Universitat Oberta de Cataluyna

TFG - Aplicaciones y sistemas distribuidos · 2016-17 semestre 2

Profesor: Félix Freitag

Desarrollo de un gestor de tareas autoalojado y distribuido

Trabajo de fin de grado

Adrián González Blanco

Grau d'Enginyeria Informàtica

26 de junio de 2018

Esta obra está bajo una licencia Creative Commons «Reconocimiento-NoCommercial-CompartirIgual 3.0 España».

Ficha del trabajo final

Título del trabajo	Desarrollo de un gestor de tareas autoalojado y distribuido
Nombre del autor	Adrián González Blanco
Nombre del consultor	Félix Freitag
Fecha de entrega (mm/aaaa)	06/2018
Área del Trabajo Final	Aplicacions i sistemes distribuïts
Titulación	Grau d'Enginyeria Informàtica

Resumen del trabajo:

En los últimos años se ha puesto en evidencia los riesgos que para los usuarios supone el uso de las aplicaciones en la nube ofrecidas por los gigantes de la industria. Sin embargo, dado que estas empresas controlan los servicios mayoritarios, resulta difícil cesar en su uso. Como respuesta a esta problemática, ha surgido un movimiento que aboga por el autoalojamiento de aplicaciones web, de forma que los usuarios son dueños de sus propios datos.

En este contexto, en el presente trabajo se ha desarrollado un gestor de tareas autoalojado y distribuido. Este se compone de un servidor web desarrollado sobre Django; una API, construida con Django REST Framework, y un cliente CLI, implementado con Python. Este último permite el trabajo desconectado y la posterior sincronización mediante una llamada a la API.

En el centro del desarrollo se encuentra el protocolo de sincronización, que adopta una estrategia híbrida entre la sincronización de estados y la de operaciones. Así, en cada sincronización, el cliente envía las operaciones que ha registrado localmente y el servidor responde haciéndole llegar todos aquellos objetos que hayan sido actualizados desde la última sincronización de dicho cliente, incluyendo aquellos fruto de la aplicación de las operaciones enviadas.

Para el despliegue del servidor se ha realizado mediante contenedores Docker. Con ello, se facilita la orquestación automática del servidor web, así como el proxy inverso Nginx y la instancia PostgreSQL que le dan servicio. Por su parte, para la distribución del cliente se ha utilizado `pip`, `setuptools` y `venv`.

***Abstract*:**

In recent years, the risks posed to users by the use of cloud applications offered by industry giants have become evident. However, as these companies control the majority of these services, it is difficult to stop using them. In response to this situation, a movement has emerged that advocates self-hosting of web applications, so that users own their own data.

In this context, a self-hosted and distributed task manager has been developed in the present work. This consists of a web server developed on Django; an API, built with Django REST Framework, and a CLI client, implemented with Python. The latter allows disconnected work and subsequent synchronization via an API call.

At the heart of development is the synchronization protocol, which adopts a hybrid strategy between state and operations synchronization. Thus, in each synchronization, the client sends the operations it has registered locally and the server responds by sending all those objects that have been updated since the last synchronization of the client, including those resulting from the application of the operations sent.

The server was deployed using Docker containers. This facilitates the automatic orchestration of the web server, as well as the Nginx reverse proxy and the PostgreSQL instance that serve it. `pip`*,* `setuptools` *and* `venv` *have been used for client distribution.*

Palabras clave:

sistemas distribuidos, web, api, python, selfhosting, tareas

Índice general

Índice general v

Índice de figuras vii

1 Introducción 1
1. Contexto y justificación del trabajo . . . 1
1.1. Motivaciones . . . 1
2. Objetivos del Trabajo . . . 2
2.1. Objetivos generales . . . 2
2.2. Objetivos específicos . . . 2
3. Enfoque y método seguido . . . 3
4. Planificación del Trabajo . . . 3
4.1. Descomposición del alcance . . . 3
4.1.1. Tareas . . . 3
4.1.2. Hitos . . . 4
4.1.3. Cronograma . . . 4
5. Breve sumario de productos obtenidos . . . 7
6. Breve descripción de los otros capítulos de la memoria . . . 7

2 Estructura del trabajo y artefactos obtenidos 9

3 Servidor web y cliente HTML 11
1. Diseño . . . 11
1.1. Elección de tecnología . . . 11
1.2. Diseño conceptual . . . 13
2. Implementación . . . 16
3. Resultado . . . 17
4. Despliegue . . . 19

4 API 21
1. Diseño . . . 21
1.1. Protocolo de sincronización . . . 22
2. Implementación . . . 26
3. Resultado . . . 26
4. Despliegue . . . 27

5 Cliente distribuido 31
1. Diseño . . . 31

2. Implementación 32
3. Resultado 33
4. Despliegue 34

6 Conclusiones **37**
1. Seguimiento de la planificación y metodología 37
2. Logro de objetivos 38
3. Conclusiones y lecciones aprendidas 38
4. Futuras líneas de trabajo 39

Bibliografía **43**

Índice de figuras

11. Diagrama de Gantt del proyecto 6

31. Esquema de funcionamiento básico de Django 13
32. Diagrama de clases UML 14
33. Cliente HTML: vista de los próximos 7 días 18
34. Cliente HTML: administración de usuarios 18
35. Logotipo de la aplicación 19
36. Despliegue del servidor en producción mediante Docker Compose 20

41. Estructura del mensaje enviado por el cliente durante una sincronización 25
42. Estructura del mensaje de respuesta del servidor durante una sincronización 25
43. Interfaz web de la API: raíz 27
44. Interfaz web de la API: vista de lista de proyectos 28
45. Interfaz web de la API: vista de un proyecto 29

51. Esquema resumen de la arquitectura funcional del sistema 33
52. Ejemplo de funcionamiento y aspecto del cliente distribuido 34

1

Introducción

1. Contexto y justificación del trabajo

El trabajo realizado consiste en el desarrollo de un gestor de tareas web, libre y autoalojado (*self-hosted*), junto con un cliente distribuido. La aplicación servidor dispone de una API para facilitar su integración con otras aplicaciones y dar servicio al cliente desarrollado. El cliente y la API implementan un algoritmo de sincronización que permite que el primero trabaje fuera de línea de forma totalmente operativa y adquiera posteriormente la consistencia.

Dada el área en la que se sitúa este trabajo, se ha hecho especial énfasis en las problemáticas propias de los sistemas distribuidos que afectan al sistema planteado, en detrimento de aspectos más centrados en el diseño y la programación de aplicaciones web con sofisticadas interfaces. Así, la mayor parte del tiempo empleado se ha destinado a la solución de los clásicos problemas de consistencia y disponibilidad, y no a aspectos más estéticos.

1.1. Motivaciones

Las motivaciones que guiaron la definición del proyecto responden a tres grandes tipos: formativas, funcionales y organizativas. En primer lugar, las motivaciones formativas responden a un deseo de descubrir, profundizar y practicar con tecnologías específicas de desarrollo (Django, Python, PostgreSQL, ZODB, HTML, CSS o Javascript) y de despliegue (Docker), la mayoría prácticamente inéditas durante el grado. También se ha buscado poner en práctica algunos de los problemas de sistemas distribuidos introducidos en la asignatura homónima de la titulación. Por último, también ha influido en el trabajo el afán personal por probar nuevos paradigmas de desarrollo como el TDD (*test-driven development*), las arquitecturas basadas en MVC (Modelo-vista-controlador) y los sistemas ORM (*Object-relational Mapping*).

En segundo lugar, la elección del producto final (un gestor de tareas autoalojado) responde a la reciente corriente que aboga por una descentralización de Internet y en busca de un modelo distribuido y federado de computación. En este último, los datos volverían a pertenecer a los usuarios, en lugar de a las grandes corporaciones que dominan el sector. En este contexto, no se han encontrado aplicaciones existentes que se adecuen a los requisitos funcionales y de simplicidad de la aplicación desarrollada en este trabajo. Por lo tanto, uno de los productos resultantes del proyecto serán necesariamente los *scripts* de despliegue, que facilitarán la puesta en funcionamiento de nuevas instancias del servidor.

Por último, con el trabajo también se trató de contribuir al desarrollo y práctica de las habilidades organizativas. Esto pasa por la definición adecuada del proyecto (detallar sus objetivos y delimitar su alcance), la planificación el trabajo (descomponerlo en tareas e hitos y enmarcarlo en el tiempo) y un correcto seguimiento y control del progreso. Como resultado de estas labores se han generado diferentes artefactos documentales, principalmente en forma de PAC 2 y PAC 3.

2. Objetivos del Trabajo

Las motivaciones antes indicadas se tradujeron en una serie de objetivos. Estos pueden dividirse en dos tipos, según su nivel de generalidad. A continuación se detalla cada uno de ellos. El orden es indicativo de la prioridad o el peso cada objetivo.

2.1. Objetivos generales

1. Llevar a cabo de forma satisfactoria un TFG, siguiendo la metodología propuesta y generando la documentación requerida.
2. Desarrollar un gestor de tareas web autoalojado.
3. Profundizar en las problemáticas de los sistemas distribuidos.
4. Aprender, profundizar y practicar con tecnologías de desarrollo específicas como Django, Python, PostgreSQL, SQLite, HTML, CSS o Javascript.
5. Aprender, profundizar y practicar con tecnologías de despliegue como Docker.
6. Descubrir y practicar la metodología TDD (*test-driven development*).
7. Implementar un sistema mediante el patrón MVC (Modelo-vista-controlador).
8. Poner en práctica las técnicas de gestión de proyectos.

2.2. Objetivos específicos

1. Seguir la metodología propuesta para el TFG de acuerdo a las indicaciones del profesor responsable.
2. Elaborar adecuadamente los artefactos documentales propios del TFG (Memoria, Presentación Virtual, Cuestionario de autoevaluación y resultados del proyecto) en forma y tiempo.
3. Desarrollar una aplicación web de gestión de tareas mediante Django y PostgreSQL.
4. Desarrollar una API REST.
5. Desarrollar un cliente, mediante Python y un sistema de bases de datos, que pueda funcionar de forma desconectada y se comunique con el servidor a través de la API para la sincronización.
6. Elaborar una documentación de la aplicación.
7. Desarrollar los *scripts* de despliegue de la aplicación.
8. Desplegar las instancias de demostración en un servidor VPS.
9. Realizar una formación inicial en las tecnologías de desarrollo: Django y las extensiones necesarias.
10. Realizar una formación inicial en las tecnologías de despliegue: Docker.
11. Seguir el paradigma de TDD (*test-driven development*).
12. Poner en práctica el patrón MVC (Modelo-vista-controlador).
13. Planificar, seguir y controlar un proyecto de desarrollo de *software*.

3. Enfoque y método seguido

La estrategia seguida para el desarrollo de la aplicación distribuida ha sido la de realizar un desarrollo desde cero, sin reaprovechar o extender soluciones existentes. Sin embargo, y como cabría esperar de una aplicación de esta índole, este desarrollo se ha apoyado en varias bibliotecas y *frameworks* existentes. Con ello, se ha buscado contribuir al logro de dos de los objetivos del trabajo: el aprendizaje de tecnologías y metodologías de desarrollo, en ocasiones introducidas por los propios *frameworks*, y tratar de ejecutar el proyecto en tiempo y plazo, considerando los recursos limitados.

En cuanto a la organización del trabajo, para su desarrollo se decidió seguir una metodología de TDD (*test-driven development*), adoptando un ciclo de desarrollo iterativo. Con ello, se buscó disponer siempre de entregables (más o menos desarrollados) y guiar el proceso de desarrollo por las pruebas (unitarias y funcionales), de forma que se garantice la estabilidad de los productos resultantes y se reduzca la fricción intrínseca al despliegue.

4. Planificación del Trabajo

4.1. Descomposición del alcance

Para facilitar la organización, desarrollo y seguimiento del proyecto se descompuso su alcance (definido a partir de la descripción y los objetivos planteados) en un conjunto de tareas y de hitos que habrían de alcanzarse durante el desarrollo del trabajo.

4.1.1. Tareas

A continuación, se presentan las tareas definidas agrupadas por fases:

- **Fase de *Definición inicial***: planteamiento inicial del tema del TFG.
  - Elección del tema: selección del tema del TFG.
  - Aprobación del tema: ratificación del tema del TFG por parte del profesor responsable.
  - Definición inicial: primera propuesta de objetivos y alcance del proyecto.
- **Fase de *Planificación***: elaboración del plan de trabajo.
  - Descripción, objetivos y alcance: concreción de descripción, objetivos y alcance definitivos.
  - Tareas e hitos: descomposición del trabajo a realizar en tareas e hitos.
  - Cronograma: planificación de las tareas e hitos en el tiempo.
- **Fase de *Desarrollo***: programación de servidor y cliente de la aplicación.
  - Formación inicial: estudio de las herramientas a utilizar y primeras pruebas.
  - Diseño de la arquitectura: descomposición y organización en componentes y algoritmos.
  - Preparación del entorno: instalación y configuración de las herramientas de desarrollo.
  - Servidor estándar: desarrollo de la aplicación servidor con funcionalidades estándar.
  - API: desarrollo de una API para la interacción con el servidor.
  - Cliente estándar: desarrollo de un cliente CLI distribuido que se comunique con el servidor a través de la API.

    - *Scripts* de despliegue: automatización del despliegue de instancias de la aplicación.
    - Servidor avanzado: desarrollo de funcionalidades avanzadas en el servidor (ver apartado 4).
- **Documentos y artefactos**: realización de los entregables resultado del proyecto.
    - Elaboración del *Plan de trabajo*: documento que conformó la PAC 1.
    - Elaboración de la *Memoria*: documento troncal del TFG.
    - Elaboración de la documentación: documentación de la aplicación desarrollada.
    - Despliegue de instancias demo: instancias de demostración de la aplicación.
    - Elaboración del *Informe de autoevaluación*: valoración del trabajo.
    - Elaboración de la *Presentación*: presentación virtual del trabajo realizado.
- **Entregas**: elaboración de la documentación a presentar para cada entrega de la asignatura.
    - Elaboración PAC 2: documentación de la segunda PAC.
    - Elaboración PAC 3: documentación de la tercera PAC.
    - Elaboración Entrega final: documentación de la entrega final del TFG.

4.1.2. Hitos

Como resultado de este trabajo se generaron una serie de productos, incluidos inicialmente en forma de *hitos* en la planificación. Estos productos pueden dividirse en dos tipos: los resultados propios del proceso de desarrollo (código fuente, documentación e instancias de demostración) y los artefactos documentales propios de cualquier TFG (el Plan de Trabajo, la Memoria, la Presentación Virtual y el Cuestionario de Autoevaluación). Asimismo, se han elaborado sendos informes de seguimiento para las dos últimas PAC.

