasswordInput(render_value=False)
    )
    password_confirm = forms.CharField(
        label=_("Password_(again)"),
        widget=forms.PasswordInput(render_value=False)
    )
    email = forms.EmailField(
        label=_("Email"),
        widget=forms.TextInput(), required=True)

    code = forms.CharField(
        max_length=64,
        required=False,
        widget=forms.HiddenInput()
    )

class SignupView(FormView):
    ...
    form_class = SignupForm
    ...
```

El registro de un usuario inicia automáticamente la validación del correo electrónico y así también la validación de la cuenta.

### Confirmación .

Para la verificación de la cuenta se ha añadido confirmación vía correo electrónico. Para ello se dispone de dos clases que permiten almacenar los correos electrónicos de los usuarios y se estos han estado verificados o no y otra que contiene la llave que se tiene que usar para verificar la cuenta.

El proceso de verificación empieza cuando el usuario se crea una cuenta de usuario, entonces se envía un correo al usuario con la url que contiene la llave de validación.

domain/accounts/confirm\_email/key/

Esta URL ejecuta la vista encargada de validar la cuenta, y nos muestra una interfaz desde la que confirmar el correo ( 22).

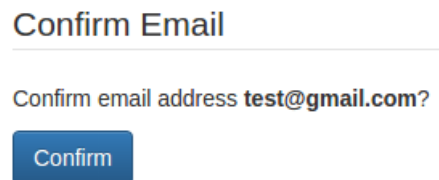


Figura 22: Pantalla de confirmación del correo electrónico

Al confirmar nos redirige a la pantalla de log in que podemos ver en la figura 23.

### 10.1.2. LogIn

Para entrar lo podremos hacer con el botón que encontramos arriba a la derecha, que hará que se renderice el template de log in (figura 23).

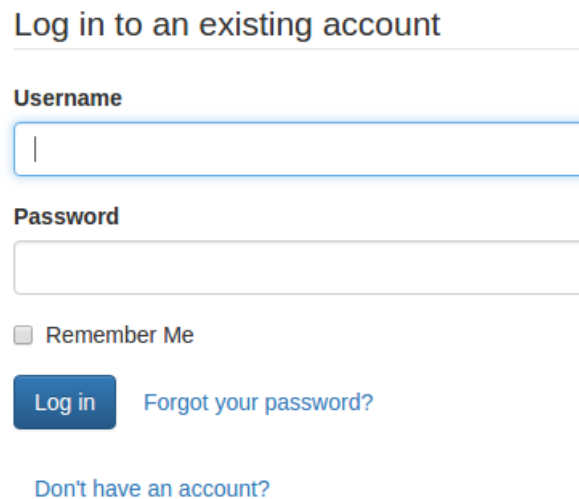


Figura 23: Interface para entrar

Una vez se introducen las credenciales correctamente y se pulsa sobre el botón Log in esto ejecuta la vista asociada. Esta vista ejecuta la función login de la librería django.contrib.auth,



que guarda el id del usuario usando la librería sesión de django, que gestiona las cookies. Con la función `sesión.set_expiry(0)` haremos que la sesión se mantenga abierta hasta que el navegador limpie las cookies.

## 10.2. Gestión de aplicaciones

Para facilitar la autenticación de los usuarios mediante oautuh 2 se ha utilizado la librería oautuh2 toolkit, esta librería nos proporciona todas las clases necesarias para poder usar este protocolo. Las clases que esta librería nos crea en el sistema son las siguientes.

