UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Un entorno colaborativo para el diseño, desarrollo y compilación de notebooks de trabajo Jupyter compartidos.

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Javier Rodríguez Domínguez

Tutor: José Manuel Bernabeu Aubán

Curso 2017-2018

Resum

????

Paraules clau: ????, ?????????, ????, ???????????????

Resumen

El trabajo realizado ha consistido en la creación de un entorno colaborativo permisionado para *Jupyter*. *Jupyter* es una herramienta para la computación interactiva en diferentes lenguajes de programación. Ofrece una forma de trabajar muy intuitiva y visual, por lo que adaptarla a un ámbito colaborativo puede ser muy interesante para los usuarios. Para alcanzar el objetivo propuesto se han llevado a cabo una serie de tareas, como son la sincronización en tiempo real de los notebooks, la creación de un sistema de roles y permisos, la autenticación de los usuarios y el mantenimiento de sus sesiones, la persistencia de los datos y, por último, la compartición de documentos. Se ha hecho uso de tecnologías como *Tornado* para los servicios *web*, *Auth0* para la autenticación y *SQLite* como sistema gestor de la base de datos, entre otras.

Palabras clave: ?????, ???, ????????????

Abstract

????

Key words: ?????, ????? ?????, ?????????????

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

La computación interactiva permite llevar a cabo tareas de programación de manera intuitiva e inmeditata. Al ofrecer un entorno responsivo, el usuario puede realizar cambios y ver su resultado al instante, haciendo que el flujo de interacción entre él y la herramienta sea muy ágil.

Existe un gran tradición en el ámbito de la computación interactiva, siendo los primeros ejemplos terminales o *shells*. Éstos suelen ofrecer una interfaz mediante línea de comandos. Para encontrar herramientas que soporten entornos gráficos hay que avanzar hasta el uso de *notebooks*, documentos donde los bloques de código y los resultados de sus computaciones conviven en un una serie de celdas.

Existe un gran numero de plataformas que hacen uso de este sistema de notebooks, desde su introducción en 1988 con *Mathematica*[1]. Algunos ejemplos son *Maple* [2], *Matlab*[3] o *Jupyter*[4]. Es este último en el que se va a centrar el desarrollo del proyecto, que consistirá en la creación de un entorno colaborativo para su uso concurrente por más de un usuario, ya que no se ofrece de forma nativa en la plataforma. Para aquellos que no conozcan la herramienta, se va a realizar una breve introducción:

Jupyter es una herramienta para la programación interactiva en diferentes leguajes. Nace como la sucesión natural de *IPython (Interactive Python)*. Se trata de un programa que extendía las funcionalidades de la propia terminal de *Python* con la intención de que su uso fuera más interactivo. Más adelante, este proyecto pasaría a utilizar un sistema de *notebooks*. La estructura del *notebook* es bastante sencilla. Internamente es un documento *JSON*, con bloques denominados celdas, los cuales tienen una entrada *(input)* y una salida *(output)*. Para que el *notebook* sea legible, se visualiza de una manera ordenada, mostrando sólo el código y los resultados obtenidos al usuario y haciendo que el uso de la herramienta sea lo más fluido posible. Esta mejora sería bastante significativa, pues se podía mantener una persistencia entre sesiones, ya que los *notebooks* pueden ser guardados y cargados. El siguiente paso fué saltar al proyecto *Jupyter*, cuyo cambio más importante fué la posibilidad de trabajar con lenguajes diferentes a *Python*. Ésto se consigue desacoplando la lógica de la ejecución del código del servicio principal de edición de *notebooks*. El código a ejecutar se manda a un proceso que lo recibe, lo ejecuta y lo devuelve. Estos procesos se llaman *kernels*, y a día de hoy existen *kernels* para más de 100 lenguajes que pueden ser utilizados con *Jupyter*[5].

[1] https://www.wolfram.com/mathematica/
[2] https://www.maplesoft.com/
[3] https://www.mathworks.com/products/matlab.html
[4] http://jupyter.org/
[5] https://github.com/jupyter/jupyter/wiki/Jupyter-kernels

Al no facilitar de manera nativa la colaboración entre usuarios, *Jupyter* nos deja la posibilidad de trabajar mandando el *notebook* entre los diferentes autores a medida que se va modificando. Ésto es un método poco eficiente, principalmente por dos razones. En primer lugar, *Jupyter* tiene una mala interacción con los servicios de control de versiones, pues el usuario modifica las celdas, pero realmente lo que cambia es el *JSON* interno. La herramienta de control de versiones busca los cambios entre estos documentos *JSON*, y encuentra muchas más diferencias de las que se realizaron por los usuarios, pues ellos sólo alteraron las celdas, pero internamente estos cambios se representaron de una forma mucho más extensa. Ésto supone problemas en cuanto a la usabilidad de este control de versiones, pues los cambios no son fácilmente legibles. Por otro lado, *Jupyter* es ampliamente utilizado en entornos de análisis de datos. En estas circunstancias, se suele trabajar con grandes cantidades de información, que tiene que ser cargada y procesada. Cuando se comparte un *notebook*, se comparte sólo el *JSON* que lo forma, no el estado de ejecución del *kernel*. Esto quiere decir que cada vez que alguien quiere hacer una prueba con los datos, tiene que procesarlos y cargarlos en memoria, computaciones que pueden ser bastante costosas.

En resumen, trabajar de forma colaborativa haciendo uso de *Jupyter* a día de hoy es una tarea complicada debido a la naturaleza de la plataforma, y, al ser muy utilizada en este tipo de situaciones, plantea un problema que puede ser de gran ayuda resolver.

1.1 Motivación

El trabajo se ha llevado a cabo dentro del ámbito de unas prácticas de empresa en el Instituto Tecnológico de Informática[6]. Desde el instituto se me realizaron una serie de propuestas interesantes para la realización de este Trabajo de Fin de Grado. La primera fué la adaptación de la base de datos *RethinkDB* [7] para introducirla como un componente en un sistema de *PaaS*. Por otro lado se planteó la creación de un entorno colaborativo para el diseño, desarrollo y compilación de notebooks de trabajo *Jupyter* compartidos. Después de estudiar las dos opciones a fondo, me decanté por la segunda propuesta, pues ya había hecho uso de la plataforma y me había enfrentado a los problemas que supone utilizarla en un ámbito colaborativo.

Jupyter es una plataforma que puede ser utilizada para diversos fines, ya que la manera que tiene de presentar el código y los resultados es muy visual. Ésto ha hecho que sea un recurso muy utilizado en entornos donde es importante entender qué hace cada trozo de código, como por ejemplo en la educación o en el análisis de datos. En estas situaciones se trabaja mucho en grupos, y la posibilidad de utilizar *Jupyter* de manera colaborativa entre los miembros de estos grupos puede facilitar mucho el desarrollo de los proyectos.

Existen herramientas que proponen un *Jupyter* colaborativo a través de internet, como *Google Colab* [8] o *CoCalc* [9]. Estas plataformas permiten una edición de los *notebooks* simultánea, haciendo que dos o más usuarios puedan modificar los contenidos del *notebook* a la vez y sin control alguno. Desde mi punto de vista, este enfoque a un *Jupyter* colaborativo no es el más adecuado, ya que puede resultar caótico, y pienso que un entorno con políticas más restrictivas en cuanto a la modificación de los documentos puede ser más acorde con el modelo de ejecución. Además, aunque estas herramientas previamente comentadas ofrecen planes gratuitos, el poder computacional de las máquinas que utilizan

[6] https://www.iti.es/
[7] https://www.rethinkdb.com/
[8] https://colab.research.google.com
[9] https://cocalc.com/

es muy bajo, haciendo que la interacción sea tosca y pesada. Si se quieren utilizar alternativas más potentes se aplicará el coste monetario correspondiente, por lo que proponer una alternativa propia permitirá utilizar las máquinas de las que se disponga.

A nivel personal, pienso que tomar como base para mi Trabajo de Fin de Grado un proyecto real puede ser muy beneficioso, ya que tengo la oportunidad de ver como se trabaja en entornos profesionales, y más concretamente en el ámbito real de un instituto tecnológico. Además, me tengo que enfrentar a problemas nuevos para mí, como el diseño de una arquitectura o la implementación de un servicio *web*.

1.2 Objetivos

El objetivo principal del proyecto es el desarrollo de un entorno colaborativo para *Jupyter*. Entendemos desarrollo como todo el proceso creativo, desde la concepción de la idea y el diseño formal hasta la implementación final del código.

En cuanto a objetivos más acotados y específicos, tenemos los siguientes, cada uno con su correspondientes apartados de diseño e implementación:

- Permitir que los usuarios modifiquen y vean los cambios en los documentos en tiempo real.
- Gestionar los usuarios, sus permisos y sus sesiones.
- Gestionar los *notebooks* y el acceso de los usuarios a estos *notebooks*.
- Soporte de colaboración en distintas sesiones de trabajo.

Estos objetivos específicos se pueden entender como pequeños apartados que, una vez unidos y cohesionados, supondrán el cumplimiento del objetivo principal.

1.3 Impacto esperado

Se espera que la herramienta resultante facilite el uso de *Jupyter* en situaciones colaborativas, reduciendo el tiempo y esfuerzo empleado en poner en común los cambios que se vayan a realizar. También se pretende que se trabaje en tiempo real, agilizando todo el proceso de desarrollar la tarea en cuestión.

1.4 Metodología

La metodología seguida para el desarrollo del trabajo ha sido la siguiente. En primer lugar, se realizó un estudio de las herramientas similares a *Jupyter* con el fin de situarla dentro del contexto tecnológico del momento. Después se realizó un estudio a fondo del código de la plataforma, para conseguir una visión general de su funcionamiento interno. Siguiendo los objetivos marcados, se ha procedido a realizar primero un diseño estructural de cada una de las partes. Este diseño se ha comparado con otros exhaustivamente para elegir el que mejor se adecuara al apartado tratado. Más tarde se procede a implementar la parte en base al diseño, y por último a hacer pruebas con el fin de encontrar fallos o *bugs* y poder solucionarlos.

En cuanto a la interacción con el tutor y compañeros en la empresa, se han realizado reuniones periódicas a lo largo del desarrollo del proyecto. Se discuten los puntos a tratar y se proponen soluciones a los posibles problemas encontrados.

1.5 Estructura de la memoria

La memoria seguirá una estructura bastante convencional según la guías que ofrece la *UPV*. Se comenzará hablando del estado del arte en el capítulo 2, donde se hará una comparativa de *Jupyter* con plataformas similares, en los capítulos 3, 4 y 5 se planteará el problema, las diferentes soluciones propuestas para resolverlo, así como el diseño y desarrollo de éstas. En los dos capítulos siguientes, 6 y 7 se hablará de la implantación del sistema, cómo desplegarlo y ponerlo en marcha, así como de las pruebas realizadas a éste. Por último, se hará una conclusión del trabajo, exponiendo si se han alcanzado los objetivos y de qué manera, y se propondrán mejoras que se pueden llevar a cabo en un futuro, capítulos 8 y 9.

CAPÍTULO 2
Estado del arte

La plataforma *Jupyter* utiliza un modelo de funcionamiento basado en la programación interactiva. El usuario puede realizar computaciones y recibir resultados al instante, haciendo que el flujo de trabajo sea muy cómodo, ya que se pueden realizar cambios y correcciones con una gran immediatez.

La programación interactiva es un recurso que se lleva utilizando desde la década de los 60. Los ejemplos más comunes para este tipo de modelo de funcionamiento son los intérpretes de comandos o *shells*. Algunos lenguajes de programación que ofrecen una *shell* con una interfaz de linea de comandos (*CLI*) son *Python* [1], *JavaScript* [2], *Ruby* [3] o *Lisp* [4]. Es éste último lenguaje en el que se origina el concepto que sirve como base para la programación interactiva basada en intérpretes de comandos. Se trata del *REPL*, el bucle de Lectura-Evaluación-Impresión, en inglés *Read-Eval-Print-Loop*. Este bucle está compuesto por la lectura de las expresiones introducidas por el usuario (*Read*), la evaluación de estas expresiones por el intérprete (*Eval*), la impresión de los resultados obtenidos (*Print*) y, por último, se vuelve a aceptar una nueva entrada para se computada cerrándose el bucle (*Loop*). Podemos encontrar un estudio más amplio sobre el modelo de programación interactiva, y en concreto sobre *Lisp* en el artículo de Erik Sandewall [1].

Aparte de las ventajas previamente comentadas sobre este modelo de programación basado en los *shells*, encontramos muchos defectos, como pueden ser la pérdida de información entre sesiones o la ausencia de *rich output*, es decir la imposibilidad de mostrar resultados como imágenes o gráficos. Con la intención de solventar estos problemas, concretamente en el lenguaje *Python*, en 2001 Fernando Pérez desarrolló la plataforma *IPython* (*Interactive Python*) [2]. Esta plataforma extendía las funcionalidades del *shell* por defecto de *Python*, añadiendo algunas mejoras, como la posibilidad de mostrar la *historia* (las instrucciones ejecutadas anteriormente) o la identificación de errores.

El siguiente paso en el desarrollo de *Ipython* sería lograr la persistencia absoluta de los datos, ya no solo mediante el uso de la *historia* sino utilizando un sistema de *notebooks*. En 1988, Steve Wolfram desarrolla *Mathematica* [6], un programa para la computación matemática. Brian Hayes realiza una breve introducción en [5]. Este programa fué el primero en implementar este sistema de *notebooks*, que sería utilizado por *Ipython* más tarde. Los *notebooks* son documentos con celdas que sirven como entrada. El usuario puede escribir código en ellas y ejecutarlas, lo que dará un resultado que se mostrará en la parte inferior de la celda, es decir, la salida. Este sistema propone un enfoque muy visual e interacti-

[1] https://www.python.org/
[2] https://www.javascript.com/
[3] https://www.ruby-lang.org/en/
[4] https://es.wikipedia.org/wiki/Lisp

vo a los usuarios, pues pueden ver resultados parciales y hacer pruebas en fragmentos aislados de código.