Así, los siguientes fueron los hitos a alcanzar durante el desarrollo del trabajo:

1. Plan de trabajo
2. Documentación de la aplicación
3. Instancias de demostración
4. Memoria del TFG
5. Informe de autoevaluación
6. Presentación virtual
7. PAC 1
8. PAC 2
9. PAC 3
10. Entrega final

4.1.3. Cronograma

La figura 11 muestra un cronograma (diagrama de Gantt) con la planificación temporal de las principales tareas e hitos en las que se ha descompuesto el proyecto. Se muestran en verde los hitos y en rojo las fechas límite de entrega oficiales. La tarea *Servidor*

avanzado se presenta en color gris para diferenciarla de las demás, dado su carácter más opcional y flexible.

Cabe indicar que esta planificación tiene en cuenta una serie de posibles limitaciones en la disponibilidad temporal. Estas responden a asuntos familiares[1] que de producirse habrían restringido de forma muy notable la dedicación al proyecto durante las semanas cercanas al comienzo del mes de mayo[2].

En consecuencia, para mitigar este riesgo y poner en peligro el éxito de la planificación se optó por separar la funcionalidad de la aplicación servidor en dos grandes bloques, siendo el alcance y cumplimiento de este último de carácter orientativo y opcional. Así, en caso de concretarse dicha restricción en el tiempo disponible, solo se debería ver afectado el grado de cumplimiento del bloque opcional.

Estos bloques o tareas se han denominado *Servidor estándar* y *Servidor avanzado* respectivamente, siendo el último el de cumplimiento opcional. Los nombres responden al nivel de funcionalidad que alcanzaría la aplicación de cumplirse estos. En concreto, las características contempladas para cada uno de los niveles funcionales fueron las siguientes:

Servidor estándar:

1. Multiusuario: cada usuario dispone de sus propios proyectos y tareas.
2. Multiproyecto: poder definir varios proyectos/listas de tareas.
3. Prioridades en tareas.
4. Etiquetas en tareas.
5. Fechas de vencimiento en tareas.
6. Jerarquizar tareas: definir unas como subtareas de otras (a modo de árboles).
7. Jerarquizar proyectos: incluir unos dentro de otros.
8. Botón para añadir una tarea rápidamente.
9. Proyecto predeterminado "Bandeja de entrada": donde registrar tareas que se clasificarán posteriormente, algo necesario para seguir métodos de organización como GTG (*Getting Things Done*).
10. Vistas resumen predefinidas de las tareas: para *hoy* y para los *próximos 7 días*.
11. Añadir comentarios a las tareas.
12. Ordenar las tareas de un proyecto: por descripción, fecha, prioridad, etc.
13. Compartir proyectos/listas entre usuarios.

Servidor avanzado:

1. Interfaz dinámica (SPA, *Single page application*): evitar la recarga de la página tras cada modificación.
2. Importar y exportar tareas de proyectos, usando ficheros de texto.
3. Sintaxis inteligente para especificar los parámetros de una tarea al crearla: por ejemplo, `"Limpiar el coche #mi_proyecto @mi_etiqueta !!1 12:30"`.
4. Atajos de teclado: para añadir, mover o modificar tareas.
5. Añadir adjuntos a las tareas.

[1] El nacimiento de mi hija.
[2] La fecha probable de parto.

6. Tareas recurrentes: se repiten en el tiempo a una frecuencia determinada o un día determinado cada año.
7. Vistas resumen personalizadas: mediante filtros sobre los atributos de tareas.
8. Búsqueda de tareas: mediante preindexado.
9. Recordatorios de vencimientos: por mail, SMS, etc.

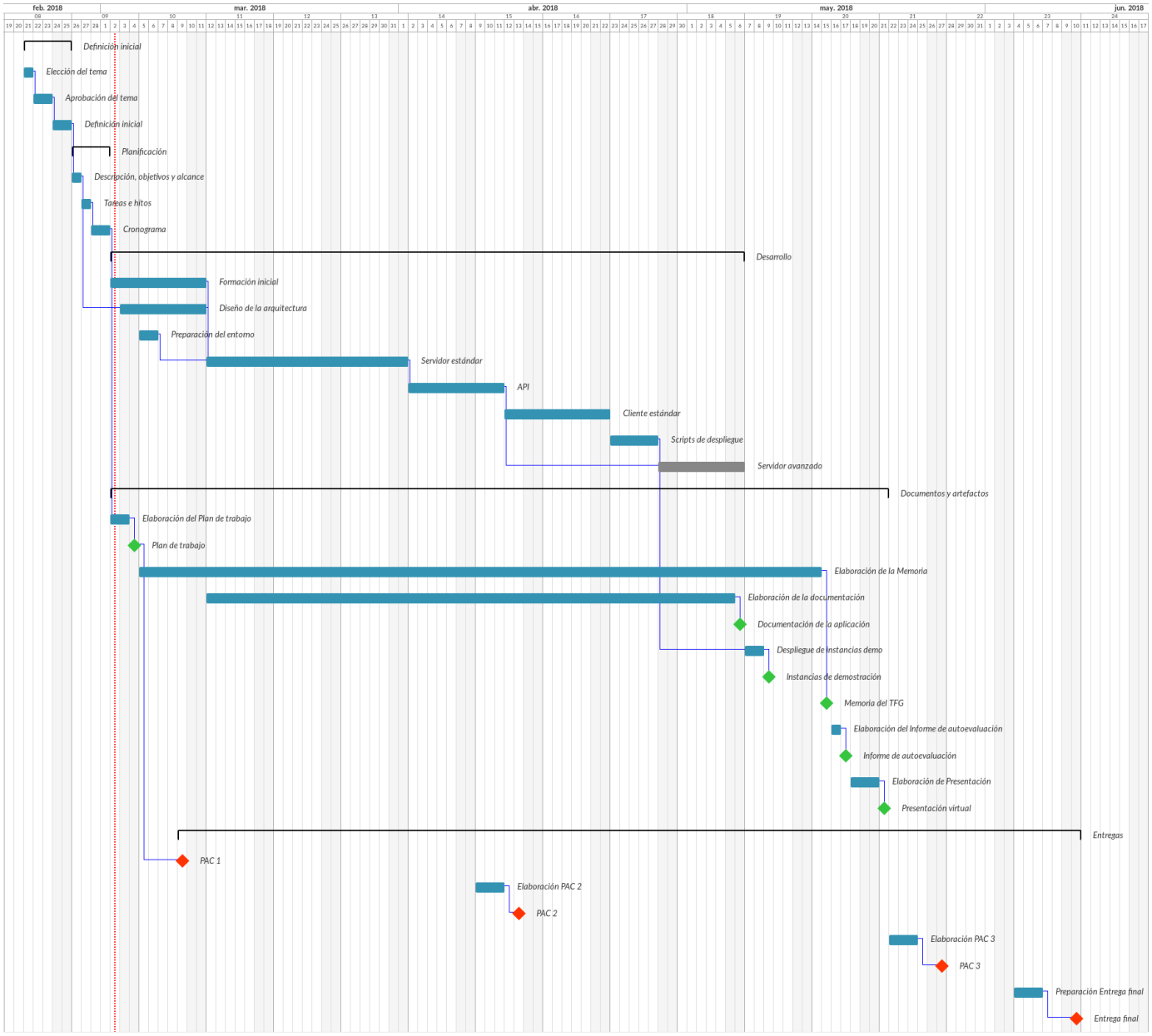

Figura 11. Diagrama de Gantt del proyecto

5. Breve sumario de productos obtenidos

Como resultado de este trabajo se han obtenido tres productos funcionales principales, que se enumeran a continuación:

1. **Servidor web**

   Un servidor totalmente funcional, incluyendo las tres capas habituales (interfaz, lógica y datos), que ofrece un cliente HTML para que el usuario pueda realizar la gestión de las tareas.

2. **API**

   Este componente, integrado en el servidor web, ofrece los servicios de gestión de tareas a otros programas y procesos mediante una interfaz programática. Además, la API dispone de una interfaz web para facilitar su exploración y uso.

3. **Cliente distribuido**

   Un sencillo cliente en línea de comando (CLI) que puede funcionar de forma desconectada y sincronizarse en cualquier momento con el servidor.

Asimismo, los aspectos relacionados con el despliegue se encuentran distribuidos dentro proyecto y se introducen al final de los bloques relacionados con cada uno de estos tres productos principales.

6. Breve descripción de los otros capítulos de la memoria

Los capítulos que siguen al presente se adentran en los detalles de diseño, implementación y resultados obtenidos de cada uno de los productos funcionales mencionados anteriormente. Dado que para todos ellos se ha seguido una misma estructura, sus capítulos específicos están precedidos de uno adicional de carácter introductorio en el que se exponen aspectos transversales a los tres bloques que le siguen. Así, la memoria consta de cuatro capítulos específicos:

2. *Estructura del trabajo y artefactos obtenidos*.
3. *Servidor web y cliente HTML*.
4. *API*.
5. *Cliente distribuido*.

2

Estructura del trabajo y artefactos obtenidos

Los resultados del trabajo, así como el esfuerzo dedicado a su desarrollo, se orquestan en tres grandes apartados o componentes funcionales, que se corresponden, además, con diferentes artefactos. Es son:

1. Servidor y cliente web.
2. API.
3. Cliente distribuido.

Los siguientes apartados del presente documento se centrarán en cada uno de estos apartados. En particular, se descompondrán en cuatro subapartados, correspondientes a las distintas etapas del proceso de desarrollo:

1. **Diseño**

   En este subapartado se expondrán y justificarán las principales decisiones en cuanto a la definición de los componentes tecnológicos y los esquemas arquitectónicos que dirigieron la construcción del componente.

2. **Implementación**

   Este bloque explica las problemáticas encontradas y las decisiones tomadas a la hora de desarrollar los programas y elementos necesarios para ofrecer la funcionalidad requerida.

3. **Resultado**

   En el tercer subapartado se muestran los resultados obtenidos y se proporcionan ciertas aclaraciones sobre el nivel de alcance funcional del componente, así como otros aspectos relacionados.

4. **Despliegue**

   Finalmente, el apartado de despliegue expone las herramientas de despliegue ofrecidas para la puesta en funcionamiento del componente.

3

Servidor web y cliente HTML

El servidor web es el elemento central del trabajo, ya que, además de ser el depositario de la gran mayoría del tiempo y esfuerzo empleados en el desarrollo, se trata de un componente vertebral sobre el que se articulan el resto: el cliente web es parte del propio servidor; la API funciona como un módulo complementario que se nutre de la funcionalidad del servidor, y el cliente distribuido, a su vez, consume los servicios de la API. Por todo ello, puede considerarse el componente más importante del sistema.

El cliente HTML se integra dentro del propio *framework* sobre el que se ha desarrollado el servidor web. Así, dada su estrecha relación con el servidor, se ha optado por incluirlo dentro del presente bloque. Sin embargo, hay que señalar que realmente se tratarían de componentes diferenciados, puesto que con otras arquitecturas el cliente web podría implementarse de forma independiente y solo comunicarse con el servidor a nivel de API, como sucede con el cliente distribuido que se ha desarrollado.

A renglón seguido, se presenta la información más relevante en lo que se refiere al diseño e implementación del servidor. También se ofrece una breve exposición de los resultados obtenidos y de las herramientas de despliegue ofrecidas para su puesta en funcionamiento.

1. Diseño

1.1. Elección de tecnología

Uno de los objetivos del proyecto es el aprendizaje de tecnologías de desarrollo específicas. En este sentido, se ha buscado un conjunto de herramientas que permitiesen afrontar el desarrollo del servidor, cliente HTML y cliente distribuido de la manera más eficiente posible. Esta eficiencia depende en gran medida del nivel de reaprovechamiento de las herramientas para los distintos ámbitos de desarrollo. Así, Python se mostró una alternativa muy atractiva, puesto que ofrece *frameworks* y herramientas para programación web, creación de aplicaciones cliente e incluso su despliegue.

Asimismo, puesto que Python es un lenguaje de alto nivel con un inmenso ecosistema de componentes libres, a priori facilitaría la tarea de desarrollo, al abstraer al programador de las labores más tediosas y repetitivas, y permitir así que sus esfuerzos se centren en construir la lógica de negocio. La principal crítica al uso de este tipo de lenguajes (dinámicos de alto nivel) es el rendimiento de la aplicación resultante, pero dado la naturaleza y los objetivos del proyecto se consideró que los beneficios superarían con creces a las desventajas de esta aproximación.

La otra gran alternativa considerada fue Javascript, que también se considera un lenguaje *fullstack*. Sin embargo, se rechazó en favor de Python, debido a que en este último ya se contaba con experiencia previa y además goza de una gran aceptación en el área de Inteligencia Artifical y Big Data, dos de las áreas con mayor crecimiento en la actualidad. Una vez escogido el lenguaje de programación, el siguiente paso fue elegir un *framework* web de entre los muchos disponibles. En este sentido, se optó por el de mayor arraigo y fama en el ecosistema Python: Django.

Django es un *framework* web que ofrece una enorme funcionalidad de partida, a diferencia de otros paquetes que proponen la funcionalidad mínima sobre la que ir construyendo modularmente la aplicación final. Estos últimos reciben el nombre de *microframeworks* y su máximo exponente en Python es Flask. Se optó por Django puesto que dispone de una gran documentación y existen multitud de libros relacionados. El aspecto definitivo que decantó la balanza fue la existencia de un libro que, además de adentrarse en Django, expone de forma práctica las prácticas y virtudes del TDD (*test-driven development*).

El patrón de arquitectura en el que se basa Django es el MVC (Modelo-vista-controlador) —aunque existe un cierto debate en cuanto a los nombres que recibe cada elemento dentro del *framework*—. De esta forma, los componentes sobre los que se articula Django son cuatro:

1. **Modelos**: definen las clases y atributos que representarán el dominio en el que se enmarca el problema.
2. **URLs**: dirigen las peticiones HTTP a las vistas adecuadas, de acuerdo a unos patrones en la ruta solicitada.
3. **Vistas**: consultan y manipulan los datos de las instancias de los modelos para generar respuestas a las peticiones HTTP recibidas.
4. **Plantillas**: facilitan la construcción de respuestas HTML por parte de las vistas.

A continuación, se presenta el flujo habitual que sigue una petición a un sitio web Django. La figura 31 lo resume gráficamente.