**Applications** : En esta clase se almacenaran las aplicaciones dadas de alta por los usuarios. Estas aplicaciones podrán conseguir tokens de otros usuarios mediante el uso del protocolo establecido.

**AccesTokens** : En esta clase se almacenan los acces token que son validos en este momento y ligados al usuario que pertenecen. Cada acces token tiene un tiempo de vida que también se almacena en esta clase.

**RefreshToken** : Cada acces token tiene asignado un refresh token para poder refrescar un token al que el tiempo de vida se le haya acabado.

Una vez accedamos a la gestión de aplicaciones encontramos la interface desde la que podremos crear aplicaciones como podemos ver en la figura 24.

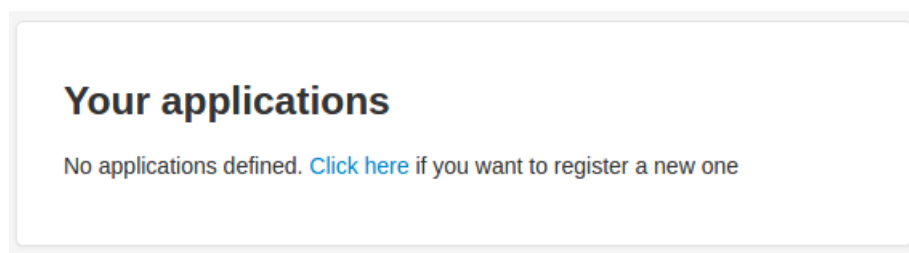
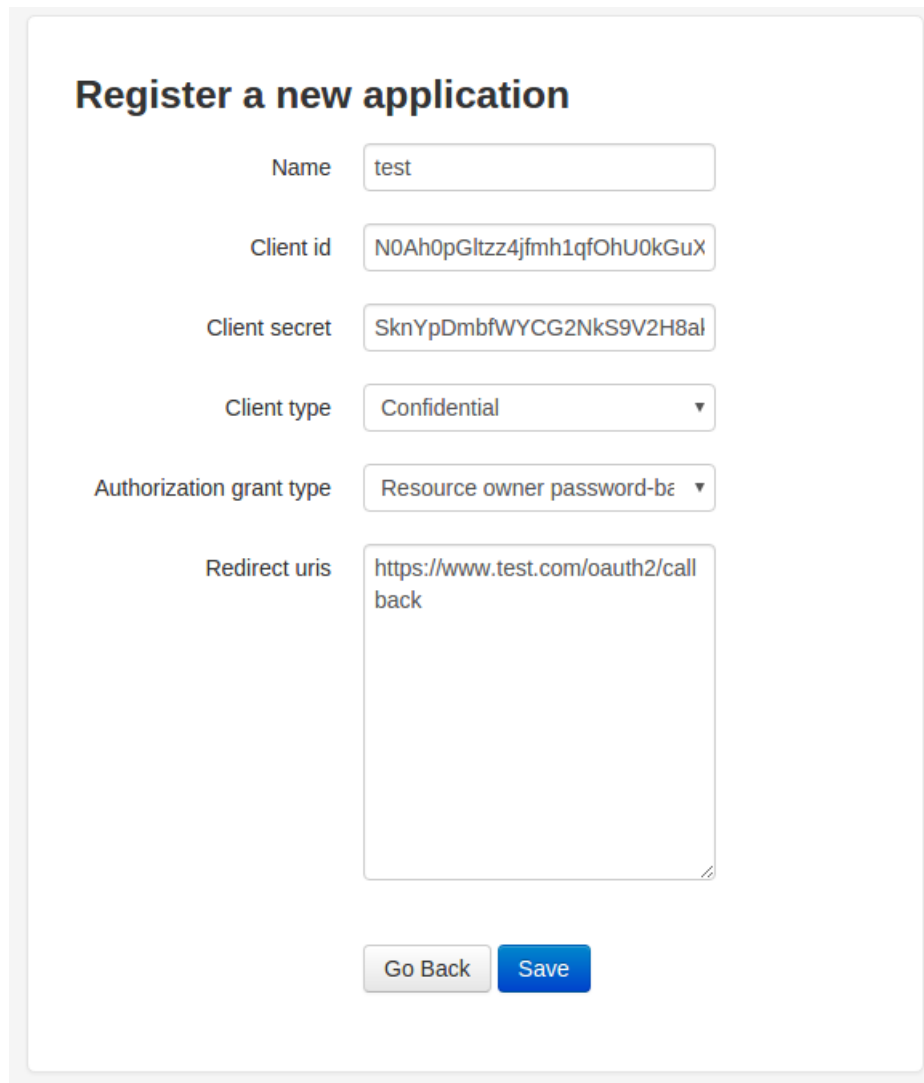


Figura 24: Interfaces para ver las aplicaciones

Al pulsar sobre el link que nos indica, nos dirigirá a un formulario donde podremos crear la aplicación, el formulario lo podemos ver en la figura 25

Los campos `client id` y `client secret` se nos llenaran automáticamente, en este caso `client type` y `authorization type` se dejan escoger por cuestiones practicas a la hora de desarrollar pero también se fijarían automáticamente en una version de producción. En este caso solo se ha testado la autorización con gran type `Resource owner password-based`, en la que la aplicación gestiona las credenciales de los usuarios que ha de autenticar contra nuestra aplicación. No se entrara en detalle de como se implementa ya que lo gestiona el framework `oauth2 toolkit` y no tiene más complejidad que dar de alta la aplicación en nuestra base de datos.



**Register a new application**

Name

Client id

Client secret

Client type

Authorization grant type

Redirect uris

Figura 25: Formulario para la creación de aplicaciones

### 10.3. Endpoints

Los endpoints son los que dan las funcionalidades de la API. Se explicaran de forma pormenorizada pero en 2 grupos ya que si bien nuestra API da servicio para gestionar muchos recursos la implementación conceptualmente es muy parecida entre ellos como ya se deducía en la especificación y diseño. Estos 2 grupos son los endpoints que realizan las operaciones CRUD y los endpoints con funcionalidades más específicas que funcionan de forma asíncrona mediante celery. También se explicara el funcionamiento de django rest framework (DRF)

### 10.3.1. CRUD

Las operaciones crud se realizan mediante diferentes endpoints siguiendo el estilo rest-ful. Para definir las urls por donde se podrá acceder a estas operaciones se definen mediante el uso de la clase router a continuación se adjunta un ejemplo:

```
router.register(
    r'person',
    views_genetree.PersonViewSet,
    base_name='person')
```

Este código hará que todas las request a la url enruten a la view de PersonViewSet, en esta view según el tipo de method realizara las operaciones pertinentes. PersonViewSet esta implementado usando la clase ModelViewSet que nos proporciona DRF, esta clase esta pensada para funcionar con el ORM de django y en nuestro caso usamos neomodel un object graph mapper (OGM) que explicaremos más adelante, por este motivo se ha modificado la función get\_object, con el objetivo que funcione correctamente. Una parte del código de PersonViewSet que nos ayude a entender como se ha implementado es el siguiente:

```
class PersonViewSet(viewsets.ModelViewSet):
    permission_classes = [IsAuthenticated, TokenHasReadWriteScope]
    queryset = models_person.Person.nodes
    serializer_class = serializer_person.PersonSerializer
    lookup_field = 'id'

    def get_object(self):
        qset = copy.deepcopy(self.queryset)
        try:
            res = qset.get(id=self.kwargs[self.lookup_field])
            return res
        except:
            raise Http404("No Person matches the given query.")
```

Como podemos ver lo que nos pide la clase ModelViewSet es que definamos que permisos tiene el recurso (permission\_classes), sobre que conjunto de datos trabajara (queryset), que clase usamos para serializar los datos (serializer\_class) y el atributo por el que se indexan los objetos de nuestro queryset. Con el overwrite de get\_object conseguimos modificar el comportamiento de ModelViewSet para que funcione correctamente con neomodel, la deepcopy se realiza para que las operaciones realizadas sobre nuestro queryset no lo modifiquen.

Los serializadores están definidos usando la clase BaseSerializer de DRF, para ello tenemos que definir las siguientes funciones:

**to\_internal\_value** : Esta función se encarga validar los datos que se quieren serializar y transformarlos en el tipo de datos correspondiente ya que llegan en forma de string dentro de

un JSON.

**to\_representation** : Esta función, dado un objeto de nuestro modelo se encarga de crear un JSON que representa la información del objeto.

**create** : Una vez validados los datos implementa de que forma se han de crear los objetos.

**update** : Una vez validados los datos implementa de que forma se han de actualizar los objetos.

Aparte de las operaciones sobre el objeto, se han implementado endpoints para poder realizar operaciones sobre los subrecursos. Esto se ha llevado a cabo con el decorador `@detail_router` y `@list_router`, que permiten definir extensiones a los endpoints. Aquí podemos ver un ejemplo:

```
@detail_route(
    methods=[ 'post ' ],
    url_path='marriage')
def post_marriage(self , request , id=None):
    request.data[ 'spouse2' ] = id
    marriage_serializer = \
        serializer_person.MarriageSerializer(data=request.data)
    if marriage_serializer.is_valid():
        marriage_serializer.save()
        return Response(marriage_serializer.data ,
                        status=status.HTTP_201_CREATED)
    else:
        return Response(marriage_serializer.errors ,
                        status=status.HTTP_400_BAD_REQUEST)
```

Este endpoint nos permite crear matrimonios partiendo de un individuo.

### 10.3.2. Asíncronos

Los endpoints asíncronos son casos especiales del dentro de un recurso, por ello se han implementado usando el decorador `detail_router` y `list_router` según ha convenido. Aquí podemos ver del `detail.router` que se ha implementado dentro de `TreeViewSet` para poder subir árboles genealógicos en formato GEDCOM

```
@list_route(
    methods=[ 'post ' ],
    url_path='gedcom')
def upload_gedcom_tree(self , request):
    serializer = serializer_tree.GedcomSerializer(
        data=request.data , user=request.user.id)
    if serializer.is_valid():
        serializer.save()
```

```
return Response(serializer.data,
                  status=status.HTTP_202_ACCEPTED)
```

Como podemos ver no dista mucho del código que hemos visto para crear matrimonios desde el PersonViewSet. Ya que es el seralizador el que realmente hace el trabajo de llamar a la tarea de celery en el create.

```
def create(self, validated_data):
    fil = validated_data.pop('file')
    res = super(GedcomSerializer, self).create(validated_data)
    gedcom_uploader_task.apply_async((fil, res))
    return res
```

La llamada gedcom\_uploader\_task.apply\_async((fil, res)) envía a RabbitMQ la tarea, en un objeto python serializado y en cuando haya un worker disponible este ejecutara la tarea.

A continuación se explicara como se han implementado las dos funcionalidades más importantes de nuestra API.

### 10.3.3. UploadGedcom

Esta tarea se realiza de forma asíncrona de la forma que se ha explicado anteriormente, se explicara que trabajo realizan los workers para crear el árbol subido en nuestra BDOG. Los workers se valen de la clase GedcomUploader para llevar a cabo esta tarea, como funciona esta clase:

Primero de todo se inicializa cargando el archivo Gedcom y *parseandolo* con la la herramienta que proporciona la classe Gedcom, un parser gedcom desarrollado por la comunidad rootsdev. Primero vamos a explicar como funciona este parser:

Consta de dos elementos principales, los elements y el parser en si, los elementos están inicializados de la siguiente forma en el constructor:

```
def __init__(self, level, pointer, tag, value):
    """ Initialize an element.

    You must include a level, pointer, tag, and value. Normally
    initialized by the Gedcom parser, not by a user.
    """
    # basic element info
    self.__level = level
    self.__pointer = pointer
    self.__tag = tag
    self.__value = value
```

```
# structuring
self.__children = []
self.__parent = None
```

**level** : Indica el nivel dentro del árbol.

**pointer** : Contiene un apuntador a el elemento del gedcom que representa.

**tag** : Contiene un string que indica el tipo del elemento segun el formato GEDCOM

**value** : Contiene un string con el valor del elemento.

**children** : Contiene una lista con todos los hijos según el formato GEDCOM.

**parent** : Contiene el apuntador al padre del elemento si lo tiene.

Por otro lado el parser, implementado en la clase gedcom, se inicializa de la siguiente forma:

```
def __init__(self, filemem):
    """ Initialize a GEDCOM data object. You must supply a Gedcom file. """
    self.__element_list = []
    self.__element_dict = {}
    self.__element_top = Element(-1, "", "TOP", "")
    self.__parse(filemem)
```

**element\_list** : Contiene una lista de elementos en el orden en el que aparecen en el archivo GEDCOM.

**element\_dict** : Contiene un diccionario con todos los elementos que aparecen en el archivo GEDCOM, la llave del diccionario es el puntero al elemento

**element\_top y parse** : Son atributos que se usan internamente.

Cuando llamamos a la función `__parse()` el documento GEDCOM, es parseado y inicializan los atributos `element_list` y `element_dict`, luego con diferentes operaciones podemos consultar los datos para que nos retorne información relevante. Se ha encontrado que las herramientas de consulta no cubrían todas las necesidades, por ello se ha implementado la siguiente función en la librería.

```

def marriage(self, individual1, individual2):
    if not individual1.is_individual() and not individual2.is_individual():
        raise ValueError("Operation only valid for elements with INDI tag")
    # Get and analyze families where individual is spouse.
    fams_families1 = set(self.families(individual1, "FAMS"))
    fams_families2 = set(self.families(individual2, "FAMS"))
    family = list(fams_families1.intersection(fams_families2))
    if family:
        # print family
        # print family[0].children()
        for famdata in family[0].children():
            if famdata.tag() == "MARR" or famdata.tag() == 'DIV':
                for marrdata in famdata.children():
                    date = ''
                    place = ''
                    if marrdata.tag() == "DATE":
                        date = marrdata.value()
                    if marrdata.tag() == "PLAC":
                        place = marrdata.value()
                return date, place
    return None, None

```

Esta función dado dos elementos que han estado casados retorna una tupla con la fecha y el lugar del evento.

Por otro lado el parser solo funcionaba leyendo archivos sin tener en cuenta que estos podían estar “chunkeados” como es nuestro caso (i.e. para facilitar la lectura de archivos de gran tamaño) por lo que se ha modificado la función de lectura que había implementada. Una vez revisadas estas modificaciones se enviara un pull request para que toda la comunidad pueda usar estas nuevas funcionalidades.

Ahora una vez hemos entendido como funciona el parser, la clase GedcomUploader simplemente recoge los datos parseados, los recorre y crea el árbol en nuestra base de datos.

#### 10.3.4. Encontrar similitudes

Para encontrar similitudes, igual que para la funcionalidad que nos permite subir un archivo GEDCOM, se ha realizado de manera asíncrona. Como ya se ha explicado de forma extendida en los apartados anteriores como se han implementado los endpoints de este tipo, en este apartado nos centraremos en explicar como se ha desarrollado la funcionalidad en si misma.

Para empezar hace falta aclarar que si bien se ha querido dar esta funcionalidad para poder trabajar consultas más complejas con neo4j no se ha buscado hacer esta búsqueda de forma

eficiente.

El criterio que se ha usado para inferir si dos personas son similares es la similitud de los eventos que tienen cada una. Para que nuestro sistema pueda realizar estas búsquedas en un tiempo razonable se ha aprovechado la adyacencia libre de índice de las BDOG.

Evidentemente nuestro modelo se ha diseñado de tal forma que se pueda aprovechar este potencial, ya que el uso de esta propiedad está totalmente ligado al diseño de nuestro modelo de datos. Como ya hemos explicado en el diseño tanto las fechas como las localizaciones (i.e. atributos que definen todos nuestros eventos independientemente del tipo) son nodos únicos en nuestro modelo, eso quiere decir que tanto las fechas como las localizaciones una vez creadas simplemente irán añadiendo aristas a los nuevos eventos que las requieran. Para conseguir este comportamiento y que se creen en una estructura arbórea se han creado 2 módulos django que funcionan de manera independiente del proyecto.

### **Date node structure**

Primero explicaremos el módulo `date_node_structre`, encargado de gestionar las fechas. Este módulo consta de 5 clases:

**Day** : Representa un día, contiene los datos especificados en el diseño del modelo de datos.<sup>13</sup>

**Month** : Representa un mes, contiene los datos especificados en el diseño del modelo de datos.<sup>13</sup>

**Year** : Representa un año, contiene los datos especificados en el diseño del modelo de datos.<sup>13</sup>

**Root** : Nodo raíz del que parten todos los años.

**NodeDate** : Esta clase es la que hace todo el trabajo, dada una fecha en Django, nos crea todos los nodos necesarios o simplemente retorna el nodo día de la fecha introducida.

A medida que vamos introduciendo fechas el grafo resultante tiene el aspecto del time tree que se encuentra representado en la figura 14 de la página 54.

Una de las mejoras que se ha añadido a la clase `day` durante el desarrollo que no se había previsto en la especificación es el guardar la fecha como un ordinal en el nodo `Day`, para facilitar las comparaciones posteriormente.

### **Geocomponent node structure**

A pesar que las localizaciones siguen una lógica muy parecida a los timeline trees, a la hora de implementarlos surgen varias complicaciones:

- Como asegurarse la unicidad de los nodos.
- Como aportar suficiente flexibilidad al sistema para aceptar cualquier tipo de localización.

Para solucionar estos dos puntos se ha usado el concepto de geocomponente. Este concepto nace de la geocodificación, una técnica que consiste en convertir coordenadas especificadas a lenguaje natural, de ahí vamos a la geocodificación inversa, que como el nombre indica consiste en convertir



unas coordenadas a una localización en lenguaje natural. Así al hacer geocodificación inversa obtenemos un JSON con la información necesaria para dar nombre a una localización. Usando la API de Google el JSON retornado al consultar una localización tiene el siguiente aspecto.

Consulta:

```
https://maps.googleapis.com/maps/api/geocode/
xml?address=1600+Amphitheatre+Parkway,+Mountain+View,+CA&key=YOUR_APIKEY
```

Respuesta:

```
{
  "results" : [
    {
      "address_components" : [
        {
          "long_name" : "1600",
          "short_name" : "1600",
          "types" : [ "street_number" ]
        },
        {
          "long_name" : "Amphitheatre Pkwy",
          "short_name" : "Amphitheatre Pkwy",
          "types" : [ "route" ]
        },
        {
          "long_name" : "Mountain View",
          "short_name" : "Mountain View",
          "types" : [ "locality", "political" ]
        },
        {
          "long_name" : "Santa Clara County",
          "short_name" : "Santa Clara County",
          "types" : [ "administrative_area_level_2", "political" ]
        },
        {
          "long_name" : "California",
          "short_name" : "CA",
          "types" : [ "administrative_area_level_1", "political" ]
        },
        {
          "long_name" : "United States",
          "short_name" : "US",
          "types" : [ "country", "political" ]
        }
      ]
    }
  ]
}
```

```

        "long_name" : "94043",
        "short_name" : "94043",
        "types" : [ "postal_code" ]
    }
],
"formatted_address" : "1600 Amphitheatre Parkway, Mountain View,
    CA 94043, USA",
"geometry" : {
    "location" : {
        "lat" : 37.4224764,
        "lng" : -122.0842499
    },
    "location_type" : "ROOFTOP",
    "viewport" : {
        "northeast" : {
            "lat" : 37.4238253802915,
            "lng" : -122.0829009197085
        },
        "southwest" : {
            "lat" : 37.4211274197085,
            "lng" : -122.0855988802915
        }
    }
},
"place_id" : "ChIJ2eUgeAK6j4ARbn5u_wAGqWA",
"types" : [ "street_address" ]
}
],
"status" : "OK"
}

```

Como podemos ver el JSON nos retorna varios address\_components, estos componentes son los que usaremos para crear nuestra estructura de nodos. Para ver como creamos nuestros nodos veamos como funciona el modulo geocoding node structure:

Este modulo consta de 4 clases:

**RootLocation** : Esta clase simplemente sirve para crear un nodo base al las localizaciones de tipo country, que son la de mas alto nivel.

**ComponentType** : Esta clase indica el tipo de un AddressComponent

**AddressComponent** : Aquí tenemos toda la información relevante de un componente, el JSON con los componentes que la definen y un string con la dirección formateada.

**Location** : Esta clase es la que hace la magia, creando la estructura de geocomponentes si aun no se encuentra en nuestra BDOG o retornando el nodo solicitado en su defecto.

El árbol resultante tiene el aspecto del árbol que podemos ver en la figura 15.

El porque de estas decisiones de diseño y esta forma de implementar, todo parte de la del artículo Data Fusion Algorithms [5], en este artículo se nos plantea como encontrar nodos similares en un hypergraph partiendo de los hyperedge, dado que neo4j no parte del concepto de hypergraph, tenemos que extraer los atributos de un nodo y hacerlos unicos para simular un hyperedge, en este caso han sido localización y fecha de los nodos evento, esto se ha hecho partiendo del la explicación que nos da neo4j en su documentación [4].

De esta forma podemos aprovechar la adyacencia libre de indices para buscar nodos evento potencialmente similares en un coste razonable.

La consulta realizada para efectuar esta búsqueda es la siguiente:

```
def __get_query_similars():
    return Template("""
        START a=node({ self })
        MATCH a-[:$relation]-(e1)-[:LOCATION|SUBSECTION*]-(loc)-
            [:LOCATION]-(e2)-[:$relation]-(b),
            e1-[:DATE-BEGIN]-(e1_begin),
            e1-[:DATE-END]-(e1_end),
            e2-[:DATE-BEGIN]-(e2_begin),
            e2-[:DATE-END]-(e2_end)
        WHERE a <> b AND b.genere = a.genere
        AND NOT (a<-[:MEMBER]-( )-[:MEMBER]->b)
        AND (e1_begin.ordinal <= e2_begin.ordinal
            AND e1_end.ordinal >= e2_begin.ordinal
            OR e1_begin.ordinal <= e2_end.ordinal
            AND e1_end.ordinal >= e2_end.ordinal
            OR e2_begin.ordinal <= e1_begin.ordinal
            AND e2_end.ordinal >= e1_end.ordinal)
        RETURN b
    """)
```

Como podemos ver esta consulta es un python template, esto nos permite re-usarla para los diferentes tipos de eventos simplemente cambiando \$relation por el nombre de la relación del evento con la persona.

Aprovecharemos esta consulta para explicar como funciona cypher, el lenguaje de consulta que usa el motor de Neo4j.

Primero de todo con la clausula START indicamos el nodo de partida de la consulta. Después

con MATCH indicamos el recorrido que ha de realizar la consulta en el grafo, nombrando los nodos y relaciones que nos interesen para luego filtrar en el WHERE o retornar en el RETURN. En este caso en la primera línea del MATCH estamos pidiendo todas las personas que tengan eventos que compartan la localización. El asterisco nos hará una búsqueda recursiva en todas las subsecciones del nodo AddressComponent, pero no en sus supersecciones, por la limitación que aplica la arista. Después encontrara las fechas en las que se realizaron estos eventos. Una vez obtenidos estos datos es trivial encontrar los nodos que potencialmente son iguales.

Dado que el objetivo del proyecto no era encontrar un algoritmo especialmente bueno para encontrar similitudes por ahora se ha estimado que dos personas son similares si comparten un número determinado de eventos potencialmente iguales y sus atributos como persona son similares. Lógicamente esto se podría mejorar aplicando heurísticos más complejos y que tuvieran en cuenta más cosas.

A pesar de esta última puntualización se considera que la parte más importante de la funcionalidad está cubierta y da resultados que en su mayor parte son potencialmente buenos. Evidentemente el decidir si dos personas finalmente son las mismas es decisión del usuario.



## 11. Pruebas

Para los test se han usado dos herramientas, el framework de testing que proporciona django rest framework sobre el framework de testing de django y POSTMAN. Todos los test que se han realizado han sido de integración.

### 11.1. Django rest framework test suit

Para realizar test de integración automatizados se ha usado la suit que proporciona django rest framework, esta suit consta de la clase `APITestCase`, que permite ejecutar los test con el comando incluido en el `managment.py`, `managment.py` es una herramienta que proporciona django que permite programar tareas que se ejecutan mediante el comando `./managment.py command` en la terminal, para este caso concreto podremos ejecutar todos los test mediante el comando `./management.py test`, luego podemos concretar el test que queremos ejecutar añadiendo `test.nombre del test`. Los test implementados con la clase `APITestCase` constan de dos operaciones principales, `setUp` y `tearDown`, `setUp` se ejecuta antes de cada test y se puede usar para establecer el entorno con los datos deseados y `tearDown` se ejecuta después de cada test y podemos devolver la el entorno a su estado inicial, para no crear efectos de lateralidad entre los test (i.e. que la ejecución de un test afecte a al resultado de otro).

Para inicializar los test con diferentes datasets que prueben el mayor numero de pruebas posibles se ha logrado creando la clase `setup_env`. En esta clase se encuentran todos los enviroments que los test puedan necesitar, esta clase se ha creado para mejorar la reusabilidad y escalabilidad de los test.

A continuación se adjunta el código de un test de integración como ejemplo.

```
class endpointsTestCase(APITestCase):

    def setUp(self):
        self.setup = setup()
        self.setup.clean_up()
        self.setup.setup()

    def tearDown(self):
        self.setup.clean_up()

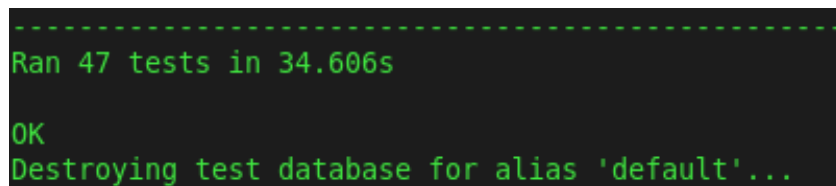
    def test_get_person_serach(self):
        '''
        search person
        '''
        url = reverse(
            'person-detail',
```

```
        kwargs={'id ': self.setup.person1.id}
    )
    url += 'search/'
    response = self.client.get(
        url,
        format='json ',
        HTTP_AUTHORIZATION=self.setup.token_bearer
    )
    print response
    self.assertTrue(status.is_success(response.status_code))
```

Este caso es un test muy simple, comprueba que realmente el endpoint retorna el status\_code esperado, en este caso un 201.

Estos tipos de test de tan alto nivel y tan poco pormenorizados en lenguajes no compilados como python son de gran importancia ya que la mayoría de fallos surgen en tiempo de ejecución. Lógicamente igual que se testan los endpoints también se testan las funciones importantes y se comprueba que retornen el resultado esperado según un setup determinado.

Una vez pasados los test vemos el mensaje de la figura 26