Ipython había conseguido crear una plataforma interactiva para *Python*, y se dieron cuenta de que ésta podia ser trasladada a más lenguajes. Fué entonces cuando se produjo *The big split* [5] (La gran partición) y nació *Jupyter* [3]. *Jupyter* permitía la inclusión de nuevos lenguajes de programación en el *frontend* de tipo *notebook* que se había desarrollado. La tarea realizada para lograr ésto fué la de desacoplar el código de ejecución de *Python* del *frontend*. Se creó un protocolo que utilizaba mensajería *ZeroMQ* [6] y que servía como un canal genérico para comunicarse con cualquier proceso de ejecución de otro programa [7]. Estos "procesos" reciben el nombre de *kernels*.

Al igual que muchas de las *shells* de lenguajes comentadas anteriormente, la de *Python* hacía uso de un *kernel* para la ejecución del código. Para la definición de *kernel* nos podemos basar en la interprentación de Sandewall "*El kernel de un sistema de programación debe contener los siguientes programas: Un parser que transforma los programas del usuario a la representación interna. Un program-printer que realiza la operación contraria. Un intérprete para los programas en la representación interna y/o un compilador que transforma la estructura de datos interna en languaje máquina para ser computado*"[1]. Por lo tanto, el *kernel* deberá poder recibir la información, computarla y devolverla al programa principal.

De esta manera, cualquier programa que fuera capaz de recibir una entrada por medio del protocolo de *Jupyter* y devolver una salida podía realizar la función de *kernel*. Sin embargo, un *kernel* que no mantenga el contexto entre las diferentes computaciones no sirve de mucho, pues no se podrá separar el código en celdas y que las variables mantengan sus valores. Al no poder separar el código, el concepto de *notebook* pierde su sentido completamente, por lo que para que un *kernel* sea útil debe mantener el contexto.

Para definir contexto, se hace referencia a Anind K. Dey and Gregory D. Abowd [4] "*El contexto es cualquier información que puede ser utilizada para caracterizar la situación de una entidad. Una entidad es una persona, lugar o objeto que se considera relevante para la interacción entre un usuario y una aplicación, incluyendo el propio usuario y aplicación*". Este contexto va a ser el entorno sobre el que se ejecute cada computación mandada a un *kernel*, y tiene gran importancia dentro de cuaquier plataforma de programación interactiva.

Una vez comentados todos estos puntos y componentes dentro de *Jupyter*, así como brevemente la historia de la plataforma, se pasará a estudiar otras aplicaciones que ofrecen una funcionalidad o servicio similares a ella:

- ***Mathematica***. Como se ha comentado previamente, *Mathematica* es la plataforma en la que *Jupyter* se inspira a la hora de realizar su salto a el uso de *notebooks*. Es una gran referencia en cuanto a programación interactiva, por todo el recorrido que presenta. Difiere con *Jupyter* sobretodo en cuanto a versatilidad. Mientras que *Mathematica* está muy centrado en la computación matemática, *Jupyter* puede ser utilizado con una variedad mayor de fines, incluso se puede hacer uso de un *kernel* de *Mathematica* en *Jupyter* [7].

- ***Apache Zeppelin*** [8]. Se trata de la alternativa de *Apache* a *Jupyter*. Funciona de manera muy parecida, pudiendo hacer uso de diferentes lenguajes para la computación. Implementa la interfaz de *notebook*, con algunas mejoras, como por ejemplo poder

[5] https://blog.jupyter.org/the-big-split-9d7b88a031a7
[6] http://zeromq.org/
[7] https://github.com/mmatera/iwolfram
[8] https://zeppelin.apache.org/

hacer uso de formularios. También tiene integración con *Apache Spark* [9], pudiendo realizar computaciones paralelas.

Se han mostrado alternativas a la plataforma *Jupyter*, pero lo que se pretende alcanzar en el desarrollo del proyecto es poder hacer un uso colaborativo permisionado de uno de estos sistemas. En este momento, ya existen plataformas que nos permiten estas funcionalidades:

- ***Google Colab*** [10]. Es la opción que plantea *Google* para la edición concurrente de notebooks *Jupyter* por dos o más usuarios. Sigue el modelo de colaboración visto en *Google Docs* [11], donde todos los usuarios pueden realizar modificaciones en todo el documento. Hace uso de *Google drive* [12] para el almacenamiento de los *notebooks* así como para la compartición.
- ***CoCalc*** [13]. Creado por la empresa *Sagemath* Inc. ofrece un entorno colaborativo de computación matemática en la nube. Se hace uso de una estructura basada en proyectos, que se pueden compartir y en los que se pueden crear *notebooks* de *Jupyter* entre otros muchos tipos de documentos. La colaboración entre los usuarios dentro de éstos sigue el mismo modelo que *Google colab*. El almacenamiento de los proyectos se realiza en la propia plataforma *CoCalc*.
- ***Apache Zeppelin***. *Zeppelin* ofrece, a parte de todo lo comentado anteriormente, la colaboración haciendo uso de *Google Drive* de forma nativa. Los modelos de almacenamiento, edición y compartición son los mismos que en el caso de *Google colab*, pero siendo los documentos *notebooks Zeppelin*.

En conclusión, *Jupyter* nace como una evolución lógica dentro de la programación interactiva, mostrando una gran verstilidad y una infraestructura robusta. Introduce el *frontend* de *notebook* a muchos lenguajes, haciendo que puedan ser utilizados de una manera más visual. Se ha visto también que existen otras plataformas para la programación interactiva mediante *notebooks*, así como algunas que incluso nos permiten hacerlo de manera colaborativa. En la sección 2.1 se mostrarán los aspectos que difieren con nuestra visión de colaboración para estas plataformas, y en la sección 2.2 se plantearán aquellas propuestas que se adapten a nuestro punto de vista.

2.1 Crítica al estado del arte

Tal y como hemos visto en el apartado anterior, existen plataformas que ya implementan entornos colaborativos utilizando el sistema de programación interactiva mediante *notebooks*.

Estos entornos ofrecen un modelo de edición y ejecución no permisionado, basado en el concepto de que los usuarios interactuen libremente a lo largo de todo el documento. En un entorno de edición de texto ésto funciona perfectamente, como es el caso de *Google Docs*, ya que, aunque se borre alguna parte del documento creada por otro usuario, no hay dudas sobre el estado del documento, pues lo que se ve es lo que se tiene. Cuando se traslada ese modelo de edición a un entorno de programación interactiva, pueden surgir

[9] https://spark.apache.org/
[10] https://colab.research.google.com/
[11] https://www.google.es/intl/es/docs/about/
[12] https://www.google.com/intl/es_ALL/drive/
[13] https://cocalc.com/

problemas relacionados sobre todo con el contexto del *kernel* en el que se está realizando la ejecución.

Los *kernels* de *Jupyter* no siguen una ejecución coherente con el orden de celdas del *notebook*, sino que funcionan siguiendo el orden de ejecución que dicta el usuario, o, en este caso, los usuarios. Al mandar una celda a ejecución, ésta se ejecuta en base al contexto del *kernel* en ese momento. Si un usuario ha realizado una ejecución a destiempo, puede alterar el contexto que esperaba otro usuario, modificando el resultado de su computación.

Como el contexto del *kernel* es oculto para los usuarios, la resolución de problemas es muy poco intuitiva, pues no se tiene porqué saber donde se encuentra el error. Esto puede resultar en una interacción tosca y compleja entre los usuarios y la plataforma.

2.2 Propuesta

Para resolver los problemas que se plantean en la sección 2.1, se tiene que seguir un modelo de edición y ejecución más restrictivo. En este proyecto se ha planteado una modelo permisionado que solo permite la edición del documento por un usuario a la vez. La colaboración se consigue pasando los permisos de modificación entre los diferentes usuarios. De esta manera, se consigue una ejecución secuencial de las celdas, y todos los usuarios conocen el estado del *notebook* en todo momento, pues sólo uno está realizando cambios mientras el resto los observa en tiempo real.

En el caso de querer realizar una modificación del documento concurrente por dos o más usuarios habría que encontrar la manera de que éstos conocieran el contexto en el que su código va a ser ejecutado. También deberían ser avisados de los cambios realizados por el resto de usuarios así como de si éstos han modificado dicho contexto. De no cumplirse éstos puntos, nos encontraríamos ante el mismo problema mencionado anteriormente, en el que no se puede tener la certeza de qué resultado va a devolver cada ejecución.

CAPÍTULO 3
Análisis del problema

Para conocer la dimensión del problema y poder realizar un análisis, tenemos que ahondar en la estructura interna de la plataforma que vamos a modificar, en este caso *Jupyter*. De esta manera, vamos a entender qué componentes están realizando qué función, y podremos modificarlos de forma acorde a la solución que planteemos.

3.1 Análisis de *Jupyter*

3.1.1. Estructura de *Jupyter*

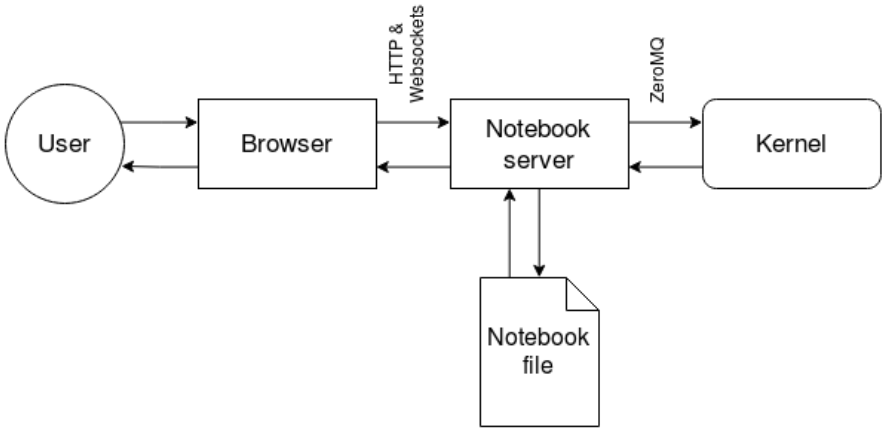

Figura 3.1: Componentes de *Jupyter Notebook*

La estructura básica de Jupyter la podemos encontrar en la figura 3.1. En ella identificamos los tres elementos principales que componen la plataforma: El servidor de *notebooks "Notebook server"*, el archivo *notebook "Notebook File"* y el *kernel*.

Servidor de *notebooks*

Es el servicio desplegado cuando se inicia la aplicación. Accedemos a él a través de la conexión mediante un navegador. En este servidor podemos ver el directorio desde el cuál se ha lanzado el proceso y seleccionar cada uno de los archivos *notebook* bien para visualizarlos y ejecutarlos o para realizar alguna otra acción como eliminarlos, renombrarlos, etc. También podemos ver en otra pestaña qué archivos están en ejecución, y detener su *kernel* asociado.

Estructuralmente, el servidor de *notebooks* es el proceso que sirve como nexo de unión entre el usuario y el *notebook*, al que accederá y modificará. Implementa todas las conexiones entre el navegador y el *notebook* (en este caso los *handlers* para las llamadas *HTTP*); así como las conexiones entre el *notebook* y el *kernel*, especificadas en la documentación de *Jupyter*, más concretamente en *The wire protocol* [7], que utilizan la tecnología *ZeroMQ*.

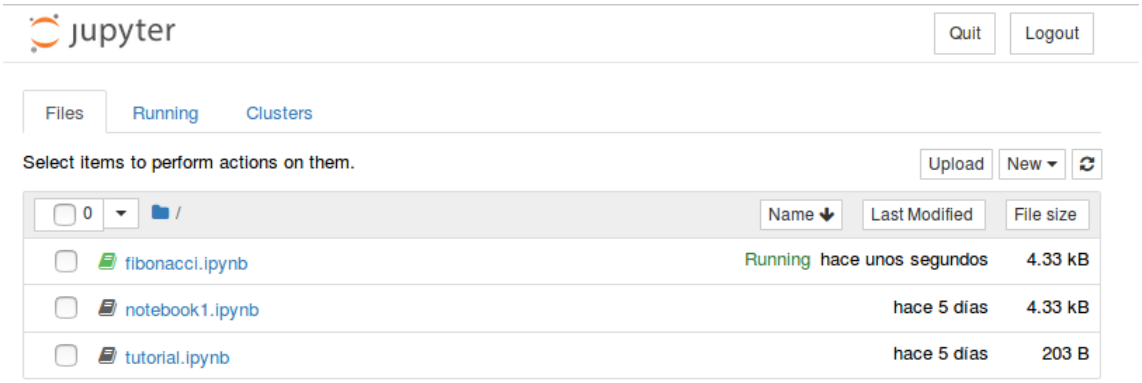

Figura 3.2: Interfaz del servidor de *notebooks*

En la figura 3.2 podemos ver una captura de la interfaz del servidor de *notebooks*. Podemos identificar cada uno de los *notebooks* en la lista, así como la pestaña *running*, donde podríamos detener la ejecución de los *kernels* en marcha.

Archivo *notebook*

Es el documento principal dentro de todo el ecosistema de *Jupyter*. Toda la plataforma va enfocada a estos archivos, donde encontramos el código y los resultados de los mismos. El *notebook* se divide en celdas, que son las unidades de ejecución, es decir, al ejecutar una celda, se ejecuta solamente el contenido de ésta. Las celdas se dividen en dos secciones, una para la entrada, donde el usuario introduce el código; y otra para la salida, donde, en el caso de que el código ejecutado en la celda tenga una salida disponible, se mostrará el resultado de la ejecución.

La función principal del *notebook* es la persistencia, pues nos sirve para no perder el trabajo realizado, ya que al finalizar una sesión éste queda guardado en disco. Los *notebooks* son internamente archivos *JSON*, donde encontramos una *array* de celdas, con sus respectivas entradas y salidas, así como los metadatos del mismo *notebook*.