1. El gestor de URLs de Django recibe una petición HTTP y según la ruta a la que apunta invoca a la vista pertinente.
2. La vista, por su parte recibe la petición, acompañada de información contextual como las *cookies*, el usuario (si se había identificado en el sitio), etc.
3. Según la petición, la vista consulta el modelo (y lo manipula si fuese necesario) y construye la información necesaria para enviar a la plantilla.
4. A continuación, la vista ejecuta la plantilla, que devuelve el código HTML resultante.
5. Finalmente, la vista construye una respuesta HTTP con el contenido HTML y se la envía de vuelta al cliente.

Django permite trabajar con varios sistemas gestores de bases de datos, entre ellos SQLite, MySQL/MariaDB, PostgreSQL y Oracle. Para el entorno de producción de la aplicación servidor se ha optado por PostgreSQL, puesto que es el que ofrecía el mejor comportamiento y se adecuaba mejor a los objetivos, de acuerdo al pequeño estudio que se realizó. Para el entorno de desarrollo se prefirió utilizar SQLite, tal y como se recomienda en la propia documentación de Django. El motivo es que, al ser una base de datos empotrada en un único fichero, facilita sobremanera el mantenimiento, borrado e inicialización de los datos durante el desarrollo y las pruebas.

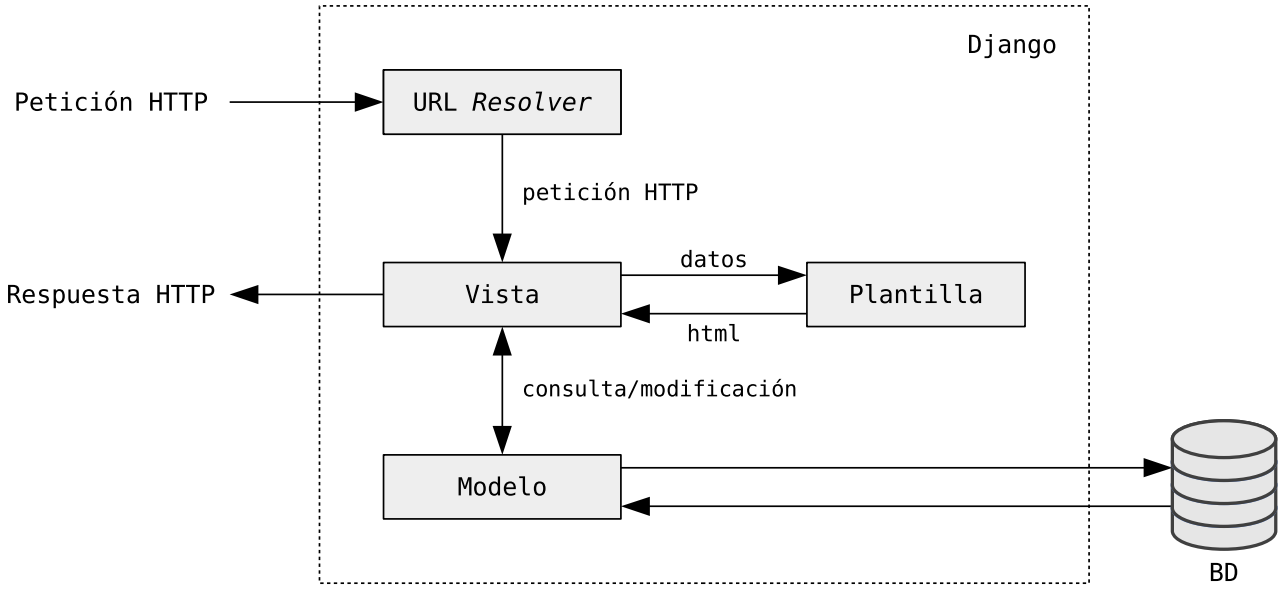

Figura 31. Esquema de funcionamiento básico de Django

Por último, para el diseño de las plantillas ha sido necesario emplear HTML, CSS y Javascript. Para simplificar esta tarea se ha utilizado el *framework* Bootstrap, que provee de herramientas que facilitan esta labor. Entre sus características se puede destacar que facilita el diseño de páginas web adaptativas, evita la mayoría de los problemas de compatibilidades entre navegadores, ofrece un estilo base moderno y elegante, dispone de una serie de componentes interactivos de serie que enriquecen la experiencia del usuario, etc. Bootstrap, a su vez, se apoya en otras herramientas como jQuery o SASS (*Syntactically Awesome Stylesheets*).

1.2. Diseño conceptual

Una vez determinada la tecnología a utilizar, se procedió al diseño conceptual de la aplicación. Naturalmente, dada la limitación de recursos (especialmente temporales) asociada a un trabajo de estas características, el alcance funcional de la aplicación dista de lo que cabría esperar de una solución profesional. En cualquier caso, se ha tratado de ser algo ambicioso y garantizar que el desarrollo y el resultado del trabajo sean suficientes para cumplir todos los objetivos propuestos.

El objetivo central que ha guiado todo el trabajo es el estudio, diseño y desarrollo de un sistema distribuido. En consecuencia, ha sido necesario dar un mayor peso específico al algoritmo de sincronización y a los procesos que de este se derivan. Así pues, el primer paso para el diseño del sistema fue el estudio de soluciones similares existentes. En particular, se consideraron tres gestores de aplicaciones distribuidos: Todoist , Wunderlist y Remember the Milk .

Además de poder estudiar el funcionamiento a nivel de usuario de cada aplicación, todas ellas disponen de una API pública y bien documentada. Gracias a ello, resultó mucho más sencillo deducir el funcionamiento interno de estos servicios. En particular, se ha optado por adoptar un esquema de funcionamiento inspirado principalmente en el encontrado en Todoist, ya que se ha considerado el más adecuado dadas las características de este proyecto. Esta decisión ha resultado crucial a la hora de moldear el resto de componentes del trabajo.

En línea con lo expuesto anteriormente, no se buscó igualar el nivel funcional de

una aplicación como las estudiadas, lo que resultaría absolutamente inviable. En su lugar, se optó por considerar solamente un pequeño subconjunto de las características habituales en estos productos. Esto ha permitido, entre otros aspectos, no tener que centrarse tan solo en el desarrollo del componente servidor y poder implementar tanto una pequeña API como un cliente distribuido, de forma que se ha obtenido una visión más amplia de este tipo de sistemas y de sus problemáticas.

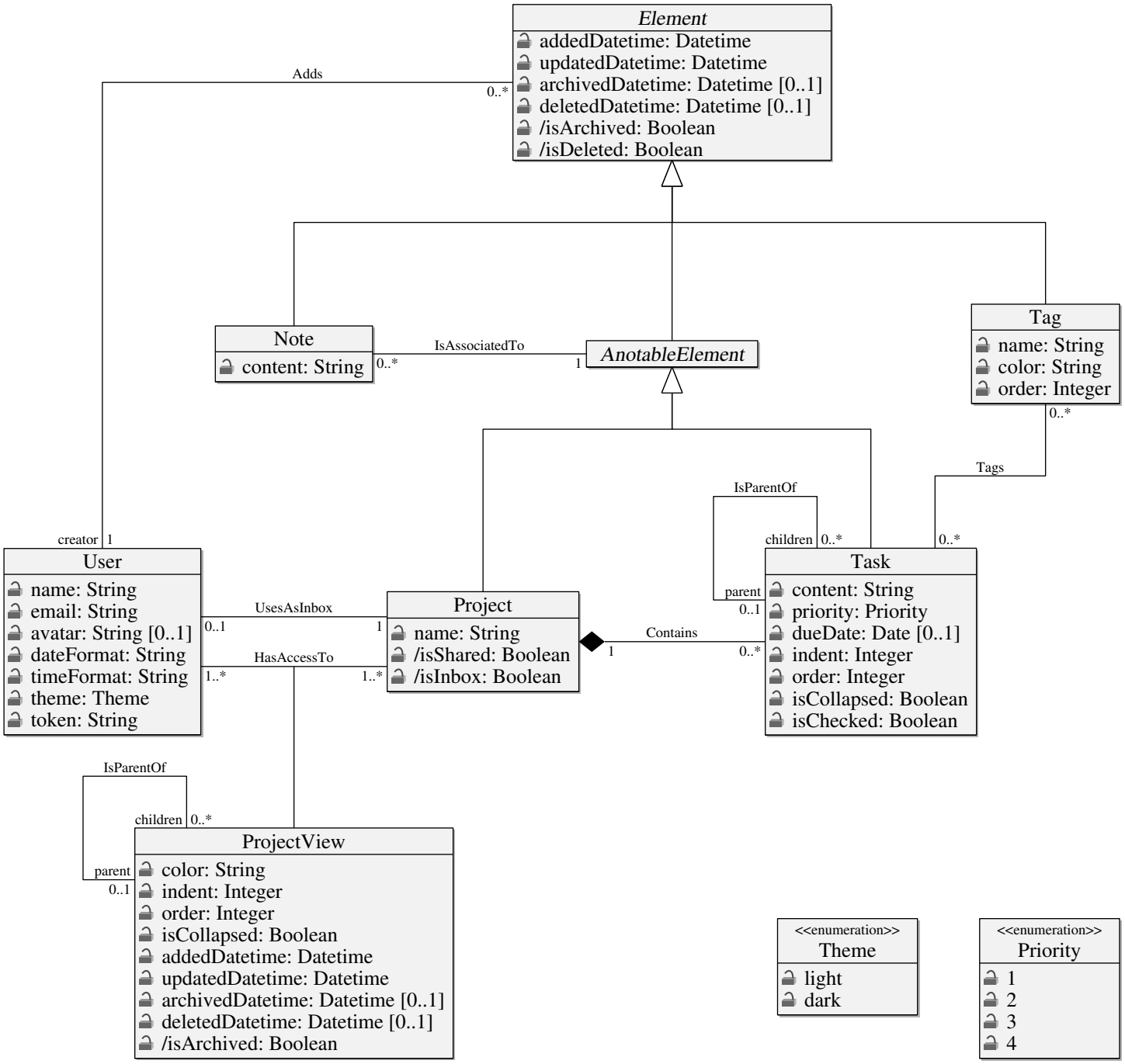

Figura 32. Diagrama de clases UML

En particular, se contemplaron cuatro tipos de entidades principales: tareas, proyectos, etiquetas y notas. Sus correspondientes clases están acompañadas de otras más secundarias o menos específicas, que completan el pequeño sistema (usuarios, enumeraciones, superclases, etc.). La figura 32 muestra un diagrama de clases UML

con el diseño conceptual final de la aplicación —el diagrama se ha visto sometido a modificaciones menores durante la etapa de desarrollo.

Observaciones

- Atributos derivados:
  - `Element./isArchived`: es verdadero si y solo si `archivedDate` no es nulo.
  - `Element./isDeleted`: es verdadero si y solo si `deletedDate` no es nulo.
  - `Project./isShared`: es verdadero si y solo si hay más de un usuario con acceso al proyecto (asociación `HasAccessTo`).
  - `Project./isInbox`: es verdadero si y solo si el objeto participa en una asociación `UsesAsInbox`.
- Restricciones:
  - `UsesAsInbox` es un subconjunto de la asociación `HasAccessTo`: los usuarios siempre tienen acceso al proyecto que actúa como su Bandeja de Entrada.
  - No se podrá borrar un proyecto que sea Bandeja de entrada de un usuario.
- `User.avatar` es un identificador alfanumérico de una imagen. Todas las imágenes se guardan dentro de una misma carpeta, organizadas según su identificador.
- `User.token` es un parámetro necesario para el acceso por API.
- Los atributos `color` (clases `Tag` y `ProjectView`) contienen un identificador HTML, como por ejemplo: “#FFCC00” o “red”.
- Una tarea puede encontrarse en cuatro estados diferentes:
  a) **Pendiente**:
  Aún no se ha completado; se halla desmarcada. Sus atributos valdrán:
    - `isChecked = False`
    - `isArchived = False`
    - `isDeleted = False`

  b) **Completada**:
  Se halla marcada pero no archivada (permanece visible). Solo es posible cuando la tarea es subtarea de otra tarea que aún no se ha completado. Los valores de sus atributos serán:
    - `isChecked = True`
    - `isArchived = False`
    - `isDeleted = False`

  c) **Archivada**:
  Se halla completada y archivada (no visible). Sus atributos valdrán:
    - `isChecked = True`
    - `isArchived = True`
    - `isDeleted = False`

  d) **Borrada**:
  Se ha borrado, pero aún se encuentra en la papelera; todavía es recuperable. Los valores de sus atributos serán:
    - `isChecked = -`
    - `isArchived = -`
    - `isDeleted = False`
- Los elementos borrados (con atributo `isDeleted=True`) permanecen almacenados hasta que transcurra un periodo de tiempo predefinido y parametrizable desde su fecha de borrado. Mientras el periodo no se agote, la tarea será accesible desde un mecanismo equivalente a una «papelera de reciclaje».

2. Implementación

El diseño conceptual propuesto se tradujo a la hora del desarrollo en una jerarquía de clases, con sus correspondientes atributos y métodos, dentro del modelo de datos de Django. El *framework* ha permitido una representación fiel del diseño de partida, puesto que cada modelo se implementa como una clase y sobre esta se definen sus atributos y métodos.

Además, Django dispone de un ORM (*Object-relational Mapping*) integrado, que se encarga de ocultar al programador el modelo relacional de la base de datos subyacente, de forma que el tratamiento de datos se realiza mediante objetos y métodos. Con ello, se mantiene una mayor cercanía con el modelo conceptual de partida y ya no es necesario realizar las traducciones a los modelos lógico-relacional y físico.

De esta forma, la implementación consistió fundamentalmente en construir los modelos de Django a imagen y semejanza del modelo conceptual, definir las URL del sitio web, diseñar y elaborar las plantillas que generarán el HTML, y programar las vistas que se encarguen de interactuar con los modelos y las plantillas. Cabe indicar que, como cabría esperar, el grueso del trabajo se ocupó de la implementación de las vistas, que son las que portan toda la lógica de negocio de la aplicación.

La funcionalidad del servidor puede explicarse a través de las acciones que contempla sobre los cuatro elementos de información mencionados anteriormente (tareas, proyectos, etiquetas y notas). Las acciones más básicas se pueden resumir por el acrónimo CRUD (*Create*, *Read*, *Update*, *Delete*), que se corresponderían con la creación, lectura, edición y eliminación de elementos. Pero, además, ha sido necesario implementar acciones más complejas o específicas, como el archivado de elementos, la compartición de proyectos o el completado de tareas.

A renglón seguido se enumeran las acciones implementadas, agrupadas según el tipo de elemento al que afectan:

- **Proyectos**:
  - Consulta (del proyecto y sus tareas).
  - Creación.
  - Edición: nombre y color asociado.
  - Eliminación.
  - Archivado.
  - Recuperación (del archivado).
  - Compartición.
  - Eliminación de la compartición.
  - Ordenación automática de sus tareas.
- **Tareas**:
  - Consulta.
  - Creación.
  - Edición: contenido, fecha de vencimiento, prioridad, proyecto.
  - Eliminación.
  - Archivado.
  - Recuperación (del archivado).
  - Completado.