```
-----
Ran 47 tests in 34.606s

OK
Destroying test database for alias 'default'...
```

Figura 26: Test realizados con la django



## 11.2. POSTMAN

Por otro lado se han realizado test a más alto nivel con la herramienta POSTMAN, con esta herramienta podemos hacer llamadas a nuestra API simulando que somos un tercero que la esta usando. POSTMAN también proporciona herramientas de automatización de los test mediante las colecciones de llamadas.

Primero nos fijaremos en las colecciones, nos aparecen en el panel lateral y es donde se almacenaran nuestros test (Figura 27).

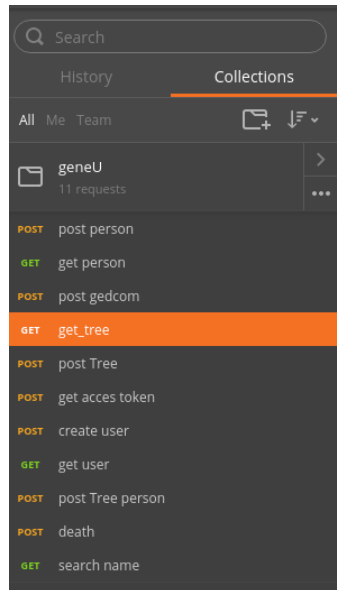


Figura 27: Vista de las colecciones en la interfaz de POSTMAN

En la figura 28 podemos ver como se ven los test, estos son ejecutados con la *Send* o cuando ejecutamos todos de manera automática.

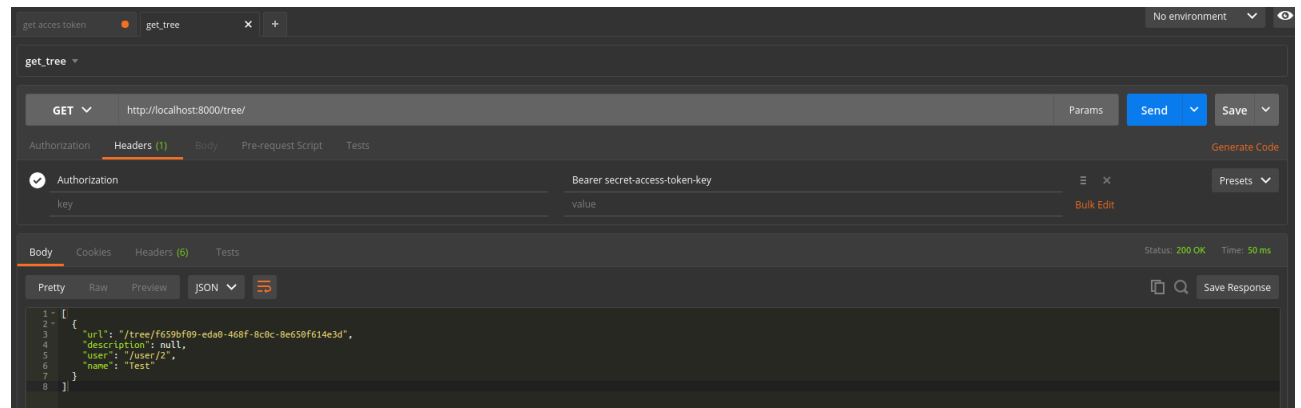


Figura 28: Vista de un test en la interfaz de POSTMAN

Para facilitar la ejecución de los test con POSTMAN se ha creado un nuevo comando en el `managment.py`, herramienta que hemos visto en el apartado anterior 11.1, con el fin de primero limpiar todo el envirmoent y luego inicilizarlo con un dataset. Este es el codigo:

```
class Command(BaseCommand):
    help = "clean all nodes and relations of the db"

    def add_arguments(self, parser):
        parser.add_argument('-s', '--set', nargs='?', help='...')

    def handle(self, *args, **options):

        subprocess.call(["python", "manage.py", "clean-neo-db"])
        subprocess.call(["python", "manage.py", "setup-loc-environ"])
        subprocess.call(["python", "manage.py", "setup-date-environ"])

        subprocess.call(["python", "manage.py", "reset-db"])
        subprocess.call(["python", "manage.py", "migrate"])
        subprocess.call([
            "python", "manage.py", "createsuperuser",
            "--username=admin", "--email=admin@geneu.com"
        ])
        set_nj4 = ["python", "manage.py", "set-neo4j"]
        if options['set']:
            set_nj4 += ["-s", options['set']]
```