La ejecución del notebook se realiza haciendo uso de los *kernels*, y la explicaré más adelante, pues hace falta tener el concepto de *kernel* dentro de la plataforma *Jupyter* claro para entender su funcionamiento.

En la figura 3.3 podemos ver un ejemplo de *notebook*. En ella se pueden apreciar detalles, como la posibilidad de mostrar celdas formateadas en *markdown* [1], o el uso de *rich output*, como podemos ver en el gráfico de la parte inferior.

Kernel

Es el componente encargado de la ejecución del código dentro de la plataforma *Jupyter*. Se trata de un proceso externo al servidor de *notebooks* donde se manda, mediante

[1] https://es.wikipedia.org/wiki/Markdown

ZeroMQ e implementando *The wire protocol* [7], el segmento de código que se quiere ejecutar. El código será ejecutado en base al contexto (estado de las variables, funciones, etc.) que se tenga en el *kernel* en ese momento, que vendrá determinado por las ejecuciones anteriores.

Existen *kernels* para más de 100 lenguajes, así como herramientas para facilitar su desarrollo, pudiéndose hacer uso de recursos como el *Wrapper* [2], que implementa toda la mensajería que comunica el *kernel* con el cliente.

FIBONACCI

A simple python implementation

In [1]:
```
def fib(n):
    if n < 2:
        return n
    return fib(n-2) + fib(n-1)
```

In [2]:
```
fib(10)
```

Out[2]: 55

In [3]:
```
k = []
for i in range(12):
    k.append(fib(i))
k
```

Out[3]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

In [4]:
```
import pandas as pd
%matplotlib inline
from matplotlib import pyplot as plt
import pandas as pd
s = pd.Series(k, index=range(12))
s.plot.bar(figsize=(5,3))
```

Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f6abfa16ba8>

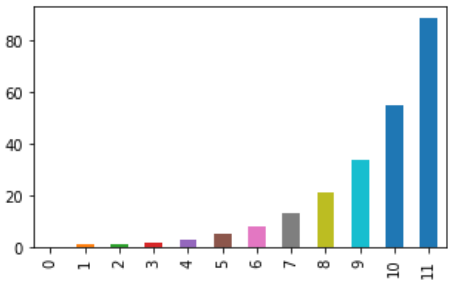

Figura 3.3: Ejemplo de *notebook*

3.1.2. Modelo de ejecución

Una vez explicados los elementos que componen la plataforma, podemos pasar a ver su modelo de ejecución. Funciona de la siguiente manera: El código que se encuentra dentro de la celda a ejecutar se manda al *kernel*. Éste lo procesa en base a su contexto, es decir, el estado de las variables guardadas en el momento en el que se ejecuta la celda. Ésto quiere decir que si se ejecuta una celda con una numeración inferior más tarde que una con una numeración superior, la inferior se ejecutará en base a lo guardado por la

[2] http://jupyter-client.readthedocs.io/en/stable/wrapperkernels.html

superior. Por lo tanto, el orden de las celdas dentro del *notebook* no tiene porqué corresponder con el orden en el que han sido ejecutadas, ya que el usuario es libre de ejecutar cada celda cuando quiera. Este modelo de ejecución puede resultar confuso, y hay que tenerlo en cuenta a la hora de buscar fallos en los resultados.

Por otro lado, cuando el *kernel* detecta que los restultados se pueden mostrar de más de una manera, por ejemplo, en texto plano o con un gráfico, manda todas las salidas posibles hacia el cliente, y es éste el encargado de decidir cuál le va a mostrar al usuario. Esto se debe a que los *kernels* no funcionan exclusivamente con *notebooks*, sino que pueden ser utilizados con otros *frontends*, como podrían ser terminales o otras aproximaciones, por ejemplo *qtconsole* [3].

3.2 Formalización del problema

Tal y como se comentaba en la sección 1.2, la finalidad del proyecto es desarrollar un entorno que permita el uso colaborativo de *Jupyter*. Para ello, tenemos que cumplir objetivos más pequeños y acotados, pues todos van a desempeñar un papel necesario para la funcionalidad de la plataforma obtenida. A continuación los expondré, segmentándolos en tareas y entrando en detalles de porqué son necesarios para el proyecto.

- **Permitir que los usuarios modifiquen y vean los cambios en los documentos en tiempo real**. El aspecto general a derivar de este objetivo es la sincronización. Necesitamos que las instancias de los documentos de los dos usuarios estén sincronizadas para cumplir este objetivo.
- **Gestionar los usuarios, sus permisos y sus sesiones**. Para ello vamos a necesitar un mecanismo de autenticación y autorización, así como un sistema de gestión de usuarios, para poder darlos de alta en la plataforma e identificarlos.
- **Gestionar los *notebooks* y el acceso de los usuarios a éstos**. Vamos a necesitar una gestión de los *notebooks*, así como hacer uso de la autorización previamente comentada para decidir si un unuasrio puede acceder o no a el *notebook* en cuestión.
- **Soporte de colaboración en distintas sesiones de trabajo**. Para soportar este objetivo va a hacer falta una capa de persistencia para mantener los datos necesarios.

Una vez se han visto los objetivos del proyecto y se han relacionado con tareas más concretas y abordables, se formalizan estas tareas y lo que se pretende conseguir con cada una de ellas. Serán los puntos a solucionar en la sección 3.4:

- **Sincronización**. La sincronización es un aspecto fundamental para una herramienta colaborativa. Nos permite conocer el estado del documento, así como los cambios producidos por otros usuarios.
- **Autenticación, autorización e identificación**. Es necesario diseñar un sistema para ver quién está modificando los documentos y si tiene permisos para hacerlo, así como asegurarse de que no pueda hacerlo si no los tiene. Una vez identificados los usuarios, se puede saber con quién se está colaborando.
- **Gestión de notebooks**. Hace falta una manera de poder añadir a nuevos usuarios a los documentos. No se puede colaborar si no hay manera de que dos o más personas editen el mismo archivo.

[3] https://ipython.org/ipython-doc/3/interactive/qtconsole.html

- **Persistencia**. Se necesita un mecanismo para mantener la identificación de los usuarios, así como sus roles y documentos asociados. De esta manera, no será necesario que se vuelva a compartir el documento cuando un usuario cierre la sesión y se vuelva a autenticar.

3.3 Análisis de la seguridad

Para garantizar a los usuarios que sus documentos no pueden ser vistos por personas no autorizadas, se ha de crear un modelo de seguridad en la plataforma. Éste debería estar muy relacionado con la autenticación y la persistencia, pues queremos comprobar y asegurarnos de que cada usuario pertenece a solamente una persona y de que estos usuarios van a ser guardados y persistidos para conexiones futuras.

Jupyter implementa de base un sistema de seguridad, en el que se pueden proteger los servidores de *notebooks* con una contraseña. Cualquier persona que conozca esta contraseña puede entrar al servidor. Este sistema no es suficiente para nuestras especificaciones, porque no identifica al usuario. No podemos saber quién se ha conectado y, por lo tanto, no le podemos aplicar su configuración y permisos.

3.4 Identificación y análisis de soluciones posibles

De la misma manera que hemos planteado y formalizado los problemas que tenemos que atacar durante la realización del proyecto, vamos a identificar y analizar las posibles soluciones para cada uno de ellos.

3.4.1. Sincronización

Para solucionar el problema de la sincronización se planteó, en una primera instancia, una sincronización por mensajería. Para ello, se haría uso de un componente llamado *Connection Bridge*, que se encargaría de las conexiones entre el *kernel* y los *notebooks*, interceptando tanto los mensajes provenientes del *notebook* que se está modificando, para mandarlos a las otras instancias y actualizarlas como los mensajes provenientes del *kernel*, para que las salidas, o *outputs*, se propaguen a todas las instancias de *notebooks* conectadas. Podemos ver un esquema de su funcionamiento en la Figura 3.4.

Esta manera de sincronizar los estados de los *notebooks* depende íntegramente en el funcionamiento del *Connection Bridge*. Éste debe actuar como *proxy*, impidiendo que el *kernel* original vea que está conectado a más de un *notebook*. Además tiene que implementar *The wire protocol* [7] tanto en su conexion con el *notebook*, pues tiene que aparecer como un *kernel* como en su conexion con el *kernel*, haciéndose pasar por un cliente.

Con este modelo, la versión del *notebook* que se está modificando, es decir, la del usuario con permisos de edición, será la que se persista en disco. Esa versión es la original, y no podemos asegurar que las instancias de los espectadores sean consistentes con ella. Si todos los mensajes se han mandado y recibido bien y el *Connection Bridge* ha funcionado sin errores, los dos *notebooks* deberían ser idénticos, pero en cuanto se produzca un error no podremos estar seguros de ello.

Por otro lado, planteamos otra manera de realizar la sincronización, esta vez sin hacer uso de componentes externos. Este nuevo modelo se basa en programar que, tanto el *notebook* que se está modificando como aquellos que se estén editando, se guarden o carguen con una frecuencia concreta desde disco. El hecho de realizar la sincronización

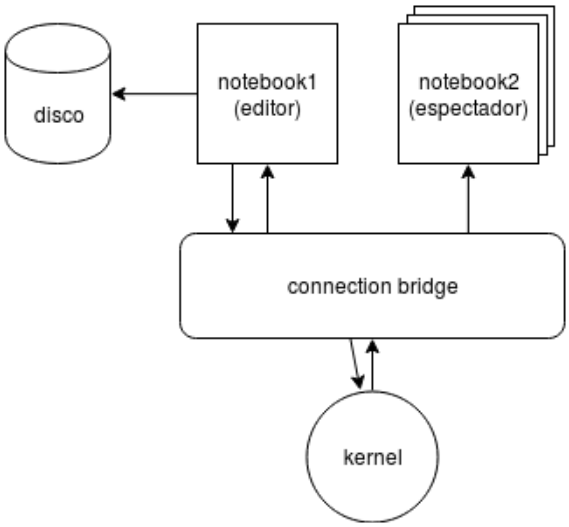

Figura 3.4: Esquema de la sincronización con *Connection Bridge*

de esta manera nos ofrece la ventaja de no tener que preocuparnos por las inconsistencias que podían surgir con la utilización del *Connection Bridge*. Al cargar el archivo desde el disco, tenemos la certeza de que va a ser consistente. Podemos ver el esquema de este modelo en la Figura 3.5.

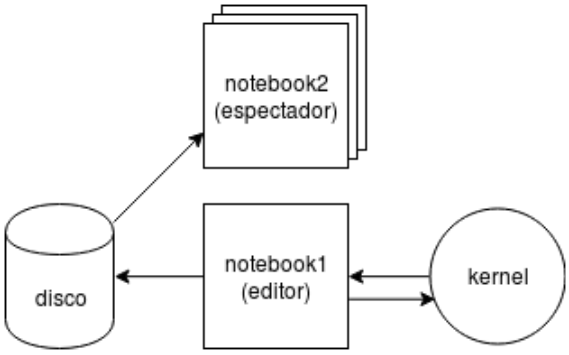

Figura 3.5: Esquema de la sincronización con carga desde disco

3.4.2. Autenticación, autorización e identificación

En cuanto a la identificación, se necesita un nombre público para que los usuarios se reconozcan entre sí. Internamente, cada usuario tendrá asignado un identificador único y se guardará una *cookie* en el navegador durante cada sesión, para no tener que identificarlo en cada petición. No se ha planteado ninguna otra solución para el problema de la identificación.

Por otro lado, para los permisos si que se han planteado diversos modelos, desde algunos con grandes libertades para todos los usuarios y basados en la confianza hasta modelos bastante restrictivos donde la autorización para realizar las acciones está muy medida.

El modelo que más hemos estudiado es un sistema de roles, de manera que cada usuario tiene un rol en cada *notebook* en el que participa. Existen cuarto roles en el sistema planteado, los cuales voy a explicar brevemente.

- **Administrador**: Se encarga de gestionar el *notebook*, pero no lo puede modificar. Es el encargado de invitar a otros usuarios a formar parte del documento, así como de renombrarlo o eliminarlo. También es el que controla quién modifica el *notebook*, puediendo cambiar los roles del resto de usuarios.
- **Editor**: Es el rol que puede modificar y ejecutar los contenidos del notebook. No puede realizar ninguna función de administración.
- **Administrador-editor**: Se trata del rol con más permisos. Es la combinación de los roles de Administrador y Editor, pudiendo llevar a cabo todas las funciones de cada uno de éstos.
- **Espectador**: Es el rol más restrictivo. Tan solo puede acceder al documento y ver los cambios que se realizan. No puede ni modificar ni administrar.

En la solución final elegida, sólo puede haber un administrador y un editor a la vez en cada *notebook*. Por lo tanto, las dos situaciones posibles son: un Administrador-editor y N espectadores o bien un Administrador, un Editor y N espectadores. Los detalles de la gestión de roles se especificarán en el capítulo 4.

Por último, para la autenticación se disponía de un gran abanico de opciones, desde algunas poco robustas, como mantener la contraseña de servidor que implementa *Jupyter* hasta otras como crear nuestro propio sistema de autenticación.

Se planteó hacer uso del sistema clásico de autenticación mediante usuario y contraseña, almacenando la contraseña *hasheada* en la base de datos. Este modelo es suficientemente robusto para nuestro sistema, y puede ser una buena opción para solucionar el problema de la seguridad.

Por otro lado, se valoró hacer uso de una autenticación mediante terceros, haciendo uso de la tecnología *OAuth* [4]. Esta opción nos permite autenticar al usuario a través de otras plataformas, lo cual es una ventaja, pues no tienen que crear una cuenta explícitamente para nuestra aplicación. Estas plataformas nos facilitarán la información necesaria para identificar al usuario en nuestra aplicación.