    - Descompletado.
- **Notas**:
    - Creación (desde la edición de tareas).
    - Edición (desde la edición de tareas).
    - Eliminación (desde la edición de tareas).
- **Etiquetas**:
    - Creación (desde la edición de tareas).
    - Edición (desde la edición de tareas).
    - Eliminación (desde la edición de tareas).

Estas acciones se ejecutan desde el área principal de la aplicación web, que podría denominarse consola de administración de tareas. Esta se estructura en tres grandes bloques. El primero de ellos se denomina *Bandeja de entrada* y es el lugar donde se depositan las tareas a las que no se les asigne un proyecto específico. De acuerdo a la metodología GTG (*Getting Things Done*), la bandeja sirve de cajón de sastre donde registrar tareas sin tener que distraerse del trabajo que se esté realizando en ese momento. Más tarde, con calma, se reorganizarán estas tareas, asignándoles prioridades, vencimientos y proyectos de destino.

El segundo bloque lo conforman dos vistas dinámicas: *Hoy* (*Today*) y *Próximos 7 días* (*Next 7 days*). La primera muestra las tareas (de cualquier proyecto) cuyo vencimiento sea hoy o bien estén vencidas. La segunda vista se comporta de forma análoga, pero contemplando, además de hoy, los próximos 6 días.

El último bloque lo conforman los proyectos específicos, que contienen conjuntos de tareas relacionadas. Los proyectos tienen asignado un color y pueden compartirse entre varios usuarios. Dentro de los proyectos se pueden ordenar automáticamente sus tareas por vencimiento, contenido (alfabéticamente) o por prioridad.

Asimismo, se ha implementado una sencilla área de administración, que por el momento se limita a la administración de usuarios. Desde este apartado un usuario con rol de administrador puede crear nuevos usuarios, eliminarlos, reestablecer su contraseña y asignar o retirar su rol de administración.

3. Resultado

La mejor forma de ver el servidor web y el cliente HTML desarrollados es mediante la interacción real con los mismos. En caso de que esto no fuese viable, las figuras 33 y 34 muestran dos capturas de pantalla del cliente HTML a título ilustrativo. La primera presenta la vista de los próximos siete días, mientras que la segunda se corresponde con el área de administración de usuarios. En ambos casos se ha introducido previamente un pequeño conjunto de datos de prueba.

Cabe indicar que se realizó un sencillo diseño de la imagen visual de la aplicación. Obviamente, no se trata de un trabajo profesional, pero contribuye a una experiencia integral del desarrollo de una solución interactiva, como la que se ha llevado a cabo. Después de unas pruebas de diseño y evaluación se llegó a la decisión de conservar dos versiones del logotipo, con y sin bisel, así como una versión en color y dos en monocromo. Por su parte, el nombre de la aplicación es Funtasktic, haciendo un juego de palabras con la diversión (*fun*), las tareas (*task*) y las Tecnologías de la Información y el Conocimiento (TIC). La figura 35 muestra las diferentes variantes del logotipo.

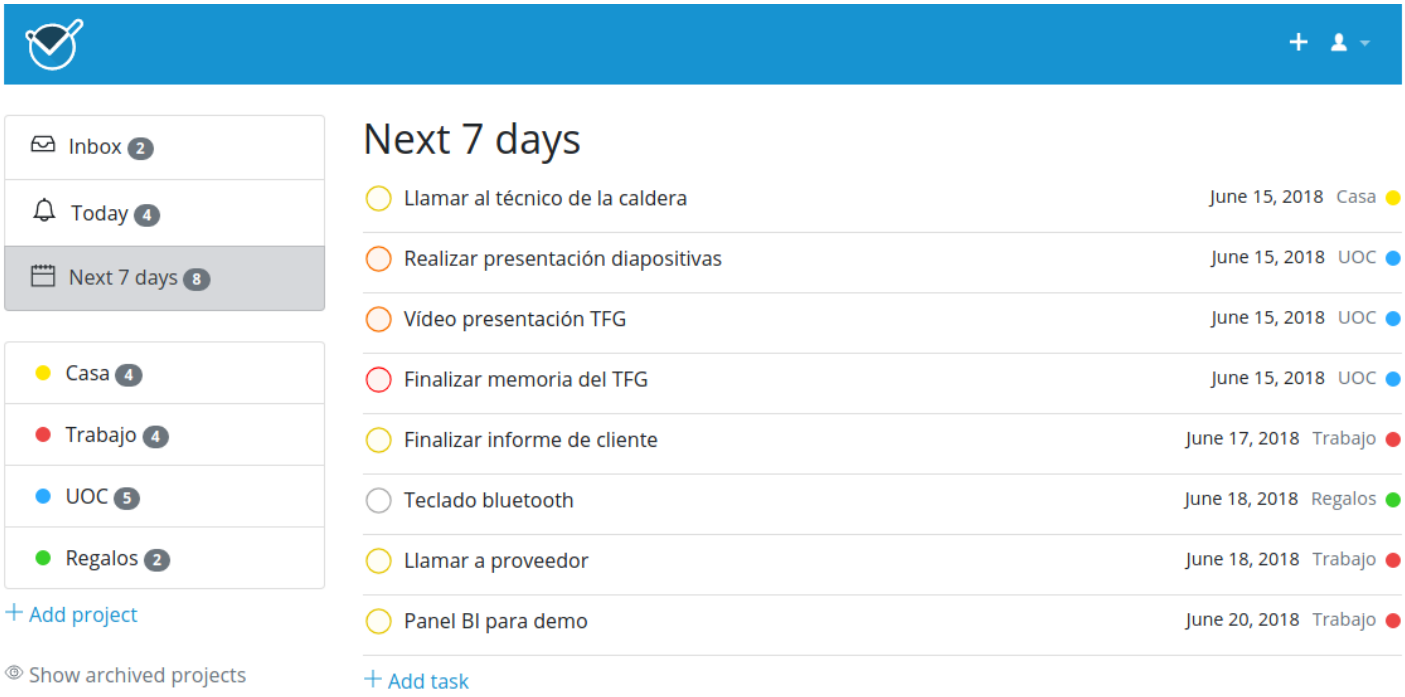

Figura 33. Cliente HTML: vista de los próximos 7 días

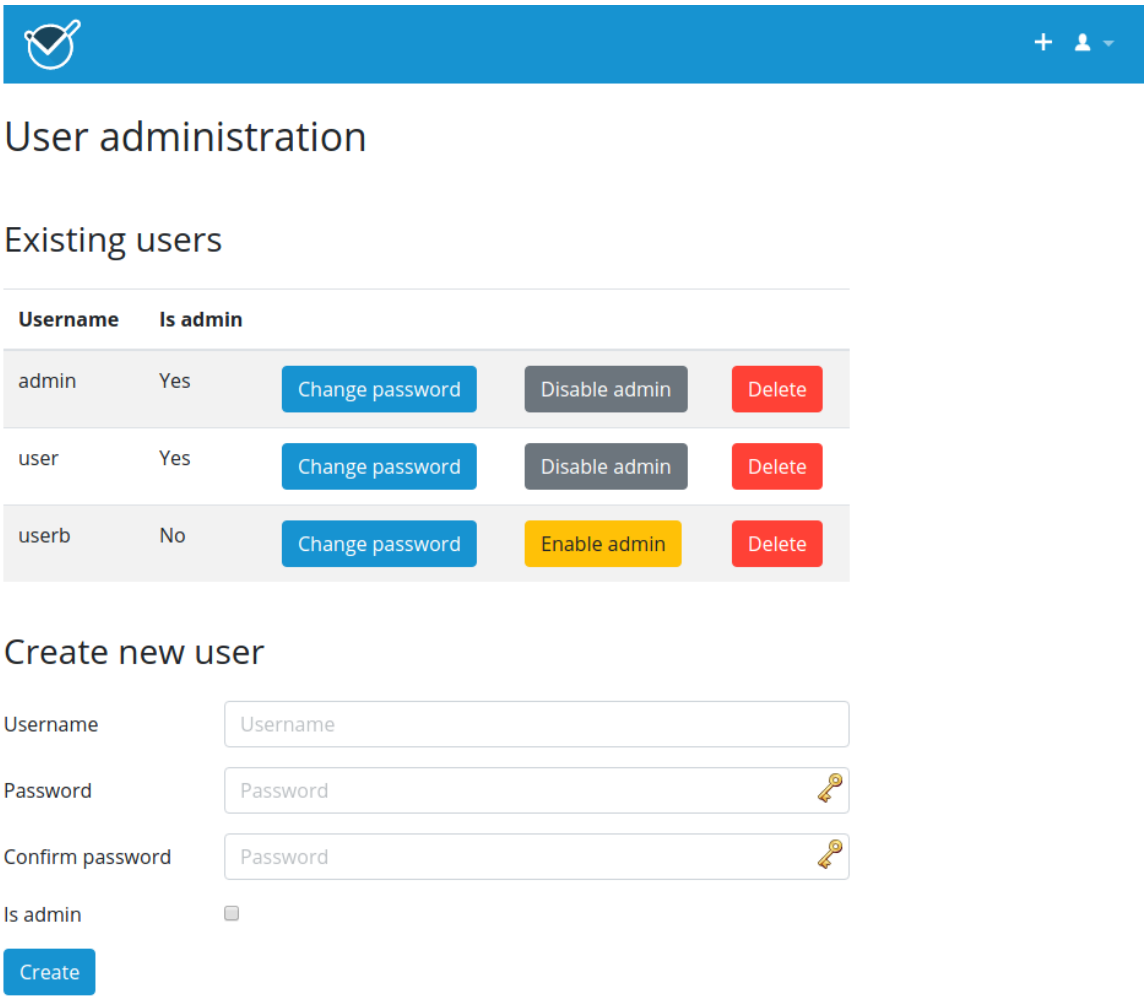

Figura 34. Cliente HTML: administración de usuarios

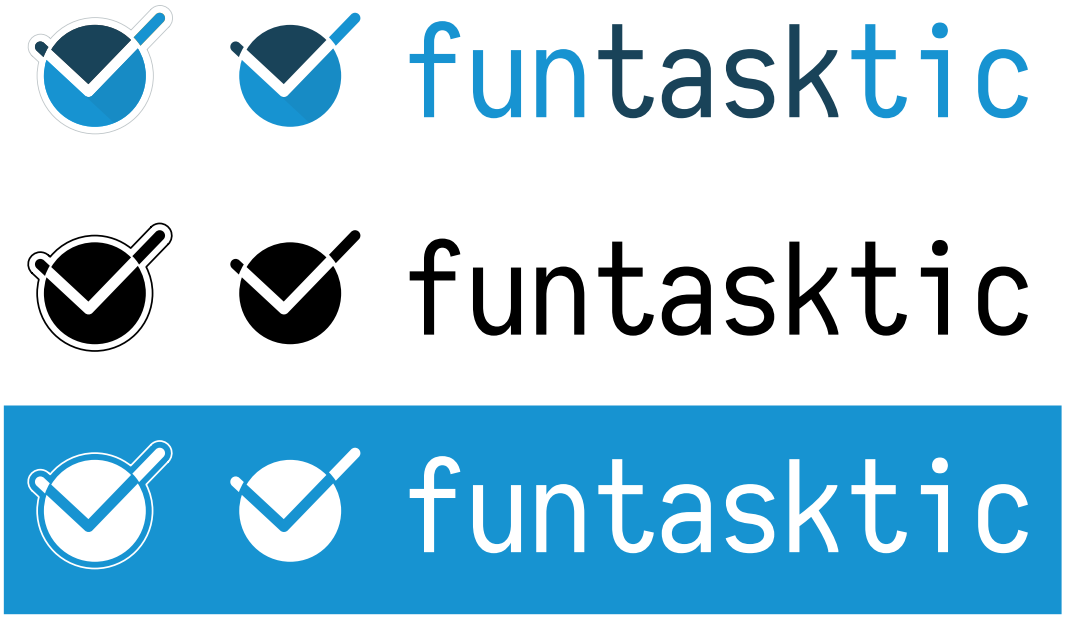

Figura 35. Logotipo de la aplicación

4. Despliegue

En el ciclo de vida de una aplicación se suelen diferenciar dos entornos o contextos de ejecución: depuración (*debug*) y producción. El primero aparece cuando los programadores ejecutan la aplicación en un entorno controlado de pruebas durante el desarrollo. El segundo, por el contrario, aparece cuando la herramienta se pone a disposición de los usuarios finales en un contexto de uso real. En el primer escenario es deseable que la aplicación facilite la labor de los programadores, proporcionando, por ejemplo, gran cantidad de información de depuración, aún siendo en detrimento del rendimiento y estabilidad del sistema. En el entorno de producción, en cambio, se busca la máxima estabilidad, seguridad y rendimiento posibles.

Django —y, por tanto, la aplicación servidor desarrollada— ofrece la posibilidad de trabajar en un entorno de desarrollo o en uno de producción y alternar fácilmente entre ellos. En el caso del entorno de depuración, el *framework* dispone de un servidor web integrado que, si bien su rendimiento es contenido, facilita la labor de programación y depuración. Para el despliegue en el entorno de producción se ha optado, tal y como se recomienda en la documentación oficial, por un servidor web dedicado.

En particular, se utilizarán dos servidores actuando sobre Django. El primero es Gunicorn, un servidor WSGI HTTP en Python. Este se encargará de recibir y gestionar las peticiones HTTP y dirigirlas al servicio Django. El segundo servidor es una instancia de Nginx actuando como *proxy* inverso. Su labor será doble. En primer lugar, recibirá las peticiones HTTP al dominio en cuestión y las redirigirá al servidor Gunicorn. Esto permite enmascarar el servidor web y además facilitaría una futura implementación de SSL sobre todo el sistema, así como establecer mecanismos de balanceo de cargas. La segunda labor es la de servir los ficheros estáticos del sitio web (hojas de estilo, imágenes y *scripts*), tarea que no debe realizar Gunicorn.

Para facilitar y automatizar el despliegue se ha utilizado tecnología de contenedores. En concreto, se han empleado Docker y Docker Compose para virtualizar los distintos servidores mencionados anteriormente. Estos se distribuyen entre tres contenedores. El primero alberga la instancia de Nginx; el segundo, los servidores Gunicorn y Django,

y el último, el servicio PostgreSQL. Para ello, ha sido necesario definir un fichero Dockerfile, que se encarga de construir la imagen del segundo contenedor, y un fichero *docker-compose*, que se ocupa de la orquestación de los distintos contenedores dentro de una misma red virtual, así como de arrancar y detener los contenedores en bloque.