```
subprocess.call(set_nj4)
```

**clean\_neo\_db** : Wipea la base de datos de neo4j

**setup\_loc\_environ** : Efectúa las inicializaciones pertinentes para poder trabajar con el modulo.

**setup\_date\_environ** : Efectúa las inicializaciones pertinentes para poder trabajar con el modulo.

**reset\_db** : Wipea la base de datos sql.

**migrate** : Inicializa la base de datos sql para poder trabajar con ciertos módulos que lo requieran.

**createsuperuser** : Crea un superusuario.

**set\_neo4j** : Inicializa la base de datos neo4j con un juego de pruebas.

\*Los comandos `clean_neo_db`, `setup_loc_environ`, `setup_date_environ`, `set_neo4j` también se han tenido que crear para poder llevar a acabo todo el proceso. Los demás vienen incluidos en Django.



## 12. Conclusiones

Una vez concluidas todas las etapas del proyecto se expondrán las conclusiones a las que se ha llegado. Se agruparán en diferentes apartados para unificar las diferentes temáticas que se quieren abordar.

### 12.1. Cambios en la planificación

Si bien se ha podido seguir la planificación en términos generales, sobretodo en las primeras etapas, donde se planteaba el proyecto, llegada la fase de implementación se ha observado que muchos objetivos que se habían establecido en primera estancia, estaban lejos de poderse conseguir, ya que se había condensado mucho trabajo en los *sprints*, por ello se tubo que hacer un replanteamiento y excluir el desarrollo de un *front-end* a favor de trabajar más en profundidad los otros aspectos del proyecto. Dada la naturaleza ágil de la metodología escogida y al tratarse de un proyecto donde el objetivo era el aprendizaje y establecer variaciones en los *sprints* no suponía un gran impacto, se replantearon y son los que ahora salen reflejados en la planificación final 4, esta modificación una vez terminado el proyecto se ha valorado positivamente dado que se han podido concentrar los esfuerzos en trabajar mejor otros aspectos del proyecto. Por otro lado se había estimado menos trabajo del finalmente realizado a la memoria, lo que a conllevado llegar muy justos a la fecha de entrega, esto se atribuye a las variaciones realizadas sobre la documentación inicial y a la profunda mejora de esta.

### 12.2. Objetivos académicos

Para desglosar de manera clara como se han cumplido cada uno de los objetivos establecidos se explicara uno por uno.

#### 12.2.1. CES1.1

**Desarrollar, mantener y evaluar sistemas y servicios software complejos o críticos.**

Este objetivo se da por alcanzado, dado que una gran parte del proyecto a consistido en desarrollar toda la plataforma que daba acceso al repositorio, de forma que esta se adecuase a estándares usados en el ámbito profesional para dar servicio a infraestructuras con altos requerimientos.

#### 12.2.2. CES1.2

**Dar solución a problemas de integración en función de las estrategias, de los estándares y tecnologías disponibles.**

Uno de los objetivos principales del proyecto ha sido integrar tecnologías ya existentes basadas en

estándares que fuesen lo más actuales posibles, ejemplo de ello es como se ha usado tecnologías como Django rest framework, RabbitMQ, Celery, OAuth2 Toolkit, Neomodel, entre otras.

### 12.2.3. CES1.4

*Desarrollar, mantener y evaluar servicios y aplicaciones distribuidas con soporte de red.*

Si bien no se ha llegado a hacer un *deployment* final del sistema en una arquitectura en red, este está pensado para funcionar de forma distribuida. Prueba de ello es el uso del patrón *message broker* 9.1.2, que finalmente ha sido desarrollado con éxito.

### 12.2.4. CES1.5

*Especificar, diseñar y implementar bases de datos.*

Una de las tareas que ha motivado este proyecto precisamente ha sido, como bien el título indica, crear un repositorio de árboles genealógicos en bases de datos NoSQL. Para ello se ha trabajado sobre las bases de datos orientadas a grafos, estudiando sus propiedades, que ventajas nos aporta frente a otros modelos y de que forma se tienen que diseñar para conseguir una mejor optimización teniendo en cuenta sus cualidades. Esto se puede comprobar a lo largo de la memoria:

**Introducción** : Bases de datos orientadas a grafos (BDOG)2.2.2

**Especificación** : Modelo conceptual de datos8.2

**Diseño** : Diseño del modelo de datos 9.5

**Implementación** : FALTA

### 12.2.5. CES1.6

**Administrar bases de datos (CIS4.3)**

Consecuencia del punto anterior se ha realizado tareas de administración tanto en la base de datos orientada a grafos, como en la SQL encargada de la persistencia de usuarios.

### 12.2.6. CES1.9

**Demostrar comprensión en la gestión y gobierno de sistemas software.**

### 12.2.7. CES2.2

**Diseñar soluciones apropiadas en uno o más dominios de la aplicación, usando métodos de ingeniería del software que integren aspectos éticos, sociales, legales y económicos.**

cos.

### 12.3. Evoluciones futuras

Si bien el objetivo del desarrollo de este proyecto tenía puramente fines educativos y no se espera que se realicen dichas evoluciones, se plantearan aspectos que hubiesen estado bien poder llevar a cabo, para mejorar la calidad del sistema y profundizar más en las tecnologías y conceptos utilizados.

#### **Desarrollo modulos para la integración de neomodel en Django rest framework :**

Si bien en el proyecto se han integrado las dos tecnologías se ha realizado de una manera ad-hoc a las necesidades del proyecto, para mejoras futuras se propone una refactorización de los serializadores. Creando un serializador al estilo *ModelSerializer* que proporciona el framework y crea los serializadores en base a nuestros modelos mediante el uso de metaprogramción, pero para Neomodel.

#### **Ampliación del parser GEDCOM :**

Actualmente las soluciones de parsers GEDCOM open source para python están muy poco maduras, se propone como mejora futura la participación en una proyecto open source para mejorar estos parsers y incorporarlos al proyecto, con el objetivo de poder subir GEDCOM extrayendo más información.

#### **Mejora del sistema montado con Celery y RabbitMQ :**

Si bien ahora se adecua a las necesidades y ofrece las fundacionales esperadas, se plantea para un trabajo futuro el guardar el estado de las tareas en una base de datos para que los usuarios puedan saber a través de la API si sus tareas están en cola, o ya están siendo procesadas.

#### **Desarrollo de un front-end y/o aplicación móvil :**

Se plantea para evoluciones futuras el desarrollo de un front-end y/o aplicación móvil, que use la API.

### 12.4. Valoración personal





## 13. Referencias

### Referencias

- [1] Michael B. Jones Dick Hardt. The oauth 2.0 authorization framework, 2012.
- [2] Ryan Heaton. About gedcom, 2014.
- [3] Alekh Jindal. Benchmarking graph databases, 2013.
- [4] Neo4j. Hyperedges.
- [5] RelcyEngineering. Data fusion algorithms, 2016.
- [6] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013.
- [7] Marko A Rodriguez and Peter Neubauer. The graph traversal pattern. *arXiv preprint arXiv:1004.1001*, 2010.
- [8] Richard Stallman. Java trap, 2004.
- [9] InfoGrid Team. Neo4j and infogrid, 2009.
- [10] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.
- [11] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in action*. Manning, 2015.