3.4.3. Gestión de *notebooks*

En lo que concierne a la gestión de los *notebooks*, se discutieron varias soluciones. Al igual que con la autorización, se puede optar por sistemas más o menos restrictivos a la hora de compartir los *notebooks*, desde que todos los usuarios puedan acceder a todos los documentos hasta que tengan que ser invitados para acceder a cada uno de ellos.

Esta última opción ha sido la elegida para nuestro proyecto, pues se quiere poder mantener una privacidad dentro de la plataforma, y que solamente accedan a los documentos aquellos usuarios que han sido autorizados a ello.

Para acceder a un documento, un usuario tiene que ser invitado por un administrador, en este caso entrará como espectador al documento en cuestión; o bien puede crear un documento nuevo, en el cuál será el único usuario, y, por tanto, el Administrador-editor.

3.4.4. Persistencia

Para la persistencia, desde el principio se planteó hacer uso de una base de datos, donde se almacenarán los usuarios, sus roles y notebooks en los que participan. Sus ca-

[4] https://oauth.net/

rarcterísticas y diseño serán mostradas en el capítulo 4, junto con más información sobre este componente.

3.5 Solución propuesta

En la sección anterior hemos detallado las posibles soluciones para cada uno de los problemas planteados. En este punto comentaremos cuáles han sido los elegidos para llevar a cabo en el desarrollo del proyecto.

- **Sincronización**: Se ha optado por el sistema de sincronización de guardado y cargado de archivos desde disco. Podemos ver su esquema en la figura 3.5.
- **Autenticación, autorización e identificación**: Se ha elegido un sistema de identificación mediante un nombre de usuario público, un sistema de autorización que asigna un rol por usuario para cada *notebook* y un sistema de autenticación basado en terceros, haciendo uso de la tecnología *OAuth*.
- **Gestión de *notebooks***: Se utiliza un sistema de invitaciones, mediante el cuál el Administrador de un documento puede invitar a otros usuarios para que formen también parte de él.
- **Persistencia**: Se va a hacer uso de una base de datos para almacenar la infomación que se quiere persistir.

CAPÍTULO 4
Diseño de la solución

Tomando como punto de partida las diferentes soluciones elegidas en el capítulo anterior, se ha llevado a cabo su diseño. Este diseño de los diferentes componentes que formarán la aplicación final pasa por dos fases. La primera será un diseño general, entender el sistema como un todo y viendo qué papel desempeña cada uno de los componentes dentro de éste. Trataremos este asunto en la sección 4.1 *Arquitectura del sistema*. Por otro lado, habrá que profundizar en cada uno de estos componentes, haciendo un estudio específico y detallado de todos ellos, aspectos que veremos en la sección 4.2 *Diseño detallado*. Para finalizar el capítulo, se hablará de las tecnologías utilizadas a lo largo de todo el proyecto, más concretamente en cada uno de los componentes.

4.1 Arquitectura del sistema

La arquitectura de nuestro sistema toma como base la arquitectura propia de *Jupyter*, dónde la pieza principal, que sirve como nexo de unión es el servidor de *notebooks*. A esta pieza se conectaban, originalmente, y como muestra la Figura 3.1: el navegador, los *notebooks* (tanto para su carga y guardado en disco como para el hecho de mantenerlos en memoria una vez estaban abiertos) y por último, los *kernels*.

Aparte de todos estos componentes ya nombrados, que se mantienen en el sistema final, pues se encargan de las funcionalidades indispensables de Jupyter, se van a añadir dos componentes más, que van a ser la base de datos y el sistema de autenticación. Podemos ver un esquema de la arquitectura final del sistema en la Figura 4.1.

En cuanto a las conexiones, se mantienen los protocolos utilizados originalmente, como eran *HTTP* entre el navegador y el Servidor de *notebooks* y *ZeroMQ* entre entre el Servidor de *notebooks* y el *kernel*.

Para los componentes nuevos, se utiliza *HTTP* en ambos casos. En primer lugar, la base de datos funciona como un servicio a parte del Servidor de *notebooks*. Esto quiere decir que se despliega de forma independiente, y que para acceder a los datos, el servidor de *notebooks* tendrá que hacer una petición al servidor de la base de datos. Este último hará la consulta especificada y devolverá la información pertinente. Por otro lado, de la autenticación se encarga la plataforma *Auth0* [1], la cual funciona como una herramienta de *Authentication as a service*, y que se detallará en más profundidad en la próxima sección. La conexión entre el servidor de *notebooks* y *Auth0* funciona mediante redirecciones. El usuario que se tiene que autenticar es redireccionado a una *URL* dentro del dominio de *auth0*, y, una vez autenticado, es devuelto al servidor de *notebooks*, dónde se ha enviado ya su información para poder ser identificado.

[1] https://auth0.com/

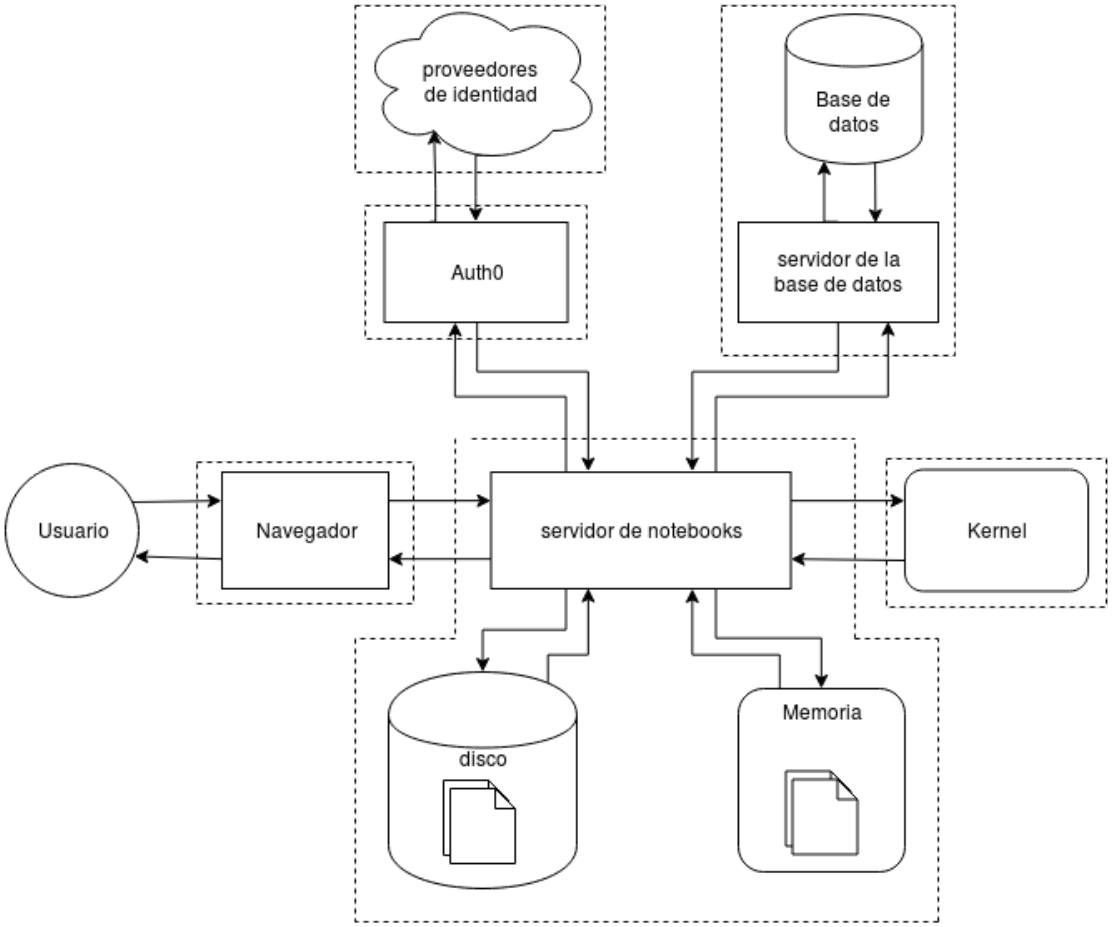

Figura 4.1: Esquema general de la arquitectura del sistema

Los diferentes procesos se han marcado con lineas discontinuas en la figura 4.1. Se puede ver como el acceso a memoria y a disco forma parte del proceso del servidor de *notebooks*, así como el acceso a la base de datos forma parte del proceso del servidor de la base de datos. El resto de componentes tienen su proceso independiente.

4.2 Diseño detallado

Una vez vista la arquitectura a nivel general que tendrá la plataforma, pasamos a ver cómo estan diseñados a nivel interno cada uno de los componentes previamente mostrados. Explicaremos el diseño la base de datos y el sistema de autenticación, ya que, a nivel de diseño, el resto de componentes se mantienen prácticamente iguales a como lo eran en el *Jupyter* original. Del funcionamiento de la herramienta resultante de nuestro proyecto hablaremos en el capítulo 5 *Desarrollo de la solución propuesta*.

4.2.1. Base de datos

Tal y como hemos explicado en el apartado anterior, la base de datos funciona como un proceso externo al servidor de *notebooks*. Ésto quiere decir que para poder acceder a ella hay que lanzar el servicio. De esta manera el servidor de notebooks puede hacer peticiones al servidor de la base de datos, el cual se encargará de realizar las consultas y devolver la información que ha sido solicitada.

Desde el punto de vista del diseño de la arquitectura, se podría haber eliminado el servidor de la base de datos, haciendo que fuera directamente el servidor de *notebooks* el que realizara las consultas. La ventaja de desacoplar el componente es que se puede acceder a la base de datos sin necesidad de hacerlo desde el servidor de *notebooks*, es decir, podemos hacer uso de la base de datos desde fuera de *Jupyter*, o tener más de una aplicación haciendo peticiones simultáneamente.

Su implementación es bastante sencilla, haciendo uso de la herramienta *Tornado* [2] para gestionar los *handlers* que se encargan de las peticiones. Por otro lado se utiliza *SQLite*[3] , que es el motor de la base de datos y la herramienta desde la cual se accede, consulta y modifica.

En lo que respecta a la organización de la información en la base de datos, podemos ver un esquema de las tablas y columnas en la Figura 4.2.

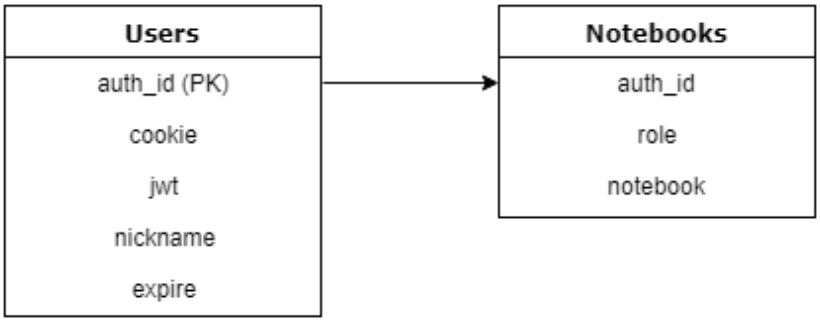

Figura 4.2: Esquema de tablas de la base de datos

La base de datos está compuesta por dos tablas. En primer lugar la tabla *Users*, donde podemos encontrar la información acerca de los usuarios registrados en la plataforma. Se compone de las siguientes columnas:

- **auth_id:** Identificador único para cada usuario, es enviado por *auth0* cuando el usuario se autentica. También constituye la clave primaria de la tabla.
- **cookie:** Se guarda la *cookie* que genera *Jupyter* para identificar la sesión del usuario. Se actualizará al cambiar de sesión.
- **jwt:** *JSON Web Token* [4] , es enviado por *auth0* al realizar la autenticación, y contiene información sobre el usuario. Se almacena encriptado.
- **nickname:** Nombre de usuario que se mostrará en la plataforma. Es enviado por *auth0* en base al proveedor de identidad que se escoja.
- **expire:** Momento en el que expira el *JWT* generado por *auth0*. El usuario se tendrá que autenticar de nuevo pasada esta fecha.

En segundo lugar, encontramos la tabla *notebooks*, que muestra la información acerca de todos los archivos *notebook* registrados en el servidor. Está compuesta por las colunas siguientes:

- **auth_id:** Es el mismo dato que en la tabla Users. Sirve como punto para relacionar las dos tablas.

[2]http://www.tornadoweb.org/en/stable/
[3]https://www.sqlite.org/index.html
[4]https://jwt.io/

- **role:** Rol asignado, puede ser uno de estos cuatro valores: Administrador-editor, Administrador, Editor o Espectador.
- **notebook:** Nombre del *notebook* que se referencia. En caso de no encontrarse en el directorio *root*, se mostrará la ruta para acceder a él.

4.2.2. Sistema de autenticación

Para llevar a cabo el sistema de autenticación se ha hecho uso de la plataforma *auth0*, como he introducido previamente en la sección Arquitectura del sistema. A continuación la introduciré y explicaré su funcionamiento.

Auth0 es una plataforma de *Authentication as a Service*. Sirve para autenticar a usuarios sin necesidad de implementar nuestro propio sistema, simplemente delegando esta tarea en ella. Para conectar nuestra aplicación con *auth0* tenemos que programar una serie de redirecciones. Cuando aparece un usuario en nuestro sistema que no está identificado se le redirige a una dirección dentro del dominio de *auth0*. Esta *URL* está asociada a nuestra aplicación, de esta manera *auth0* sabe que va a a autenticar al usuario para nuestra plataforma en concreto. Una vez aquí, el usuario tendrá diversas opciones para realizar el *login*. Estas opciones se pueden seleccionar en la configuración de la aplicación en el menú de *auth0*, y más adelante explicaremos cuáles hemos elegido. Una vez autenticado el usuario, se le devolverá de nuevo a nuestra aplicación, donde ya podrá funcionar con normalidad y acceder a sus documentos.