El resultado final es un servicio global que sirve la aplicación desarrollada a través del puerto 8000. Por lo tanto, en el caso de lanzar el despliegue en una máquina local, la aplicación estará disponible en la dirección http://localhost:8000/ o bien http://127.0.0.1:8000/. La figura 36 muestra un esquema del despliegue de la aplicación mediante Docker Compose.

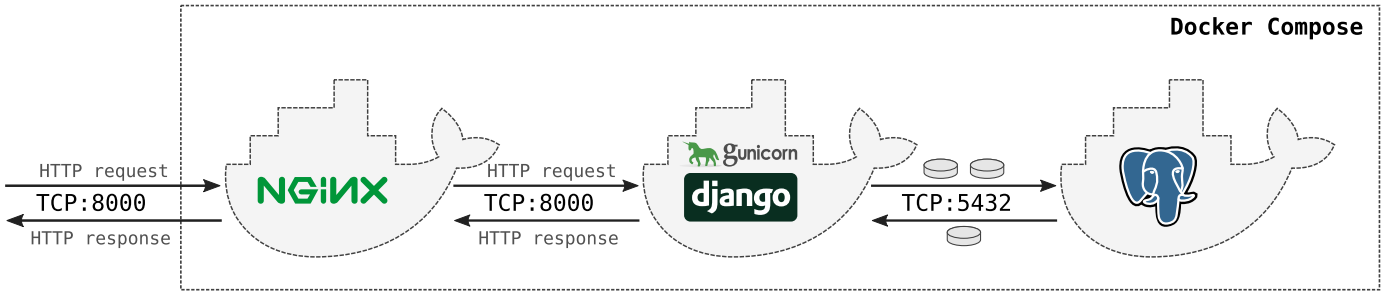

Figura 36. Despliegue del servidor en producción mediante Docker Compose

Las instrucciones detalladas para el despliegue del servidor, tanto en un entorno de producción con Docker Compose, como en un entorno virtual Python con el servidor de desarrollo Django, se encuentran en el fichero `README.md` ubicado en la carpeta raíz del servidor (carpeta `funtasktic` dentro del fichero comprimido que acompaña a este documento).

4

API

Una API (*Application Programming Interface*) es una interfaz que permite la interacción programática con una determinada aplicación, en lugar de necesitar de interacción humana, como sucede con las interfaces habituales. Esto permite la integración de aplicaciones (o sistemas completos) entre sí, de forma que se comuniquen sin necesidad de mediación humana. Sirva de evidencia de la utilidad de las API cómo la profileración de estas ha traído consigo una enorme y evidente integración entre todo tipo de productos y servicios de lo más dispar (aplicaciones que publican en redes sociales automáticamente, aplicaciones móviles que permiten manipular la iluminación de la casa, etc.).

Las API, además de facilitar la conectividad entre sistemas de terceros, también facilitan el desarrollo de clientes móviles por parte del propio fabricante del servidor. Sin ir más lejos, los clientes móviles de Todoist, Wunderlist y Remember the milk hacen uso de la API pública para realizar las operaciones, entre otras, de sincronización con el servidor. De este modo, se consideró ineludible la tarea de desarrollar también una API para Funtasktic. Nuevamente, esta no será una solución equiparable a las profesionales, sino una pequeña muestra que sirva para entender y poner en práctica el diseño y la implementación de estos servicios.

1. Diseño

El patrón más extendido para el diseño de las API web es REST (*representational state transfer*). Este, a diferencia de otros estándares mucho más complejos y específicos como SOAP, utiliza protocolos existentes y generales para su implementación. En concreto, emplea HTTP para la comunicación y, habitualmente, JSON o XML para la serialización de los datos. El patrón de diseño más habitual consiste en definir *endpoints* (URL a las que se dirigen los clientes para acceder a un determinado recurso) y asociar los diferentes verbos HTTP a las acciones de manipulación de datos habituales (CRUD, *create read update delete*). De este modo, se optó por aplicar el paradigma REST a la API desarrollada.

Para el diseño de esta pequeña API solo se han considerado las tareas, proyectos y notas. Para cada tipo de elemento se han definido dos *endpoints* diferentes, uno para el listado de los elementos existentes y la creación de nuevas instancias (CR, *create read*), y otro para la visualización, edición y eliminación (RUD, *read update delete*). Asimismo, se han añadido dos *endpoints* más: el primero muestra las tareas de la bandeja de entrada; el segundo se destina a la sincronización de clientes según el protocolo que se explicará más adelante.

A continuación se enumeran los *endpoints* (rutas relativas a la raíz del sitio web) y una breve descripción de su propósito:

- **`api/inbox/`**
  Retorna el listado con las tareas que se encuentran en la bandeja de entrada.
- **`api/projects/`**
  Retorna el listado de projectos existentes.
- **`api/projects/<id-proyecto>/`**
  Retorna sus detalles y las tareas, actualiza o elimina el proyecto indicado.
- **`api/tasks/`**
  Retorna el listado de tareas existentes.
- **`api/tasks/<id-tarea>/`**
  Retorna sus detalles, actualiza o elimina la tarea indicada.
- **`api/notes/`**
  Retorna el listado de notas existentes.
- **`api/notes/<id-nota>/`**
  Retorna sus detalles, actualiza o elimina la nota indicada.
- **`api/sync/`**
  Permite la sincronización entre los clientes y el servidor.

1.1. Protocolo de sincronización

Entender cómo funciona un protocolo de sincronización en una aplicación distribuida como la que se ha tratado de desarrollar ha sido el gran motivador del propio trabajo. En consecuencia, la mayor parte de la fase de investigación y diseño de destinó al protocolo. Durante la investigación se descubrió que este problema aparece con frecuencia en los sistemas distribuidos y existe abundante literatura al respecto. Dos son las estrategias fundamentales de sincronización: por estados o por operaciones.

La sincronización por estados se basa en que los distintos agentes intercambian objetos íntegros correspondientes a diferentes estados de estos. Fruto de un algoritmo de consenso, los diferentes agentes terminan disponiendo de la misma versión de cada objeto. El inconveniente de esta aproximación es que puede reducir considerablemente el rendimiento del sistema, puesto que, entre otras limitaciones, es necesario enviar los objetos de forma completa, incluso ante cambios menores.

La sincronización por operaciones, en cambio, opta por que los agentes intercambien las diferentes operaciones que vayan registrando localmente (y que no todos conocen). Así, cuando cada agente disponga de todas las operaciones del sistema y las aplique sobre su copia local de los objetos, se alcanzaría la consistencia global. El inconveniente de esta estrategia es que es necesario garantizar una ordenación de las operaciones coherente en todos los agentes. Esta no es en absoluto una tarea sencilla y existe una gran cantidad de algoritmos que afrontan el problema, con diferentes ventajas, limitaciones, roles de los agentes, requisitos y garantías.

El protocolo diseñado finalmente se ha inspirado en el de Todoist —aunque adaptado a un contexto de alcance más modesto—, y apuesta por un enfoque híbrido entre las dos estrategias anteriores. En concreto, los agentes enviarán las operaciones que registran al servidor. Este, por su parte, responderá con el estado de los objetos actualizados desde la última sincronización con dicho cliente. Así, en este esquema el

servidor tomará un rol central y de coordinación respecto a los clientes. Estos últimos no se comunicarán entre sí, sino que se limitarán a enviar sus operaciones al servidor y que este les indique cómo tienen que actualizar su estado.

En el momento en el que el servidor recibe la solicitud de una sincronización, en primer lugar y siempre que proceda, procesará por orden la cola de operaciones (que mantiene posibles relaciones "sucedió antes" en el contexto local del cliente). Estas operaciones darán lugar a actualizaciones de objetos, que también se incluirán en la respuesta junto con las originadas por operaciones de otros clientes o el servidor. La resolución de posibles conflictos entre las operaciones de diversos clientes se realizará solamente en el servidor, en el marco de una arquitectura cliente-servidor entre cada cliente y el servidor. Así, para simplificar el problema, se ha optado por procesar las sincronizaciones de forma serializada; cada sincronización esperará a que finalice la anterior y los conflictos aparecerán y se resolverán aisladamente, para cada sincronización (cliente a cliente). Aunque esta estrategia limita seriamente la concurrencia del servidor —por tanto, su rendimiento potencial—, lo cierto es que en una aplicación de este tipo la concurrencia es muy reducida, y con ello también la pérdida real de desempeño.

La principal ventaja de este algoritmo es que permite que los clientes puedan trabajar desconectados y no dependan de un acceso continuo al servidor. Además, se reduce la cantidad de información que hay que intercambiar y se alcanza un buen rendimiento. Otro argumento a su favor es la relativa sencillez de su implementación, ya que el mecanismo de sincronización no es complejo: la complejidad reside en la sofisticación y eficiencia del mecanismo de resolución de conflictos. Sin embargo, dados el objetivo y la disponibilidad temporal para el TFG, así como la baja concurrencia de la aplicación en particular, podría implementarse un mecanismo de sincronización sencillo y de naturaleza optimista. Con ello se dejaría margen a la mejora en futuras revisiones, tal y como consta en el último apartado del presente documento.

Para que el servidor pueda determinar si un cliente ya dispone de la última versión y, en caso de no ser así, poder enviarle los nuevos datos, genera un UUID (*Universal Unique Identifier* o identificador único universal) para cada sincronización, que le hace llegar al cliente para que lo almacene. Posteriormente, cuando el cliente inicie una nueva sincronización, incluirá el UUID de la sincronización anterior en la petición y recibirá un nuevo identificador en la respuesta, y así sucesivamente. El servidor tiene asociada a cada UUID una marca temporal correspondiente al momento en que se produjo la sincronización. Por su parte, los objetos en el servidor disponen de una marca temporal de última actualización. Así, cuando el servidor recibe el UUID de la sincronización anterior puede determinar qué objetos debe enviar al cliente para que este actualice sus datos locales y se logre así la consistencia.

Otra decisión importante consiste en no asignar marcas temporales a las operaciones en los clientes. En su lugar, como se ha indicado, cuando el servidor procese las operaciones asignará la correspondiente marca al histórico de operaciones y a la última modificación de los objetos afectados. La ventaja de esta aproximación es que se simplifica el diseño y se evitan los clásicos problemas de sesgos y desviaciones entre los relojes de cada agente. En su lugar, existe un único reloj global, el del servidor. Como contrapartida, el inconveniente principal es que, en el caso de que el cliente haya permanecido desconectado durante un largo periodo de tiempo, estas marcas temporales pueden desviarse mucho del momento real en el que se produjeron las operaciones. Sin embargo, esta limitación no debería ser preocupante si se tienen en cuenta los condicionantes de la aplicación antes mencionados: la mayor parte del tiempo los clientes funcionan de forma pseudosíncrona (con periodos de desconexión

generalmente breves) y el nivel de concurrencia es reducido.

Otra implicación de las decisiones anteriores es que los últimos cambios recibidos se imponen a los anteriores, aún en el caso de que se hubiesen generado en el orden inverso. Nuevamente, este comportamiento no debería resultar preocupante en el caso del contexto de uso pseudosíncrono considerado. Asimismo, puede destacarse que el rol central del servidor reduce la robustez del sistema en su conjunto, ya que de fallar sería imposible la sincronización en los clientes y la red quedaría particionada.

Resulta evidente que la mayor parte de la lógica de la aplicación, especialmente aquella más compleja, reside en el servidor. Es el caso de la resolución de conflictos al procesar operaciones de clientes. Al procesar las operaciones una a una y sincronización a sincronización, los conflictos podrán aparecer tan solo cuando la operación actual no pueda aplicarse. El motivo será que una operación realizada por otro cliente y desconocida en su momento por el cliente de la sincronización actual ha dado lugar a una situación que impide la ejecución de la operación. Por ejemplo, que otro cliente haya eliminado previamente un objeto al que ahora se trata de modificar su contenido. En caso de conflicto al tratar de aplicar una operación, el servidor la marcará como fallida y generará información del error. Al finalizar el procesamiento de la sincronización, incluirá en la respuesta el estado-resultado de la aplicación de cada operación, así como la información de error en caso de que resultase fallida.

Por su parte, el cliente se encargará de informar al usuario de las operaciones que han sido fallidas. Estas operaciones permanecerán a la cabeza de la cola, pero sin detonar nuevas sincronizaciones por sí solas. En futuras sincronizaciones se enviarían nuevamente. Excedido el número de intentos máximo, se procedería con el mecanismo de descarte (interactivo o no) con normalidad.

Para minimizar el número de sincronizaciones, el cliente enviará siempre al servidor las operaciones pendientes antes de recibir nuevos datos del servidor. Es decir, nunca enviará una solicitud de sincronización “vacía” (sin operaciones locales) si existen operaciones en la cola.

Así pues, las acciones que realizaría el cliente tras recibir una sincronización serían:

1. Actualizar el UUID de última sincronización.
2. Eliminar/sobrescribir los objetos creados localmente cuya operación de creación ha sido aceptada.
3. Añadir o actualizar los objetos recibidos.
4. Si fuese necesario, actualizar las referencias a los identificadores temporales (generados en su momento por él) con los identificadores reales asignados finalmente por el servidor.
5. Desencolar las operaciones ejecutadas con éxito.
6. Tratar las operaciones fallidas, si procede: incrementar contador de intentos, mostrar errores al usuario, etc.
7. Actualizar la interfaz de usuario.

Finalmente, el protocolo de sincronización también contempla una *sincronización completa*, a diferencia de la explicada anteriormente (denominada *sincronización parcial*). En una sincronización completa el servidor envía al cliente la última versión de todos los objetos y este último, por su parte, descartará toda la información local de los objetos y operaciones, limitándose a incorporar los objetos recibidos. Este tipo de sincronización sirve dos propósitos: por un lado, se ocupa de inicializar los

clientes durante su primera conexión al servidor; por otro, sirve de mecanismo de recuperación en caso de conflictos irresolubles o comportamientos inesperados.

Estructura de los mensajes

En las figuras 41 y 42, se muestra la estructura general de los mensajes intercambiados por cliente y servidor durante una sincronización. Cabe indicar que el contenido dependerá del tipo de cada operación enviada por el cliente, que hará variar la estructura del parámetro `args` de la operación. También dependerá de si la sincronización es total o parcial. Además, se utilizará el parámetro `last_sync_token` con dos propósitos: para indicar la última sincronización realizada por el cliente y para solicitar una sincronización total (enviando un valor) nulo.