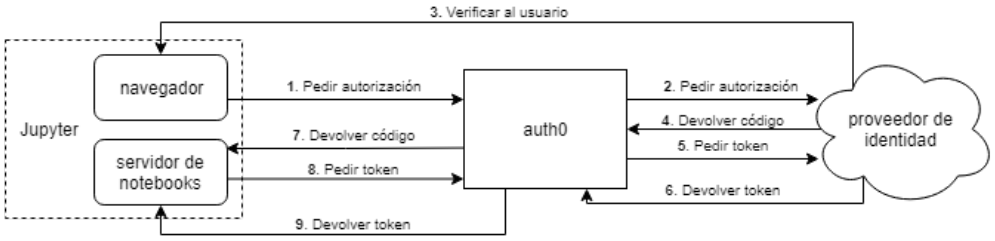

Figura 4.3: Flujo de autenticación mediante *auth0*, usando *OAuth*

La comunicación interna para que esto sea posible se puede ver en la Figura 4.3. El proceso es el siguiente:

El navegador se redirige a *Auth0* para que el usuario se autentique (*Paso 1*). *Auth0* entonces solicita una autenticación al proveedor de identidad (*Paso 2*). Después de verificar al usuario, pidiéndole que haga *login* (*Paso 3*), el proveedor de identidad le devuelve un código a a la *URL* de *callback* de *Auth0*. Al recibir el código, *Auth0* lo cambiará por un *token* de acceso haciendo una petición *POST* a la *URL* de *token* del proveedor de identidad (*paso 5*) y recibirá el *token* de acceso en la *URL* de *callback* (*paso 6*). Más tarde, otra secuencia de intercambio de *tokens* sucederá entre *Auth0* y el servidor de *notebooks*, donde *Auth0* mandará un código al servidor de *notebooks* (*paso 7*) y el servidor cambiará el código por el *token* de identificación (*pasos 8 y 9*).

Este token de identificación será el que utilizemos en nuestra plataforma. Se trata de un *JWT* que contiene la información sobre el usuario. Los *tokens JWT* están compuestos por tres partes: Una cabezera (*header*) donde aparece el algoritmo de encriptado; el contenido (*payload*), de donde extraeremos la información que hemos solicitado sobre el usuario en cuestión y, por último, una firma (*signature*), que se realiza utilizando la clave privada de aquél que envia el token, para verificar que proviene de donde dice provenir[5].

[5]https://jwt.io/introduction/

Una vez desencriptado el token, podemos acceder a su contenido, y recuperar aquella información que sea útil conocer para nuestra plataforma. Concretamente, nos interesa recuperar la siguiente información, que ya hemos introducido en la subsección anterior cuando hablábamos de la tabla *Users* en la base de datos:

- ***JWT***: Guardamos el *JWT*, ya que puede ser de utilidad más adelante porque contiene toda la información. Corresponderá a la columna *jwt* en la base de datos.
- **sub**: Es el identificador único del usuario. Tiene el formato (proveedor de identidad)|(id único dentro del proveedor). Un ejemplo sería *github|1234567890*. Corresponderá a la columna *auth_id* en la base de datos.
- **name**: Nombre de usuario que devuelve el proveedor de identidad. Corresponderá a la columna *nickname* en la base de datos.
- **exp**: Fecha en la que deja de ser válido el *JWT*. Corresponderá a la columna *expire* en la base de datos.

El diagrama de flujo que vemos en la Figura 4.3 corresponde a la secuencia de acciones llevadas a cabo en el caso de que el usuario se identifique con la aplicación de un tercero (el proveedor de identidad). *Auth0* permite la identificación de esta manera mediante muchas plataformas, como es el caso de *Google*, *Github*, *Linkedin*, etc [6]. Sin embargo, también ofrece la posibilidad de utilizar al propio *Auth0* como proveedor de identidad. En este caso, el usuario se registra con un correo y contraseña, así como el *nickname* que quiera, para ser identificado por el resto de usuarios dentro de nuestra plataforma. De esta manera, no se necesita llamar a un tercero, los pasos 2, 4, 5 y 6 vistos en el diagrama de la figura 4.3 se omitirían, y del paso 3 se encargaría el propio *Auth0*. El esquema resultante sería el de la Figura 4.4.

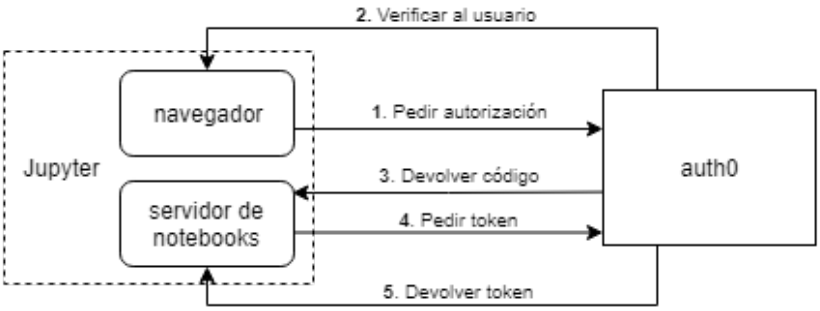

Figura 4.4: Flujo de autenticación mediante *auth0* sin proveedor de identidad externo

Para nuestro sistema hemos optado por ofrecer tanto la autenticación por terceros utilizando *OAuth* así como la posibilidad de entrar en la plataforma mediante un correo electrónico y contraseña, que será guardado directamente en el dominio de *auth0*. En la figura 4.5 podemos ver la interfaz del *login* de *auth0* para nuestra plataforma, donde se muestran las dos opciones previamente comentadas.

[6] https://auth0.com/docs/connections

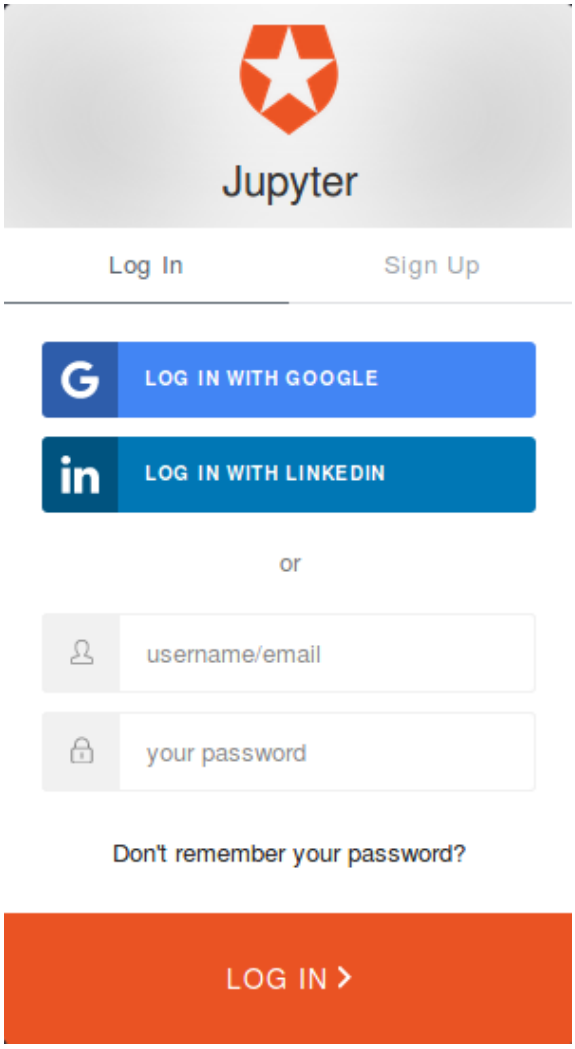

Figura 4.5: Interfaz de *login* de *auth0*

4.3 Tecnología utilizada

Hemos utilizado diversas herramientas y tecnologías a lo largo del desarrollo del proyecto. Hemos ido introduciendo muchas de ellas en lo que llevamos de memoria, en los puntos en los que resultaba relevante mencionarlas. Por lo tanto, esta sección servirá como un pequeño glosario donde se pueden encontrar todas las herramientas utilizadas sin entrar en grandes detalles sobre su funcionamiento. Estará estructurada por componentes, los vistos en la Figura 4.1, siendo cada uno una subsección de este punto.

4.3.1. Servidor de *notebooks*

El servidor de notebooks utiliza una gran variedad de tecnologías. El *backend* está programado en *Python*, haciendo uso de *Tornado* para implementar los *handlers* del servicio web. En cuanto al *frontend*, hace uso de *JavaScript* [7] y de *HTML* [8]. Los archivos *notebook* que encontramos en el servidor, ya sea en disco o en memoria, son documentos *JSON*, así que se hace uso de esta tecnología para la persistencia de la información. Por último,

[7] https://www.javascript.com/
[8] https://es.wikipedia.org/wiki/HTML

las conexiones con el resto de componentes se realizan en casi todos los casos por *HTTP*, siendo el kernel una excepción, pues hace uso de *ZeroMQ*.

4.3.2. *Kernels*

Como hemos comentado anteriormente, los *kernels* hacen uso de *ZeroMQ* para el intercambio de información mediante mensajería, tal y como está especificado en *The wire protocol* [7]. No se ha desarrollado ningún *kernel* para este trabajo, pero normalmente se programan en el lenguaje que implementan. Más información en la subsección 3.1.1.

4.3.3. Base de datos

La base de datos utiliza la herramienta *SQLite* para su gestión. Se ha hecho uso de esta plataforma porque se necesitaba una solución ligera y robusta, por lo que era una gran opción. En cuanto al servidor de la base de datos, está programado en *Python*, haciendo uso de la plataforma *Tornado* para los servicios *web* y se comunica por *HTTP* con el servidor de *notebooks*.

4.3.4. Sistema de autenticación

Tal y como hemos expuesto en la subsección 4.2.2, de la autenticación se encarga la herramienta *Auth0*. Ésta hace uso de tecnologías como *OAuth* para identificar al usuario mediante proveedores de identidad externos. También da la posibilidad de crear una cuenta en *Auth0* y que sea la propia plataforma la que autentique al usuario. Ambas opciones están contempladas en nuestra aplicación.

CAPÍTULO 5
Desarrollo de la solución propuesta

Una vez finalizado el diseño, se procede a la implementación de cada una de las partes o componentes especificados. Se procede de una manera escalonada durante el desarrollo del trabajo. En primer lugar se plantean los objetivos a cumplir, y tomando como base estos objetivos, se atacan uno por uno. Por lo tanto, se analiza el objetivo, se diseña su solución en concreto, se implementa y se prueba. En el momento en que el objetivo queda solucionado, se avanza al siguiente.

Se tomó como base para la implementación el repositorio de *Github* de *Jupyter notebook*, más concretamente su versión estable 5.5.0 [1]. Debido al gran tamaño del repositorio, fué bastante costoso familiarizarse con él y descubrir qué archivos realizaban qué funciones. Además, hubo que aprender a utilizar las tecnologías que se empleaban en el repositorio, como por ejemplo *Tornado*.

A continuación se expondrá, por orden de realización, cuál fué el proceso de desarrollo de cada uno de los objetivos.

5.1 Sincronización de *notebooks* en tiempo real

Se comenzó a realizar la implementación de la solución por este objetivo. A lo largo de todo el desarrollo y programación de las soluciones, se intentará reutilizar al máximo aquellos fragmentos de código de la plataforma original que nos puedan resultar útiles. Este es el caso de la sincronización, pues *Jupyter* realiza un guardado automático en disco siguiendo un intervalo, y, al abrir un *notebook* carga los contenidos desde disco.

Una vez identificados estos dos métodos, se procede a crear un intervalo en el que se llamará, según el rol del usuario en el *notebook* a un método o a otro, para que la instancia que ve ese usuario sea la que se cargue o se guarde (todavía no estaba implementado el sistema de roles pero ya estaba diseñado). Se fijó el tiempo del intervalo para la carga y guardado en un segundo, pues es un tiempo de espera razonable, y, en caso de ser menor, el consumo de recursos se dispara, llegando a un punto insostenible, véase la Figura 7.2.

Surgió un problema con la carga de *notebooks*, porque el método que la realiza rompe el contenedor de la interfaz sobre el que van insertadas todas las celdas del documento, y, cuando se van a insertar las nuevas celdas que se tienen que cargar, se crea otro contenedor, forzando a la ventana a hacer un *scroll* hasta arriba del todo. Ésto supone un problema para el usuario que está espectando los cambios en el *notebook*, pues no puede seguir el desarrollo que se produce, ya que cada segundo se le cambia la vista, llevándolo a la parte superior del archivo. Se solucionó este problema mediante la modificación

[1] https://github.com/jupyter/notebook/releases/tag/5.5.0

del método de cargado, haciendo que no se rompiera el contenedor, y que se eliminaran y cargaran las celdas una por una. De esta manera, al realizar el cargado, el usuario se mantiene en lugar en el que se encontraba previamente, haciendo que no pierda la vista en la que se encontraba.

5.2 Creación de un sistema de roles y permisos

El desarrollo del sistema de roles va muy ligado al desarrollo de la base de datos. El sistema de roles hace muchas consultas para conocer el rol de cada usuario en cada momento, así como para realizar modificaciones en el caso de que sea necesario.

Los roles se tienen que tener en cuenta en la aplicación principalmente en dos sitios. En primer lugar, en el ciclo de ejecución de los *notebooks*. Como se ha comentado en la sección 5.1 sobre la sincronización, se hace uso de un intervalo para el guardado y la carga de los archivos. En este punto se tiene que comprobar el rol de cada usuario, para saber si cargará o guardará el archivo, así como actualizar un *flag* creado en la interfaz para que el usuario sepa qué rol tiene. En el caso de que el usuario tenga permisos de administración se le mostrará en un submenú sus acciones disponibles, como se muestra en la Figura 5.1. Por otro lado, es importante conocer el rol para saber si un usuario tiene potestad para realizar ciertas acciones sobre un notebook, como por ejemplo renombrarlo o eliminarlo. En estos casos, en el propio *handler* de la petición, se pedirá a la base de datos que compruebe el rol del usuario, en el caso de no tener los permisos necesarios para realizarla, se enviará el código de estado *HTTP* 401, usuario no autorizado.