Figura 41. Estructura del mensaje enviado por el cliente durante una sincronización

```
{
  "last_sync_token":"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "operations": [
                  {"uuid": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
                   "type": "<tipo_de_tarea>",
                   "args": {
                             "<parámetro>": "<valor>",
                             "...": "..."
                           }
                  },
                  {"...": "..."}
                ]
}
```

Figura 42. Estructura del mensaje de respuesta del servidor durante una sincronización

```
{
  "sync_token": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "full_sync": "<True/False>",
  "id_mappings": {
                    "<op_tem_id>": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxi"
                 },
  "operations_status": {
                         "<op_id>": "OK",
                         "<op_id>": ["Error",  "message"],
                       },
  "projects": [
                {
                  "<project_param>": "<value>",
                  "...": "..."
                },
              ],
  "tasks": [
             {
               "<task_param>": "<value>",
               "...": "..."
             },
           ]
}
```

De los campos del mensaje de respuesta, tan solo los dos primeros (`sync_token` y `full_sync`) son obligatorios. El mapeado de identificadores y el estado de las operaciones solo se enviarán cuando el cliente haya enviado operaciones. Por su parte, `projects` y `tasks` solo se incluirán cuando el servidor disponga de objetos actualizados. Naturalmente, el campo `operations` del cliente podrá enviarse sin operación alguna.

2. Implementación

Para la implementación de la API web se ha empleado la biblioteca complementaria a Django denominada Django REST Framework. Esta proporciona las herramientas necesarias para el desarrollo de APIs potentes y versátiles. En el caso de la aplicación que se ha desarrollado, nuevamente, solo se ha utilizado una pequeña parte de su potencial.

El código se estructura de una forma análoga a Django, basándose en URLs y vistas. Sin embargo, en el caso de las API, en lugar de utilizar plantillas para generar el contenido de las respuestas, las vistas se valen de *serializadores*. Estos últimos, se definen de forma similar a los modelos Django. La API se encarga de recibir, deserializar y validar los mensajes recibidos, y de hacer llegar a la vista los datos en forma de objetos Python. De este modo, se facilita sobremanera la labor del desarrollador.

De cara a la autenticación de usuarios, se han contemplado dos mecanismos alternativos. El primero consiste en el tradicional acceso mediante nombre de usuario y contraseña. El otro mecanismo se basa en el uso de *tokens*, de forma que cada usuario del sistema dispone de un *token* secreto y único. Cuando el cliente necesite autenticarse para realizar una operación en la API, incluirá su *token* en la cabecera de la petición HTTP correspondiente. Cuando el servidor reciba una petición, extraerá el *token* contenido y sabrá de qué usuario se trata, además de tener garantías criptográficas que solo dicho usuario puede encontrarse detrás de la petición.

Cabe indicar que tanto las credenciales de acceso como el *token* son datos confidenciales. Por lo tanto, es fundamental para la seguridad e integridad del sistema que estos no sean obtenidos por terceros. Por consiguiente, en un entorno de producción real sería imprescindible implementar una capa SSL sobre el servidor web (por ejemplo, sobre el *proxy* inverso Nginx), de modo que toda la comunicación entre cliente y servidor quede protegida por los mecanismo criptográficos subyacentes.

Otra de las grandes ventajas de Django REST Framework es que implementa una interfaz web visual para explorar y explotar la API. Así, dirigiéndose a la URL relativa `api/` se accede a la raíz de la API, donde se presentan los enlaces a los distintos *endpoints*. Además, estos enlaces son totalmente interactivos, de forma que se facilita la navegación entre recursos y *endpoints*. Estos enlaces también se envían a los clientes que utilicen la API directamente (sin la mediación de la interfaz web). Con ello se abre la posibilidad a una exploración programática del contenido y servicios de la API por parte de clientes que no conozcan cómo esta se estructura.

3. Resultado

Las figuras 43, 44 y 45 presentan el aspecto y contenido de varias páginas representativas de la API desarrollada. En la primera se muestra la raíz de la API; en la segunda, el listado de proyectos, y en la tercera, el detalle de un proyecto.

4. Despliegue

El despliegue de la API se realiza de forma conjunta e inseparable del resto de la aplicación Django. Por consiguiente, en el apartado de despliegue del servidor web se encuentra toda la información relevante y no es necesario indicar nada más.

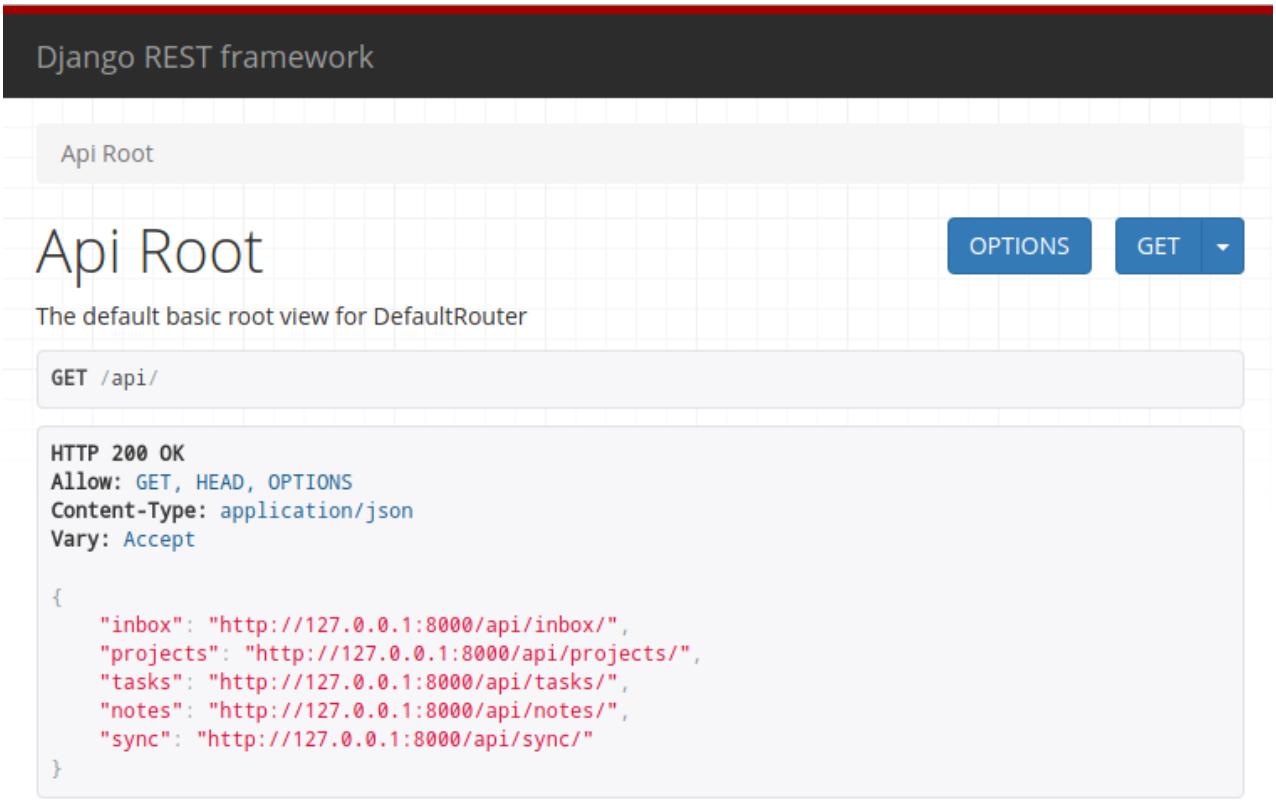

Figura 43. Interfaz web de la API: raíz

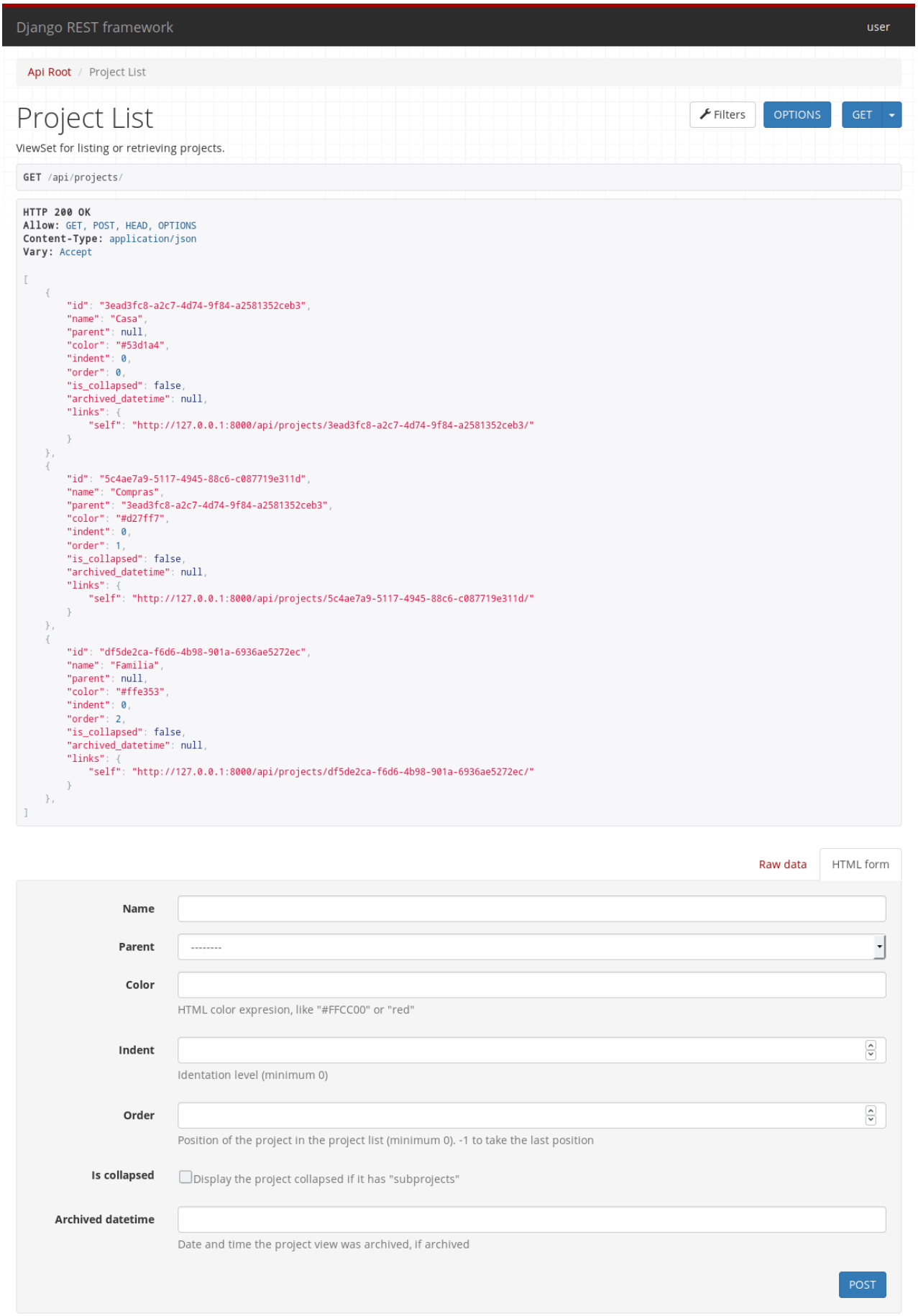

Figura 44. Interfaz web de la API: vista de lista de proyectos

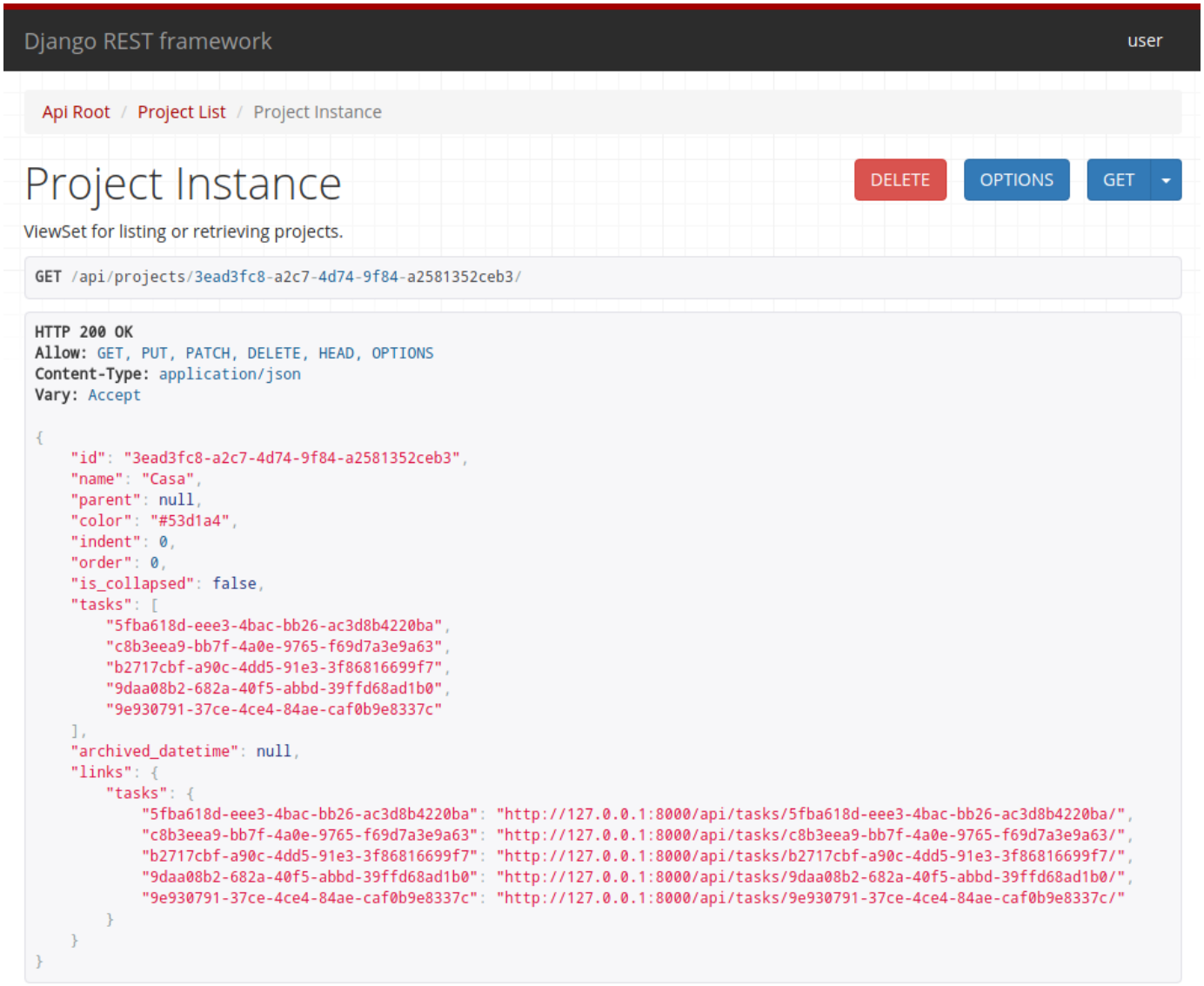

Figura 45. Interfaz web de la API: vista de un proyecto

5

Cliente distribuido

El último gran componente del sistema desarrollado es el cliente distribuido. Esta aplicación permite trabajar de forma local y desconectada, y posteriormente conectarse al servidor a través de la API para realizar la sincronización y alcanzar eventualmente la consistencia. Dada la naturaleza académica del proyecto y el tiempo dedicado a la construcción de servidor web y API, se optó por un enfoque más didáctico y demostrativo. Así, el objetivo que ha guiado el desarrollo del cliente ha sido mostrar empíricamente el funcionamiento del servidor, la API y el protocolo de sincronización en un entorno realista, aunque sencillo.

1. Diseño

De acuerdo a los objetivos planteados para el desarrollo del cliente, se buscó la infraestructura más sencilla posible que permitiese demostrar el funcionamiento del sistema en su conjunto. Así, se llegó a la conclusión de que la mejor alternativa sería una interfaz CLI (*command-line interface*), ya que reduciría significativamente la complejidad del cliente y, además, facilitaría la automatización de pruebas. El siguiente paso en el diseño de la aplicación fue definir las órdenes y parámetros ofrecidos al usuario.

En aras de la simplicidad, se optó por contemplar solo un subconjunto de las operaciones definidas previamente en el sistema. En particular, se hizo hincapié en aquellas necesarias para el proceso de sincronización. A continuación, se enumeran las operaciones incluidas en el cliente distribuido, en forma de órdenes, subórdenes y parámetros. Los valores de estos últimos se resaltan en cursiva. Los parámetros opcionales se muestran entre corchetes.

- **`init`** *`servidor usuario contraseña`*

  Se autentica en el servidor, obtiene el token del usuario y realiza la sincronización inicial (completa).

- **`status`** `[--debug]`

  Muestra el estado del servidor: última sincronización, operaciones locales pendientes, etc.

- **`sync`** `[--full] [--debug]`

  Realiza una sincronización con el servidor.

- **`projects list`** `[--archived]`

  Muestra la lista de proyectos.

- **projects show** *proyecto* [--archived]
  Muestra la lista de tareas de un proyecto.
- **projects add** *atributos_del_proyecto*
  Añade un nuevo proyecto.
- **projects update** *proyecto atributos_del_proyecto*
  Edita los datos de un proyecto.
- **projects archive** *proyecto*
  Archiva un proyecto.
- **projects unarchive** *proyecto*
  Desarchiva un proyecto.
- **projects delete** *proyecto*:
  Elimina un proyecto.
- **tasks list** [--archived]
  Muestra la lista de todas las tareas.
- **tasks add** *atributos_de_tarea*
  Añade una nueva tarea.
- **tasks update** *tarea atributos_de_tarea*
  Edita los datos de una tarea.
- **tasks archive** *tarea*
  Archiva una tarea.
- **tasks unarchive** *tarea*
  Desarchiva una tarea.
- **tasks check** *tarea*
  Marca una tarea como completada (y la archiva).
- **tasks uncheck** *tarea*
  Desmarca una tarea como completada (y la desarchiva).
- **tasks delete** *tarea*
  Elimina una tarea.

2. Implementación

Como ya se ha adelantado en los apartados anteriores, una de las decisiones iniciales fue adoptar un mismo lenguaje de programación principal para todos los componentes del sistema. En consecuencia, para la implementación del cliente también se utilizó Python. De este modo, el paso previo al desarrollo fue un estudio del ecosistema de este lenguaje, en busca de módulos o *frameworks* que facilitasen la encomienda. Se siguieron dos caminos de exploración, uno destinado a encontrar una solución de desarrollo de aplicaciones CLI y otro para encontrar una herramienta que simplificase la capa de datos.

Para la interfaz CLI se optó por la biblioteca Click, que abstrae al programador de la problemática de controlar y procesar las órdenes, subórdenes y parámetros, así como el trabajo con la entrada y la salida estándar. Desde el punto de vista de la implementación, Click se integra en el programa mediante decoradores Python que se añaden a las funciones que responden a cada una de las órdenes. De este modo, se

minimiza la interferencia de la biblioteca en el programa de partida y se obtiene un código simple, estructurado y elegante.

La capa de datos, por su parte, se apoya en la biblioteca ZODB. Esta implementa un sistema de gestión de bases de datos basado en grafos dirigidos, que se presenta mediante una API Python tremendamente idiomática. Así, el programador se limitará a definir sus clases como subclases de la clase `Persistent` y a añadir los objetos al objeto raíz de ZODB, denominado `root`. A este último se accede como a un diccionario. Dado que ZODB se basa en transacciones, también será necesario llamar a la función `commit` para confirmar los cambios realizados; la biblioteca se encarga de almacenar los datos de forma segura.

Cabe destacar que ZODB soporta varios medios de almacenamiento diferentes, desde una base de datos en memoria o un simple fichero (al modo de SQLite), hasta complejos SGBD relacionales como PostgreSQL u Oracle. Dados los objetivos fijados para el desarrollo del cliente, se optó por la alternativa más simple: guardar los objetos en un archivo del sistema de ficheros local. Así, cuando el cliente se ejecute abrirá el fichero `data.fs` ubicado en el directorio de ejecución y sobre él se realizarán todas las operaciones de lectura y escritura.

Finalmente, para interactuar con la API REST, se ha recurrido a Requests, una de las bibliotecas más utilizadas del ecosistema Python. Esta ofrece una abstracción de alto nivel e integrada a la perfección en la idiomática de Python para realizar cualquier tarea relacionada con el protocolo HTTP. En el caso del cliente desarrollado, ha facilitado entre otras labores: la autenticación con el servidor, la serialización y deserialización de los datos intercambiados, la construcción y deconstrucción de las solicitudes y respuestas HTTP (así como sus parámetros query, GET y POST), la codificación y descodificación de estados, el manejo de *cookies*, etc.

Una vez detalladas las explicaciones y el funcionamiento de los tres componentes del sistema (servidor, API y cliente distribuido), se presenta un pequeño esquema que recapitula todos los elementos tecnológicos y funcionales vistos hasta el momento, y cómo estos se orquestan. La figura 51 presenta un pequeño diagrama que resume la arquitectura funcional básica del sistema.

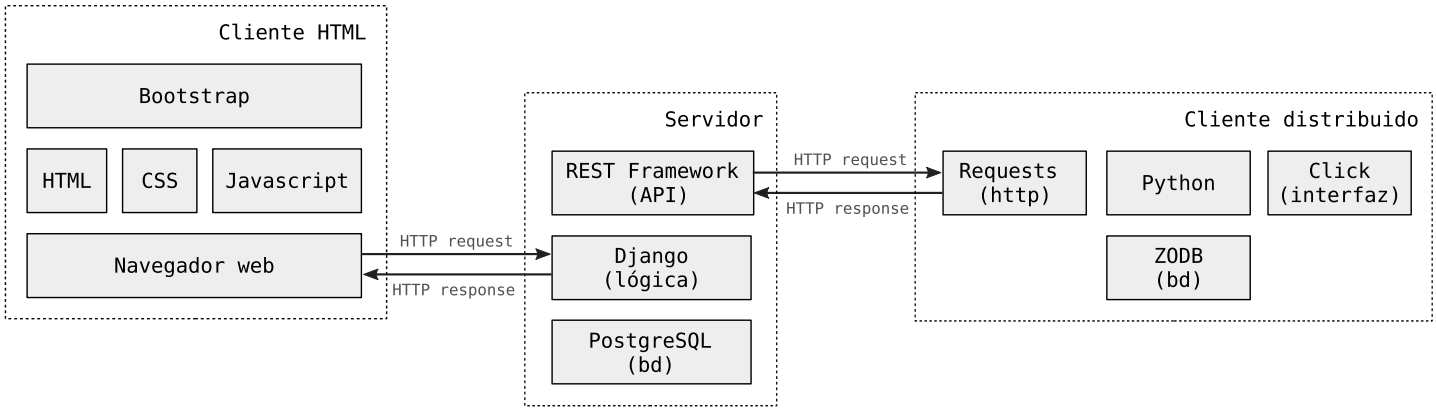

Figura 51. Esquema resumen de la arquitectura funcional del sistema

3. Resultado

La figura 52 muestra el aspecto y funcionamiento del cliente desarrollado. La captura contenida presenta el resultado de la ejecución de las siguientes operaciones: mostrar

la lista de proyectos, mostrar el contenido del proyecto *casa*, mostrar el contenido del proyecto *casa* incluyendo las tareas que han sido archivadas, borrar el projecto *azul*, mostrar el estado del cliente (en ese momento tiene las dos operaciones anteriores pendientes de sincronización), realizar una sincronización con el servidor (parcial) y finalmente mostrar el estado del servidor tras la sincronización (sin tareas pendientes).

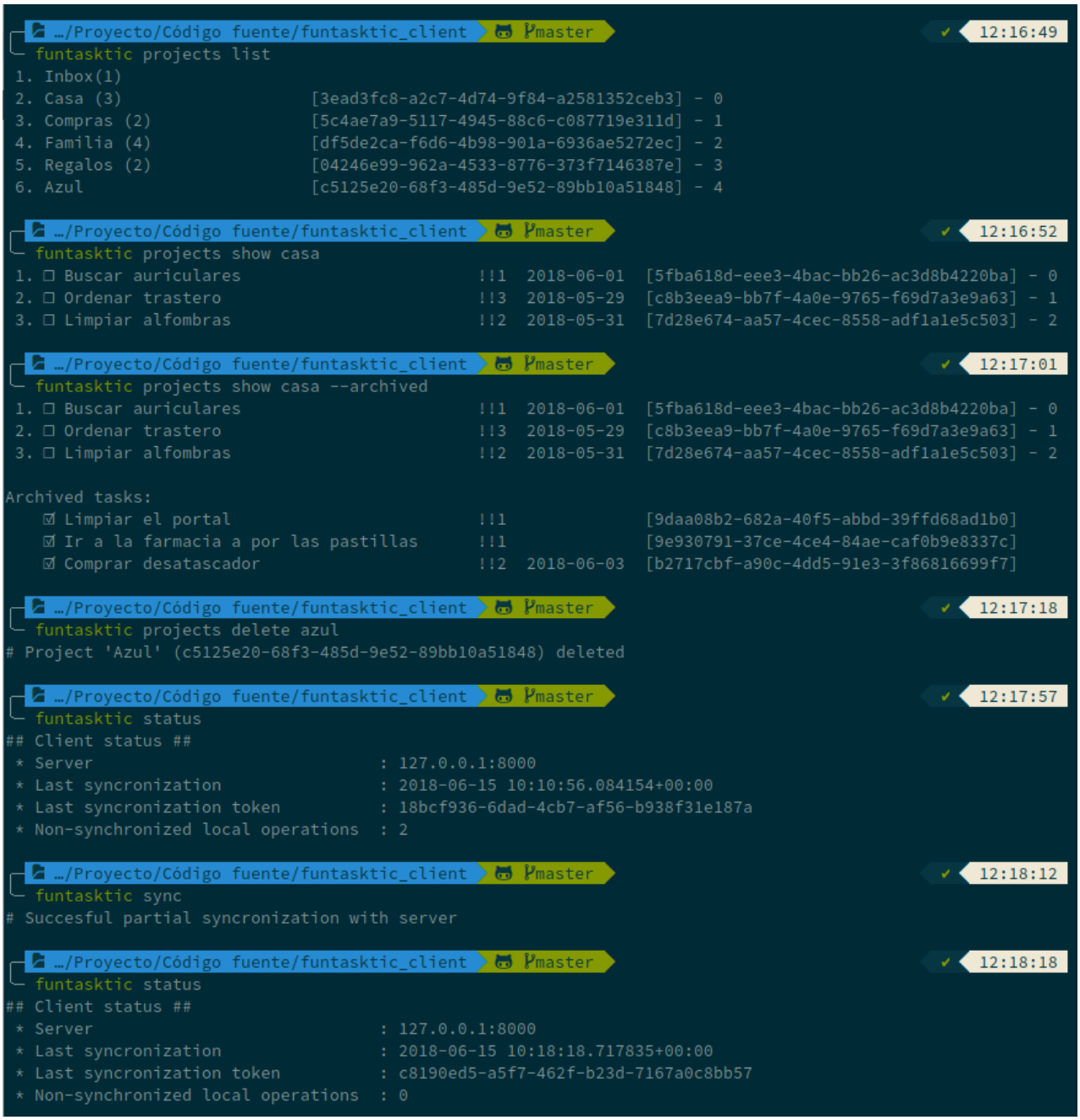

Figura 52. Ejemplo de funcionamiento y aspecto del cliente distribuido

4. Despliegue

En cuanto al despliegue de la aplicación, se ha optado por el clásico mecanismo de distribución Python basado en el gestor de paquetes pip. Para ello, se ha elaborado un fichero que declara las bibliotecas de las que dependen la aplicación, así como el nombre de la orden con la que se invocará el programa y el punto de entrada en la

aplicación (la función a la que se llamará cuando se lance la orden). Posteriormente, durante la instalación, pip consultará este fichero, instalará las bibliotecas indicadas y creará una nueva orden que apunte al punto de entrada indicado. El nombre de la orden que invoca al cliente es `funtasktic`; por tanto, todas las órdenes del cliente irán precedidas por esta palabra.

Para facilitar el despliegue, la aplicación se acompaña del fichero `README.md`, que contiene las instrucciones de instalación del cliente. Estas contemplan dos escenarios, la instalación en un entorno virtual de Python (venv) y la instalación global en el sistema. Puesto que es de esperar que la aplicación simplemente se instale para realizar pruebas de funcionamiento, se recomienda el primer método de instalación, al resultar mucho más aséptico y libre de interferir en el entorno Python existente.

6

Conclusiones

Dado el propósito educativo de este trabajo, su resultado más valioso será siempre la experiencia y el conocimiento adquiridos. En ese sentido, resulta imprescindible en este último capítulo de la memoria hacer una ejercicio de reflexión y exponer las principales conclusiones y lecciones que se han desprendido del trabajo realizado. Así, la presente sección pretende ser depositaria de la experiencia vivida, de los problemas encontrados y de la evaluación crítica de los resultados obtenidos.