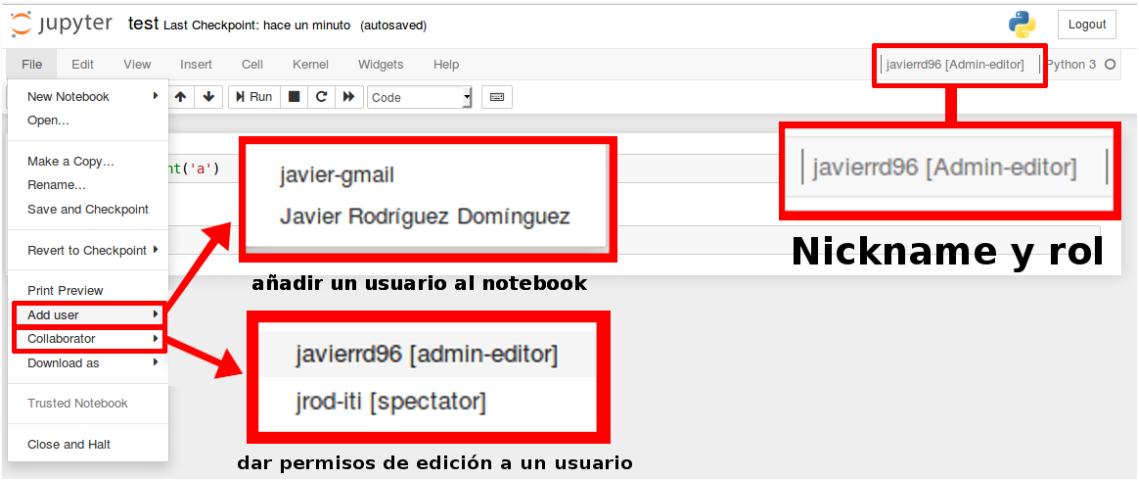

Figura 5.1: Detalle sobre los permisos del administrador

El usuario encargado de la administración de un *notebook* puede delegar los permisos de modificación en otro usuario. Para ello, aparece el submenú *Colaborator* en el menú *File* en la interfaz. El submenú es un desplegable donde se muestran todos los usuarios que forman parte del *notebook*. Este desplegable se consigue pidiéndole la información a la base de datos, e insertando cada uno de los usuarios devueltos en el código *HTML* por medio de *jquery*[2]. Cuando se hace *click* en un usuario, se manda una petición a la base de datos para convertirlo en editor. Se pueden dar los siguientes casos en función del rol del usuario seleccionado:

[2]https://jquery.com/

- **Administrador-editor o Editor**. Todo se mantiene tal y como está, pues el usuario ya tiene premisos de edición, no hay que modificar nada.

- **Administrador**. Si el usuario elegido es administrador quiere decir que existe un usuario con el rol de Editor. El Administrador pasa a Administrador-editor y el Editor pasa a Espectador.

- **Espectador**. Si se trata de un espectador, se pueden dar dos casos. En primer lugar, que exista un Editor, por lo que éste pasaría a Espectador y el Espectador elegido a Editor. Por otro lado, si no existe un Editor, tiene que existir un Administrador-editor, por lo que éste pasaría a ser Administrador y el Espectador sería el nuevo Editor.

El sistema de roles no puede funcionar sin la base de datos, porque se perdería toda la información cuando el servicio se cayera y se volviera a lanzar, ya que no se podría recordar qué usuario tiene qué rol en qué *notebook*.

5.3 Persistencia de la información necesaria

La persistencia se llevó a cabo mediante una base de datos, cuyo diseño se ha introducido en la subsección 4.2.1. Para su desarrollo se utilizó la herramienta *SQLite*, la cuál permitió realizar una implementación sencilla y robusta.

En un primer momento, la base de datos se accedía directamente desde el servidor de *notebooks*, habiendo importado la herramienta *SQLite* en este componente y realizando las llamadas en los lugares adecuados. *SQLite* trabaja con código *SQL* [3] , un lenguaje sumamente extendido, por lo que no fué complicado programar todas las consultas necesarias para el correcto funcionamiento de la plataforma, ya que la base de datos es muy pequeña (véase la Figura 4.2) y las consultas que se requieren son sencillas.

Más tarde, se optó por cambiar el acceso a la base de datos e implementarlo como un servicio *web* externo. Este cambio no supuso ningún problema a nivel de programación, y se procedió de la siguiente manera. En primer lugar se colocaron los métodos de acceso a la base de datos en un archivo *Python* nuevo. Entonces, se metió cada método dentro de un *handler* de una petición *HTTP*. Ésto se llevó a cabo haciendo uso de la plataforma *Tornado*, pues ya se conocía debido a su uso en el servidor de *notebooks*. Más tarde, se cambiaron los puntos donde originalmente se hacían los accesos en el servidor de *notebooks* por peticiones *HTTP* al servidor de la base de datos. Por último, los datos que se solicitan en las peticiones se devuelven en las respuestas *HTTP*, y son gestionados de forma asíncrona por el servidor de *notebooks*.

5.4 sesiones de los usuarios

Mantener las sesiones de los usuarios es necesarios para que no se pierda el flujo de trabajo en el caso de que, por ejemplo, se actualice la página o se cierre el navegador. En un principio, se crearon unas *cookies* que se guardaban en la base de datos e identificaban al usuario. Estas *cookies* se creaban de cero, caducaban al cabo de un tiempo, y se utilizaban para saber qué usuario estaba conectado en el navegador, manteniendo su sesión.

[3] https://es.wikipedia.org/wiki/SQL

Como un usuario puede conectarse a la plataforma desde otros ordenadores o navegadores, con este sistema se perdían sus datos, pues lo único que lo identificaba era la propia *cookie*. Al cambiar de sistema de seguridad, y utilizar una plataforma externa como *Auth0*, la *cookie* pasó a tener la utilidad que debe tener, es decir, mantener la sesión. Se eliminó la implementación de nuestra *cookie* personalizada y pasó a utilizarse una creada por *Jupyter* originalmente. Ésta se genera y guarda en la base de datos cuando un usuario se autentica. En el momento en el que el usuario se conecta desde otro navegador o ordenador, se actualiza la *cookie* en la base de datos, haciendo que cualquier otra *cookie* asociada a el usuario en otro navegador quede invalidada, haciendo que el usuario se tenga que volver a autenticar en ese navegador. Si la *cookie* llega al punto de caducar, se exige la autenticación del usuario de nuevo.

5.5 Compartición de los documentos

Respecto a la compartición de los documentos, se ha desarrollado el siguiente sistema. Cuando un usuario entra por primera vez en la plataforma, depués de realizar el *login* se encuentra con un directorio vacío, esto se debe a que no forma parte de ningún documento. El usuario tiene la opción de crear un documento, de esta manera se convertiría en el Administrador-editor de este *notebook*, véase la subsección 3.4.2. La otra opción de la que dispone es que el usuario con permisos de administración de otro documento le dé acceso. Para ello, el Administrador deberá seleccionarlo en el submenú *Add user*, en el que aparecen todos los usuarios registrados en la plataforma que no pertenezcan ya al documento.

Este submenú funciona de manera similar al explicado en la sección 5.2, insertando por medio de *jquery* los diferentes usuarios dados de alta en la plataforma (tras la pertinente consulta a la base de datos) en el *HTML* que se mostrará. Cuando se elige un usuario, se realiza una petición al servidor de la base de datos para generar una entrada con este usuario como Espectador en el *notebook* en cuestión.

A partir de este momento, se mostrará el *notebook* el directorio del usuario. Ésto se consigue haciendo una consulta a la base de datos para que nos devuelva la lista de *notebooks* en la que aparece el usuario. Esta lista, más tarde se comparará con el contenido real del directorio, y se listarán solo los *notebooks* que aparecen en ambos lados.

5.6 Autenticación de los usuarios

Tal y como se ha expuesto en la sección 5.4, la identificación de usuarios se realizaba en primera instancia mediante la *cookie* generada en el servidor de *notebooks*. Al no ser éste un método lo suficientemente robusto tanto para la seguridad de la plataforma como para el uso por parte de los usuarios, se decidió cambiar.

El nuevo sistema hace uso de la plataforma *Auth0* para la autenticación de los usuarios. Para poner en marcha este sistema, se creó una cuenta en *Auth0* y se registró la aplicación. De esta manera, se generan unas *URLs* específicas de la aplicación a las que se tiene que redirigir al usuario para que realice su *login*. También son generados el identificador de la aplicación así como una clave secreta. Por último hace falta especificar la dirección a la que se debe redirigir al usuario una vez termine su autenticación, llamada *callback URL*.

Por otro lado, en el servidor de *notebooks* se realiza un *bypass*, que consiste en redirigir a todos los usuarios no identificados en la plataforma a la *URL* de *login* de *Auth0*. En esta *URL* va incluido como argumento el identificador de nuestra aplicación. Una vez

el usuario es autenticado, se devuelve al usuario a la *callback URL* especificada en los ajustes de la aplicación de *Auth0*. Se recibe un código, que habrá que mandar haciendo una petición tipo *POST* por *HTTPS* a una *URL* de las generadas previamente junto con el identificador de la aplicación y la clave secreta. De esta manera, *Auth0* sabe que tenemos permisos para generar los *JWT*, así que se generan y se devuelven. Podemos ver una vista detallada de esta interacción en el esquema de la Figura 4.3, así como más información en la documentación de *Auth0* [4].

Con el *JWT* correspondiente al *login* del usuario en nuestro dominio, se procede a desencriptarlo, haciendo uso de la clave pública de nuestra aplicación, pues viene firmado con la privada para saber que es quién dice ser. Una vez desencriptado, se guardará la información relevante en la base de datos.

[4]https://auth0.com/docs/application-auth/current/server-side-web

CAPÍTULO 6
Implantación

Para hacer uso de la aplicación y llevar a cabo su despliegue, se deben seguir una serie de pasos, ya que hay que realizar las conexiones de forma correcta, pues si no los componentes no se podrán comunicar entre ellos.

En primer lugar, hay que crear una cuenta en la plataforma *Auth0* y dar de alta la aplicación. De esta manera coseguiremos el conjunto de *URLs* y claves necesarias para redirigir al usuario y que se pueda autenticar. También tendremos que configurar la *URL* de *callback* dentro de *Auth0*, para que la redirección del usuario de vuelta a nuestra plataforma sea exitosa. Esta *URL* deberá pertenecer al dominio en el que se ejecuta la aplicación, y hay que tener en cuenta que si se cambia, se tendrá que añadir la nueva dirección a *Auth0*.

Una vez configurada toda la información en *Auth0*, se utilizarán las *URLs* y códigos generados en el servidor de *notebooks*. Para ello, hay que modificar las llamadas que se hacen en la redirección del *login*, introduciendo las claves y direcciones que nos proporciona *Auth0*, como se ha explicado en la sección 5.6. Por otro lado, hay que ver dónde se va a lanzar el servidor de *notebooks*, ya que se necesitará una dirección para que los usuarios se conecten a la plataforma.

Por último, habrá que conectar el servidor de *notebooks* y la base de datos. Simplemente se tiene que especificar la dirección en la que se lanza la base de datos en el servidor de *notebooks*. La base de datos no inicia ninguna comunicación, por lo que no se ha de configurar nada por su parte.

Para poner en marcha el sistema, se ejecutarán el servidor de la base de datos y el servidor de *notebooks*. El directorio que aparecerá como *root* en el servidor de notebooks es aquel en el que se ejecuta la aplicación. Es necesario tener esto en cuenta, ya que si cae el servidor y se inicia desde otro punto los usuarios pueden confundirse o no saber llegar a sus archivos. *Auth0* está siempre escuchando, por lo que no es necesario hacer nada explícitamente en cuanto a esta herramienta.

CAPÍTULO 7
Pruebas

7.1 Validación del sistema

A la hora de realizar las comprobaciones sobre el funcionamiento del sistema se ha seguido una metodología empírica. Tal y como comentamos en el capítulo 5, el desarrollo de la solución se ha ido realizando por componentes o módulos dentro del sistema. Dentro de cada etapa del desarrollo de la aplicación, es decir, dentro de la impementación de cada módulo, se ha ido probando el funcionamiento del módulo en cuestión.

Para las comprobaciones relativas a cada componente se plantean casos de prueba que cubran el mayor conjunto de casos reales posibles. Se realizan todos los *tests* propuestos sobre el componente y se valora, según los resultados obtenidos, si el componente cumple con su funcionalidad de manera correcta. En el caso de no hacerlo, se revisa la implementación para detectar los errores y corregirlos.

Las pruebas han sido acumulativas a lo largo del desarrollo del proyecto, por lo que, al probar un módulo y comprobar que éste funciona correctamente, hay que ver que ninguno de los implementados anteriormente haya dejado de funcionar. En el caso de que alguno dejara de funcionar habría que resolver los conflictos entre los dos componentes.

Una vez completada la implementación del proyecto se han realizado pruebas a nivel global para asegurarnos de que el sistema cumple con todos los objetivos correctamente. Estas pruebas se han llevado a cabo como casos de uso, simulando las interacciones que se podrían realizar en la plataforma en un entorno real.

7.2 Pruebas de carga

Se ha realizado un estudio de la velocidad de refresco de los documentos en base a su tamaño. El intervalo de refresco predeterminado de los *notebooks* es de un segundo, como se explica en la sección 5.1. Este intervalo es la frecuencia con que los *notebooks* se guardan, en el caso del usuario con permisos de edición, y se cargan, en el resto de casos. Esta frecuencia se reflejará de una forma fiel siempre que se pueda realizar la computación en un tiempo menor o igual al programado. Si se tarda más en guardar o cargar el notebook nunca se podrá cumplir este tiempo especificado.