1. Seguimiento de la planificación y metodología

Para realizar el análisis del seguimiento de la planificación, cabe diferenciar dos componentes: el desglose del alcance y el tiempo. El primero atendería al grado de acierto en la previsión de las tareas y componentes necesarios para llevar a cabo el trabajo, así como de las relaciones entre estas. El componente temporal, en cambio, atiende al cumplimiento de los plazos previstos para cada elemento de la planificación.

Desde el punto de vista del desglose del alcance, la planificación ha resultado plenamente certera. Durante la ejecución del proyecto se han seguido todas las fases, tareas e hitos previstos, y no han surgido conflictos o problemas de dependencias no previstas. Por lo tanto, podría considerarse un éxito en ese sentido.

Sin embargo, la dimensión temporal de la planificación ha resultado decepcionante. Durante la planificación se contempló un riesgo —por el ya mencionado acontecimiento de naturaleza personal que afectaría a la disponibilidad temporal— y, en consecuencia, se dotó al cronograma de un margen temporal de seguridad de aproximadamente tres semanas. Desgraciadamente, una vez el riesgo se materializó, la dotación temporal prevista resultó del todo insuficiente y el impacto en la planificación terminó siendo mucho peor de los esperado[1], hasta el punto de finalizar el proyecto con casi una semana de retraso respecto a la fecha límite oficial.

Si bien esta desviación pareciera inasumible, la naturaleza del acontecimiento y su incertidumbre intrínseca justificarían en parte el grave retraso. También se puede argumentar como defensa adicional que la orientación de este trabajo hacia el aprendizaje conlleva la búsqueda de nuevas herramientas, la adquisición de su dominio y su aplicación para enfrentarse a problemas desconocidos. Todas estas tareas conllevan una gran incertidumbre. En este contexto, resulta tremendamente complicado establecer plazos con ciertas garantías, puesto que se desconoce en gran medida el coste temporal de cada uno de los elementos de la planificación.

[1] Mi hija nació prematura y con enfermedad por reflujo gastroesofágico (ERGE).

2. Logro de objetivos

Desgraciadamente, no se logró cumplir con todos los objetivos planteados inicialmente. Sin embargo, dado el alto grado de cumplimiento y teniendo en cuenta las circunstancias personales sobre las que se llevó a cabo el trabajo, el balance puede considerarse muy positivo. En concreto, de los 21 objetivos iniciales (generales y específicos) se han cumplido todos a excepción de los cuatro enumerados a continuación:

1. **Entregar en tiempo y forma**
   Como ya se ha indicado, no se ha podido cumplir el plazo de entrega.
2. **Realizar una documentación completa de la aplicación**
   La documentación desarrollada del servidor web es parcial, aunque cabe indicar que la aplicación es prácticamente autoexplicativa y sí dispone de un manual de instalación (el aspecto más dificultoso).
3. **Seguir la metodología TDD**
   Se decidió prescindir de ella en los momentos de pánico que siguieron a la restricción súbita de la disponibilidad temporal.
4. **Despliegue de instancias demo en un servidor**
   Este aspecto es el menos preocupante, pues se descartó deliberadamente en su momento. La razón fundamental es que, dado que el tribunal tendría que desplegar igualmente una instancia para poder probar los *scripts* Docker, la instancia de demo no aportaría suficiente valor al trabajo.

Más allá de los problemas derivados del factor que ha restringido la disponibilidad temporal durante más de dos meses de curso, es menester realizar un poco de autocrítica. En ese sentido, la causa fundamental de las desviaciones en tiempo y cumplimiento pueden achacarse a un exceso de ambición y a un error en la previsión de la carga de trabajo de las tareas previstas.

3. Conclusiones y lecciones aprendidas

Este apartado, si bien no es el más extenso o técnico, puede considerarse el más importante del trabajo. ¿Cuál es el poso que han dejado el trabajo y el esfuerzo realizados durante cuatro meses? La respuesta a esta pregunta se presentará en forma de lista enumerada, donde el orden de las lecciones atiende a su importancia.

1. **Es extremadamente difícil dimensionar la complejidad del desarrollo de un sistema informático nuevo.**
   Es sabido que este es un problema con el que tienen que lidiar la Ingeniería de Software y la Gestión de Proyectos. De hecho, existen infinidad de libros al respecto —sin ir más lejos, el celebérrimo *The Mythical Man-Month*—. En cualquier caso, vivirlo en las propias carnes hace ver las cosas de otra manera.
2. **Las pruebas son fundamentales en cualquier proyecto de desarrollo de cierta envergadura.**
   Sin ir más lejos, durante las labores de *refactoring* resultan vitales, puesto que es fácil incurrir en errores indirectos, que resultan muy difíciles de detectar y costosos de corregir a posteriori. Además, durante el despliegue se produce una sensación de pánico, ya que afloran multitud de errores y en el peor momento posible.
   Esto se evidenció con creces en el desarrollo del proyecto. Así, cuando se decidió prescindir de la metodología TDD, inicialmente el ritmo de desarrolló aumentó

con creces. Si embargo, en las últimas fases del proyecto (cuando la carga de código existente era notable) trabajar sin pruebas resultó ser parecido a hacer funambulismo sin red: una sensación de peligro y miedo constante cada vez que se tenía que revisar el comportamiento de un componente existente. La lección es clara: en el futuro se seguirá (con mayor o menor rigor) esta filosofía de desarrollo.

3. **El desarrollo incremental es una metodología muy adecuada para contextos de elevada incertidumbre.**

   Cuando resulta difícil estimar el grado de cumplimiento en el alcance que se podrá lograr, adoptar una metodología de desarrollo incremental puede prevenir el completo fracaso del proyecto. En lugar de jugarse el desarrollo a un todo o nada, los diferentes prototipos incrementales garantizan un entregable funcional en prácticamente la totalidad del periodo de desarrollo.

4. **Los *frameworks* facilitan la labor de los desarrolladores, pero requieren de una importante inversión temporal inicial.**

   En el caso de este trabajo, tanto Django como Django REST Framework resultaron ser un universo en sí mismos. En consecuencia, fue necesario destinar una cantidad ingente de tiempo a familiarizarse con los patrones y la filosofía de trabajo de cada uno de ellos, dedicando horas y horas al estudio de su documentación. En ocasiones, podría no justificarse esta inversión temporal inicial con los beneficios obtenidos por su uso. Puede que este proyecto fuese una de ellas.

   Mención aparte merecería el uso de los módulos ORM. Si bien es cierto que facilitan las tareas rutinarias de acceso a datos sobre bases relacionales, para otras operaciones no tan triviales es necesario recurrir a artificios nada intuitivos y muy específicos de la herramienta. Con ello, se dificulta su uso y se eleva la inversión inicial en aprendizaje.

4. Futuras líneas de trabajo

En este último bloque de la memoria se expondrán las mejoras en el sistema que podrían realizarse y que, por motivos de economía de medios, no ha sido posible incorporar al presente trabajo. Dado su gran número, se presentarán brevemente como una lista numerada siguiendo un orden arbitrario:

1. Actualizaciones automáticas (*push*) de los clientes y la interfaz web.
2. Elaboración de un manual de uso completo (o un wiki) de la aplicación.
3. Envío de notificaciones por correo electrónico o SMS.
4. Añadir opciones a los usuarios: formatos de fecha, temas, etc.
5. Gestión rigurosa de las fechas y horas, teniendo el cuenta los diferentes usos horarios de clientes y servidor.
6. Permitir ordenar manualmente las tareas y proyectos.
7. Permitir anidar tareas y proyectos, dando lugar a subtareas y subproyectos.
8. Incorporar un menú contextual a la interfaz del cliente HTML.
9. Cambiar la arquitectura del cliente HTML a uno totalmente asíncrono basado en Javascript.
10. Incorporar etiquetas a proyectos.
11. Permitir formato enriquecido en notas y tareas.

12. Cadenas de texto multi-idioma.
13. Implementar el mecanismo de eliminación de elementos borrados (recolector de basura).
14. Implementar una papelera de reciclaje para acceder a los elementos borrados recientemente.
15. Implementar una gestión de etiquetas de usuario.
16. Agrupar las tareas por días en la vista *Próximos 7 días*.
17. Separar las tareas vencidas en la vista *Hoy*.
18. Registrar y poder mostrar el historial de acciones que han afectado a cada objeto.
19. Mejorar el *feedback* con el usuario en el caso de producirse errores inesperados.
20. Perfeccionar los mecanismos de resolución de conflictos entre operaciones.
21. Incorporar mecanismos de copia de seguridad.
22. Incorporar la posibilidad de importar y exportar las tareas de un proyecto.
23. Mejorar la gestión de proyectos compartidos.

Glosario

fullstack dicho de un lenguaje de programación o de las capacidades de un programador cuando estas abarcan todo el espectro de una aplicación web, desde la funcionalidad servidor (*backend*) a la interfaz del usuario (*frontend*). 10

API (*Application Programming Interface*) servicio que permite la interacción de forma programática entre un servidor y un programa cliente. 19

CLI (*command-line interface*) interfaz no gráfica cuya interacción con el usuario se basa en la recepción de órdenes y la impresión de resultados a través de una consolo textual. 28

GTG (*Getting Things Done*) metodología de organización personal del trabajo centrada en maximizar la concentración y reducir las distracciones. 15

jQuery uno de los *frameworks* Javascript más antiguos y utilizados. 11

MVC (Modelo-vista-controlador) patrón arquitectónico de diseño de aplicaciones que se basa en la separación del servicio en tres capas diferenciadas: modelo, que gestiona la estructura y el acceso a la información; vista, que se encarga de la interacción con el usuario, y controlador, que media entre el modelo y la vista y conteniene la lógica de negocio. 1, 2, 10

ORM (*Object-relational Mapping*) componente de software que se encarga de abstraer el modelo relacional de una base de datos y presentarlo mediante una API orientada a objetos. Con ello se busca reducir el denominado *desajuste de impedancia* que surge por las diferencias entre el modelo orientado a objetos y el relacional. 1, 14

SASS (*Syntactically Awesome Stylesheets*) metalenguajes de CSS que facilita un uso más programático de esta tecnología. 11

TDD (*test-driven development*) metodología de desarrollo de aplicaciones en la que las pruebas se consideran el elemento central del proceso de programación y que siempre preceden la elaboración de código. 1–3, 10

Bibliografía

Christie, Tom (2018). *Django REST framework · sitio oficial*. URL: http://www.django-rest-framework.org/ (visitado 15-06-2018).

Coulouris, George y col. (2011). *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company. ISBN: 9780132143011.

Docker · Docker Documentation (2018). Docker Inc. URL: https://docs.docker.com/ (visitado 14-06-2018).

Docker Compose · Docker Documentation (2018). Docker Inc. URL: https://docs.docker.com/compose/ (visitado 14-06-2018).

Garcia Lopez, Pedro y col. (sep. de 2015). «Edge-centric Computing: Vision and Challenges». En: *SIGCOMM Comput. Commun. Rev.* 45.5, págs. 37-42. ISSN: 0146-4833. DOI: 10.1145/2831347.2831354. URL: http://doi.acm.org/10.1145/2831347.2831354.

Gunicorn - Python WSGI HTTP Server for UNIX (2018). URL: http://gunicorn.org/ (visitado 15-06-2018).

Kosik, Sergey (27 de ago. de 2013). *Tuesday, 27 August 2013*. URL: http://havrl.blogspot.com/2013/08/synchronization-algorithm-for.html (visitado 26-06-2018).

Kurose, James F. y Keith W. Ross (2012). *Computer Networking: A Top-Down Approach (6th Edition)*. 6th. Pearson. ISBN: 9780132856201.

Lubanovic, Bill (2014). *Introducing Python*. 1th. O'Reilly. ISBN: 9781449359362.

Mark Otto, Jacob Thornton y Bootstrap contributors (2018). *Bootstrap · sitio oficial*. URL: https://getbootstrap.com/ (visitado 15-06-2018).

McCormack, Drew (2014). *Data Synchronization*. URL: https://www.objc.io/issues/10-syncing-data/data-synchronization/ (visitado 26-06-2018).

Mccormick, Zach y Douglas C. Schmidt (2012). «Data Synchronization Patterns in Mobile Application Design». En: *Proceedings of the 19th Conference on Pattern Languages of Programs*. PLoP '12. Tucson, Arizona: The Hillside Group, 12:1-12:15. ISBN: 978-1-4503-2786-2. URL: http://dl.acm.org/citation.cfm?id=2821679.2831282.

Mobile database (22 de nov. de 2017). Wikipedia. URL: https://en.wikipedia.org/wiki/Mobile_database (visitado 26-06-2018).

Nginx · sitio oficial (2018). NGINX Inc. URL: https://www.nginx.com/ (visitado 15-06-2018).

Operational transformation (12 de mayo de 2018). Wikipedia. URL: https://en.wikipedia.org/wiki/Operational_transformation (visitado 26-06-2018).

Optimistic replication (5 de mar. de 2018). Wikipedia. URL: https://en.wikipedia.org/wiki/Optimistic_replication (visitado 26-06-2018).

Pascual, Vincent Segui y Fatos Xhafa (2013). «Evaluation of contact synchronization algorithms for the Android platform». En: *Mathematical and Computer Modelling* 57.11-12, págs. 2895-2903. DOI: 10.1016/j.mcm.2011.12.039. URL: https://doi.org/10.1016/j.mcm.2011.12.039.

Percival, Harry J.W. (2017). *Test-Driven Development with Python (2nd Edition)*. 2th. O'Reilly. ISBN: 9781449359362.

pip 10.0.1 documentation (19 de abr. de 2018). The Python Software Foundation. URL: https://pip.pypa.io/en/stable/ (visitado 15-06-2018).

Reitz, Kenneth (2018). *Requests 2.19.1 documentation*. URL: http://docs.python-requests.org/en/master/ (visitado 15-06-2018).

Remember The Milk · sitio oficial (2018). Remember The Milk. URL: https://www.rememberthemilk.com/ (visitado 14-06-2018).

Ronacher, Armin (2017). *Click Documentation (5.0)*. URL: http://click.pocoo.org/5/ (visitado 15-06-2018).

Synchronization (computer science) (25 de mayo de 2018). Wikipedia. URL: https://en.wikipedia.org/wiki/Synchronization_(computer_science) (visitado 26-06-2018).

Todoist · sitio oficial (2018). Doist. URL: https://todoist.com/ (visitado 14-06-2018).

Todoist Sync-API Documentation (2018). Doist. URL: https://developer.todoist.com/sync/v7/#overview (visitado 26-06-2018).

Wunderlist · sitio oficial (2018). 6Wunderkinder GmbH. URL: https://www.wunderlist.com/ (visitado 14-06-2018).

ZODB Documentation (2017). Zope Foundation. URL: http://www.zodb.org/en/latest/ (visitado 15-06-2018).