Para comprobar en que punto se sobrecarga el sistema, se ha utilizado un *notebook* compartido. Todas las celdas de este documento tenían tan solo una línea de código escrita. Por tanto, como las medidas se realizan por celdas, este sería el tiempo mínimo a obtener, ya que más líneas implicarían más información y más tiempo de refresco.

Las pruebas se han realizado en un equipo con un procesador *Intel Core i7-4790* y 16 *GB* de memoria *RAM*. El sistema operativo ha sido *Ubuntu* en su versión 16.04 y el navegador *Mozilla Firefox* en su versión 60.0.2.

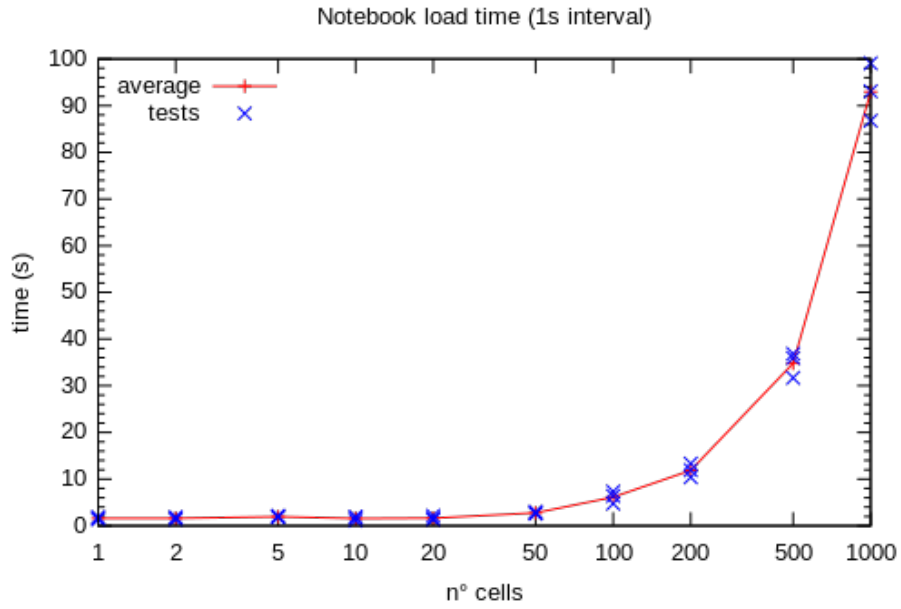

Figura 7.1: Gráfico de relación del número de celdas y tiempo de cargado de un *notebook* con un intervalo de 2 segundos

En la figura 7.1 podemos ver los resultados obtenidos en esta prueba de carga. En el eje X se muestra el numero de celdas del notebook y en el Y los segundos que ha tardado en verse reflejado el cambio. Se han realizado tres mediciones para cada *notebook*, reflejadas con las cruces azules, y se ha obtenido la media, que se muestra con la línea roja.

Se puede apreciar que mientras el *notebook* tiene menos de cien celdas se mantiene un intervalo de carga razonable, muy cercano a los dos segundos programados. En cuanto se supera este valor, el tiempo de refresco crece exponencialmente, tardando al rededor de diez segundos con doscientas celdas, treinta con quinientas celdas y hasta ciento diez con mil celdas. Podemos determinar que es este punto en el que el sistema se satura, haciendo que trabajar con *notebooks* de estas dimensiones no resulte una experiencia tan fluida.

En la figura 7.2 se realiza una comparativa de el teimpo de carga de los notebooks respecto al numero de casillas y el intervalo de refresco seleccionado. Cabe destacar que en esta figura, a diferencia de la 7.1, se utiliza una escala logarítmica en el eje *Y*, lo que hace que las curvas no sean tan pronunciadas. Podemos ver como más o menos todos los valores se mantienen estables y cerca del intervalo seleccionado hasta las 50 celdas. En este punto empiezan a crecer, llegando todos a un tiempo de carga similar con 200 casillas. Después el intervalo de 0.5 segundos se dispara, mientras que los otros tres siguen un desarrollo similar entre ellos.

Se ha elegido el intervalo de un segundo porque, aunque no sea práctica habitual el uso de notebooks con más de 100 celdas[1], en el caso de llegar a esa cantidad, un intervalo menor añadiría demasiado tiempo de espera. En el caso de no llegar a un numero tan elevado, tan solo estamos perdiendo al rededor de medio segundo en los refrescos, lo cual es un coste asumible.

[1] https://blog.jupyter.org/we-analyzed-1-million-jupyter-notebooks-now-you-can-too-guest-post-8116a964b536

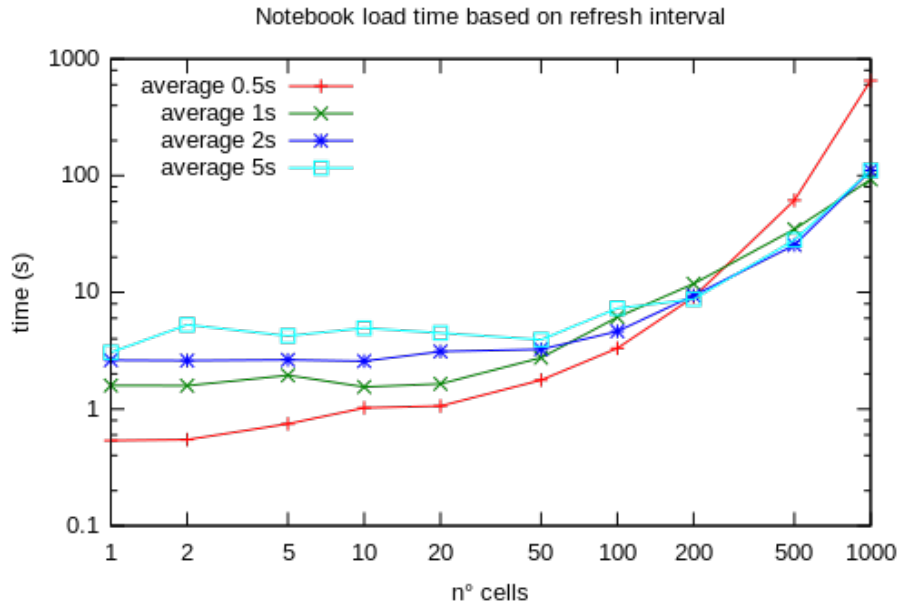

Figura 7.2: comparativa de tiempo de cargado de *notebooks* respecto al numero de celdas y frecuencia de refresco

CAPÍTULO 8
Conclusiones

Para concluir la memoria, se van a recuperar los objetivos planteados en la sección 1.2 de la introducción. Se analizará hasta qué punto se han cumplido esos objetivos, así como en qué aspectos se podrían mejorar. Las mejoras serán comentadas de forma breve, pues en el capítulo 9 se verán los trabajos futuros que se pueden aplicar a este proyecto. Se tratarán también los conocimientos aprendidos, así como la familiarización con nuevas tecnologías o herramientas empleadas en el desarrollo del proyecto. Por último, se finalizará el capítulo hablando de la relación del desarrollo del trabajo con los estudios cursados.

8.1 Cumplimiento de los objetivos

8.1.1. Sincronizar los *notebooks* en tiempo real.

La sincronización en tiempo real de los documentos se ha conseguido llevar a cabo. Se ha utilizado la solución que hacía uso de la carga y guardado de *notebooks* en disco, cuyo análisis se puede ver en la subsección 3.4.1. Este modelo es funcional, pero tiene aspectos de mejora, como pueden ser el consumo innecesario de recursos, pues se está haciendo un *polling* constante para cargar los archivos; así como el aumento de tiempo de interacción a medida que los *notebooks* van siendo más pesados, ya que no da tiempo a que se carguen los documentos enteros dentro del intervalo especificado.

Una mejora a considerar sería, por ejemplo, la actualización únicamente de las celdas modificadas. Ésto arreglaría los dos problemas previamente comentados, pues si un documento no ha sido modificado en el intervalo de tiempo, no se cargaría nada, y, en el caso de haber modificaciones, el consumo de recursos sería pequeño independientemente del tamaño del documento.

En conclusión, la sincronización de los *notebooks* se ha conseguido, ya que hemos llegado a un punto donde se puede hacer uso de la plataforma sin que sea una gran inconveniencia. Sin embargo, hay margen de mejora, y se podría llegar a soluciones más eficientes, con un menor consumo de recursos y tiempo de espera.

8.1.2. Crear un sistema de roles y permisos para los usuarios.

La creación de un sistema de roles se ha realizado correctamente. Todos los permisos que se pretenden representar en la aplicación están incluídos en algún rol. Los roles están bien diferenciados y cumplen su función dentro del entorno colaborativo.

El sistema que se ha implementado en el proyecto solo permite un editor a la vez. La colaboración se consigue pasando el rol de editor entre los diferentes usuarios, así como haciendo que todos ellos vean los cambios en tiempo real. Una de las mejoras que podría tener nuestro sistema sería la posibilidad que se editara simultáneamente el documento por más de un usuario. Ésto es un tema delicado, pues el diseño de un sistema con más de un editor concurrente presenta diversas complicaciones. Esta idea se detallará en el capítulo 9.

Por tanto, la creación del sistema de roles se puede considerar exitosa. En cuanto al modelo de interacción, el elegido es posible que resulte un poco simple y limitado, y sería un aspecto a mejorar en versiones futuras, aunque cumple su trabajo y permite una colaboración fluida.

8.1.3. Autenticar a los usuarios.

El objetivo de la autenticación se ha cumplido con creces. La plataforma *Auth0* nos ofrece una autenticación segura y robusta, haciendo uso de tecnologías muy extendidas como *OAuth*. Es una solución sencilla para los usuarios, ya que pueden utilizar la identificación de un tercero o bien crear una cuenta con su usuario y contaseña.

Auth0 nos permite añadir más capas de seguridad, como aumentar los requisitos necesarios para las contraseñas en nuestro sistema o utilizar verificación en dos pasos. No se ha hecho uso de estas opciones en la aplicación porque no se estiman necesarias, sin embargo, en caso de querer utilizarse, es tan sencillo como seleccionarlo en el menú de configuración de *Auth0*, por lo que sería un cambio immediato.

En conclusión, la autenticación se ha realizado de una manera muy completa, utilizando tecnología sencilla, moderna y robusta.

8.1.4. Compartir los documentos.

Es posible compartir los documentos dentro de la plataforma, pero es un aspecto que esta sujeto a mejoras considerables. El sistema de compartición actual muestra una lista de usuarios registrados en la plataforma. De esta manera se puede seleccionar al usuario que se quiere invitar al documento para que pase a formar parte de él. Uno de los problemas que plantea este sistema es precisamente el hecho de que los usuarios que se quieren añadir a un documento tienen que estar obligatoriamente dados de alta en la plataforma.

Una solución para este problema sería disponer de la posibilidad de compartir el documento bien mediante un *link* o un correo electrónico, de esta manera se podría invitar a usuarios externos a la plataforma a participar en documentos, haciendo que se registraran a través del enlace recibido y aparecieran directamente como Espectadores en el documento en cuestión.

Por lo tanto, la compartición de los documentos es posible y funcional, aunque se podría mejorar para aumentar su versatilidad.

8.1.5. Persistir la información necesaria.

La persistencia de la información se ha conseguido mediante el uso de una base de datos. Cumple todas las funciones que se requieren para el componente, por lo que el objetivo queda bastante completo. En cuanto a las mejoras disponibles, por el momento

no es necesario almacenar más información. En el caso de hacer falta, se debería rediseñar la estructura de tablas de la base de datos en función a la necesidad.

En conclusión, la base de datos es una solución completa para el problema de la persistencia, y se puede modificar para adaptarse a la información necesaria en caso de ésta cambiara.

8.1.6. Mantener las sesiones de los usuarios.

El guardado de las sesiones de los usuarios se ha conseguido haciendo uso de *cookies*. Se utiliza una *cookie* que genera el servidor de *notebooks*, y que funciona como identificador del usuario. Esta *cookie* se almacena en la base de datos y obliga a que el usuario se vuelva a autenticar una vez caduca.

Se trata de una solución muy común para mantener las sesiones, y que, en el caso de nuestra aplicación, funciona correctamente. Por lo tanto, podemos decir que el objetivo está cumplido.

8.2 Conocimientos adquiridos

A lo largo del desarrollo del proyecto he trabajado con tecnologías que no había tratado durante los cuatro años del grado. Ésto me ha hecho tener que investigarlas, leer sobre ellas, buscar ejemplos e intentar conocer su funcionamiento para poder utilizarlas en la aplicación que se desarrollaba. He seguido un procedimiento bastante autodidacta, aprendiendo por mí mismo a enfrentarme a un problema, buscar soluciones, compararlas y elegir aquella que mejor se adapte al problema planteado.

Por otro lado, he aprendido a trabajar en un entorno real y con una aplicación establecida y profesional. El hecho de tomar como punto de partida para el proyecto el repositorio de *Jupyter* me ha ayudado a comprender la forma de trabajar de un equipo de expertos, me ha acercado al mundo del *software* libre y me ha obligado a estructurar mejor mi código, siguiendo las guias de estilo de la plataforma.

A continuación, mostraré una breve lista de aspectos concretos y tecnologías que he utilizado y aprendido a lo largo del proyecto:

- Profundización en el protocolo *HTTP*, así como la creación de un servidor web y el uso de la herramienta *Tornado*
- Conocimiento de la plataforma *Auth0*, su uso e integración en una aplicación, además del protocolo *OAuth*.
- Ampliación de mis conocimientos sobre bases de datos, así como la puesta en marcha y uso de una.
- Trabajo con navegadores, *HTML* y *JavaScript* en ellos, además del uso y programación de *cookies*

Pienso que este trabajo ha sido muy beneficioso para mí, pues he aprendido mucho de él. Todos estos aspectos mencionados anteriormente no los aprendí en el grado, pero si que he puesto en práctica muchas otras cosas aprendidas en ese periodo, como comentaré en la sección 8.3.

8.3 Relación del trabajo desarrollado con los estudios cursados

Los conocimientos que adquirí en el grado en igeniería informática han tenido una gran importancia en el desarrollo de este proyecto. Desde mi punto de vista, la carrera me ha formado a un nivel básico en la gran mayoría de áreas de la informática, y ha profundizado en algunas de ellas, más concretamente en los campos relacionados con la rama que yo elegí, la computación. Han sido los primeros los que más han intervenido en el desarrollo de este proyecto.

Haciendo un barrido de términos generales a específicos, los contenidos aprendidos en el grado aplicados en este trabajo, han sido los siguientes:

- **Programación**: Este proyecto habría sido imposible de llevar a cabo sin los conocimientos sobre programación aprendidos. La programación es un concepto muy amplio, y se ha tratado desde muchos puntos de vista en los estudios cursados, así que es complicado definir bien todos y cada uno de los aspectos que se han utilizado. Algunos ejemplos son el uso de clases y objetos, la herencia o las estructuras de datos. En cuanto a lenguajes de programación, había trabajado ya con algunos de los utilizados, como son *Python* y *JavaScript*.

- **Bases de datos**: En el transcurso de la carrera, se había trabajado con bases de datos, lo que ha hecho que la implementación de este aspecto en concreto haya sido extremadamente sencilla. Ya conocía el lenguaje *SQL*, así que he podido realizar las consultas sin que eso supusiera un gran esfuerzo. Es cierto que nunca había creado y puesto en marcha una base de datos desde cero, por lo que he ampliado mi conocimiento en ese punto.

- **Diseño de *software***: Los conocimientos aprendidos sobre el desarrollo de *software* han sido de gran importancia al diseñar la arquitectura del sistema, los componentes y sus interacciones. Es primordial esquematizar y planificar cómo se va a afrontar el problema antes de empezar a implementar la solución.

- **Redes y mensajería**: Ha sido de gran ayuda el conocimiento sobre redes aprendido en el grado, pues se ha hecho uso de protocolos como el *HTTP*. Aunque no se había trabajado con este protocolo en mucho detalle, los conocimientos de la materia me han hecho mas llevadero el proceso. En cuanto a la mensajería, he utilizado lo aprendido para comprender el funcionamiento de los mensajes entre el servidor y los kernels. De hecho, ya había trabajado específicamente con la herramienta *ZeroMQ*.

A parte de los aspectos técnicos aprendidos, cabe destacar la importancia que tiene el hecho de trabajar la resolución de problemas. En el transcurso del grado, se insiste en que el alumnado se enfrente a problemas de manera independiente. De esta manera, cuando el alumno se enfrenta a un problema real, sabe por donde atacarlo y puede llegar a conclusiones y resultados por si mismo. Estos conocimientos han sido vitales a la hora de realizar el trabajo, pues se trata de un desarrollo en solitario, y puede resultar abrumador plantear problemas sin tener un conocimiento básico de como empezar a resolverlos.

CAPÍTULO 9
Trabajos futuros

Tal y como se ha comentado en el capítulo 8, existe un gran margen de mejora para nuestra aplicación. El objetivo principal, que era la creación de un entorno colaborativo se ha logrado, sin embargo, éste podría ser más completo. Se debe tener en cuenta que el Trabajo de Fin de Grado tiene una carga lectiva determinada, en concreto 12 créditos *ECTS*, así como unos plazos establecidos, por lo que en algún momento se tiene que detener el desarrollo del proyecto. Éste ha sido el motivo principal por el que estos trabajos futuros que se expondrán a continuación no han podido ser incorporados en el trabajo realizado.

Los dos aspectos que disponen de un margen de mejora más amplio, desde mi punto de vista, son la sincronización de los *notebooks* y el modelo de edición y ejecución de nuestra plataforma, como se comenta en las subsecciones 8.1.1 y 8.1.2 respectivamente. Se han planteado y diseñado algunas mejoras para estos dos objetivos del proyecto, como vamos a ver en las secciones de este capítulo.

9.1 Mejoras en la sincronización de *notebooks*

Se ha tratado en diferentes apartados de la memoria, como en la subsección 3.4.1 o en la sección 5.1, la solución llevada a cabo en el proyecto frente al problema de la sincronización. Consiste en sincronizar los diferentes documentos mediante la carga y guardado en disco de éstos. Esta solución se puede mejorar, tanto en el aspecto de la eficiencia, consumiendo menos recursos, así como en velocidad, reduciendo el tiempo de espera que existe entre que una modificación es realizada hasta que es vista por los espectadores.

La mejora propuesta consiste en realizar la carga de los documentos celda por celda. En el sistema actual, se carga el documento entero cada vez, lo que quiere decir que independientemente de los cambios que se hagan, se va a tardar lo mismo, pues se realiza una carga del archivo al completo. Con el nuevo sistema, el tiempo de carga dependerá del tamaño de las modificaciones realizadas desde el último guardado. Como el intervalo de tiempo utilizado es tan solo dos degundos, se puede esperar que estos cambios sean pequeños generalmente. Además, como solamente se actualizan las celdas modificadas, en el caso de no haberse producido ningún cambio desde el último guardado, el *notebook* directamente no se actualizará, por lo que se ahorrará en recursos en aquellos ciclos en los que el *notebook* no contemple cambios.

Para llevar a cabo esta mejora, se debe implementar un método de detectar qué celdas han sido alteradas y qué celdas permanecen iguales. Uno de los problemas que puede surgir es que las celdas no tienen una numeración explícita dentro del *JSON*, simplemente

se ordenan por orden de aparición. *Jupyter* nos da la opción de cambiar el orden de las celdas o incluso de fusionar dos celdas en una. Ésto puede ser una fuente de problemas a la hora de realizar las actualizaciones celda por celda. Se puede detectar de manera sencilla qué celdas han sido modificadas, pero al no tener una numeración explícita, no se pueden relacionar en el caso de haber cambiado de orden. Por ejemplo, si se intercambia el orden de las celdas 1 y 2, acción que *Jupyter* permite, nuestro sistema actualizaría las dos celdas en el documento del espectador, en lugar de intercambiarlas, ya que detectaría que las dos han sido modificadas. Este caso no es problemático, porque se puede asumir el coste de actualizar dos celdas. Sin embargo, si se fusionan las dos primeras celdas de un documento, el sistema relacionaría la celda 1 del nuevo documento (originalmente celdas 1 y 2) con la celda 1 del antiguo, la celda 2 del nuevo (originalmente la 3) con la 2 del antiguo, etc. Ésto resultaría en una actualización total del documento, ya que, desde el punto de vista del sistema, se han cambiado todas las celdas.

La solución a este problema es mantener las relaciones de igualdad entre celdas independientemente del lugar que ocupen en el documento, es decir, saber detectar si la antigua celda 4 es la nueva 5, analizando si algunas celdas han sido añadidas o eliminadas. Siguiendo este modelo, a la hora de cargar el nuevo notebook, para cada celda del antiguo se podrán realizar cuatro acciones:

- **Crear la celda**. En el caso en el que se detecte que una celda ha sido añadida en algún punto, moviendo todas las celdas posteriores, se tendrá que crear una nueva celda en el notebook espectador.
- **Actualizar la celda**. Consiste en el borrado y cargado de la celda en cuestión, se dará en el caso de que la celda haya sido modificada, pues no afecta a la alineación del resto de celdas.
- **Eliminar la celda**. Si una celda ha sido borrada por el editor, se deberá eliminar consecuentemente en el notebook del espectador.
- **No alterar la celda**. En el caso de que la celda mantenga su posición y contenido, no hay realizar ninguna acción.

Para saber qué celdas se tienen que crear, actualizar o modificar, el proceso encargado de la comparación tiene que identificarlas y mandarlas para que el Espectador pueda actuar en consecuencia.

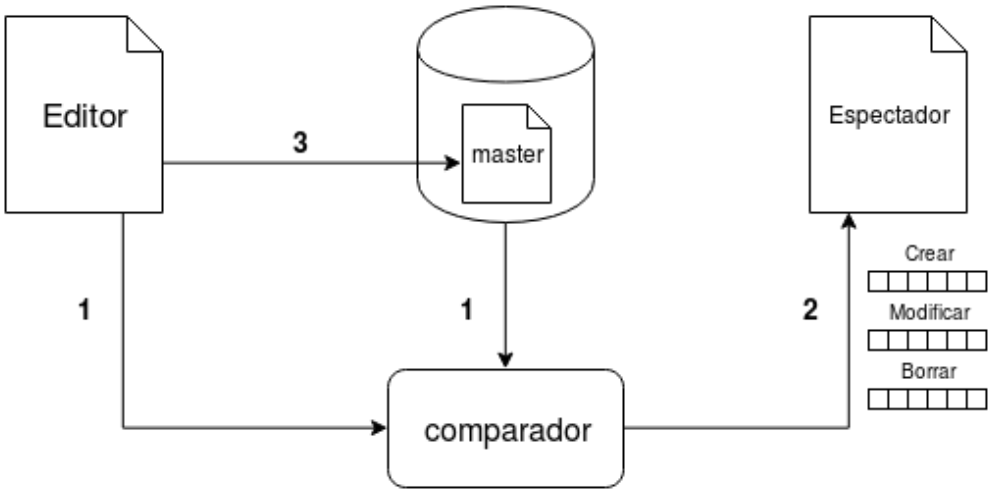

Figura 9.1: Esquema del ciclo de ejecución de la nueva sincronización

El ciclo de ejecución de este nuevo modelo se puede apreciar en la Figura 9.1. Podemos ver como se envian los dos documentos al proceso de comparación, el *notebook* del editor, que contiene los cambios realizados desde el último guardado, y el *notebook master*, que es el *notebook* guardado en el ciclo anterior (*paso 1*). El proceso encargado de comparar los dos *notebooks* (comparador) estudiará los cambios realizados en el *notebook* Editor respecto al *master* y los agrupará en celdas creadas, modificadas y eliminadas. creará tres *arrays* con los índices de las celdas en cada caso y los enviará al Espectador (*paso 2*). El espectador se encargará de actualizar su versión del *notebook* en base a las *arrays* recibidas. Por último, se guardará el *notebook* Editor en disco, para ser tratado como *master* en la próxima iteración (*paso 3*).

9.2 Mejoras en la edición y ejecución de la plataforma

El modelo de edición y ejecución de nuestra aplicación ha sido comentado en la subsección 3.4.2 y la sección 5.2. Se trata de un modelo con tan solo un editor concurrente, en el que ese permiso se va pasando por los diferentes usuarios con acceso al notebook que se está editando para conseguir la colaboración. El modelo de ejecución se mantenía inalterado con respecto al *Jupyter* original, pues al tener solamente un editor en cada momento, todas las ejecuciones de celdas van a ser secuenciales, es decir, independientemente del usuario que esté editando, en ese momento solo ese usuario podrá ejecutar las celdas del *notebook*.

El modelo de ejecución de *Jupyter* funciona de una forma peculiar, como se explicó en la subsección 3.1.2. Las celdasse ejecutan en base al contexto guardado en el *Kernel* en el momento en el que se envian. Este contexto del *Kernel* está formado por el estado asignado a las variables en las ejecuciones anteriores.

Al solo poder ser editados los documentos por un usuario en el *Jupyter* original, este modelo tiene sentido, porque es el usuario el que decide en que orden ejecuta las celdas, y, por tanto, el responsable de llevar el control de el contexto en cada punto. En el momento en el que se pretende que más de un usuario pueda realizar ejecuciones en un mismo documento, el modelo de ejecución original de *Jupyter* empieza a no cumplir su funcióm de una forma efectiva, pues las ejecuciones de un usuario pueden romper el contexto que otro esperaba en cierto punto del *notebook*.

Imaginemos que el usuario *A* está editando la celda 3 del documento, y, a su vez, el usuario *B* está modificando la celda 5. El usuario *B* había preparado una variable llamada *sol* en la celda 4, para su uso posterior. Si el usuario *A* utiliza esa estiqueta para cualquier otra variable y ejecuta, cuando *B* intente ejecutar la celda 5, su resultado no será el esperado, pues el valor de esa variable habrá cambiado en el contexto del *kernel*, y se aplicará el último valor guardado, devolviendo un resultado diferente al esperado por *B*.

Esta situación es dificil de resolver, porque se necesita que cada usuario pueda realizar las ejecuciones dentro del contexto que necesite, y, a su vez, que los espectadores puedan ver el progreso del notebook. Si se continua funcionando como hasta ahora, se podría hacer que cada usuario tuviera una instancia del notebook diferente, y asignar un *kernel* a cada uno de ellos. De esta manera, cada usuario contaría con su contexto independiente, pero ¿cómo se mustra la información en tiempo real a los espectadores? ¿Y cómo se resuelven los conflictos que se puedan generar?, por ejemplo en el caso de que diferentes usuarios hayan modificado la misma celda. Estas preguntas no tienen una respuesta sencilla, por lo que se plantea la colaboración de dos o más usuarios simultáneos haciendo uso de un modelo diferente, en el cuál a cada usuario se le deja modificar únicamente una celda en cada momento.

De esta manera, el usuario con permisos de administración podría dar los permisos a celdas en concreto, haciendo que cada usuario solo pueda modificar una celda a la vez. Para que no haya conflictos en la ejecución, se mantendrá una versión *master* del *notebook*. Esta versión será la guardada en disco, y la correcta en cada momento dado. A la hora de conocer el contexto con el que está trabajando, el usuario siempre tendrá como referencia esta versión *master*. Para que se produzcan modificaciones en ella, el usuario con permisos de administración en el *notebook* tiene que aceptar los cambios realizados por un editor en la celda editada. Cuando se acepten, se actualizará la versión *master*, y, de esta manera, se empujarán los cambios a todos los usuarios que estén editando, para que vean que el contexto ha cambiado y que les puede afectar.

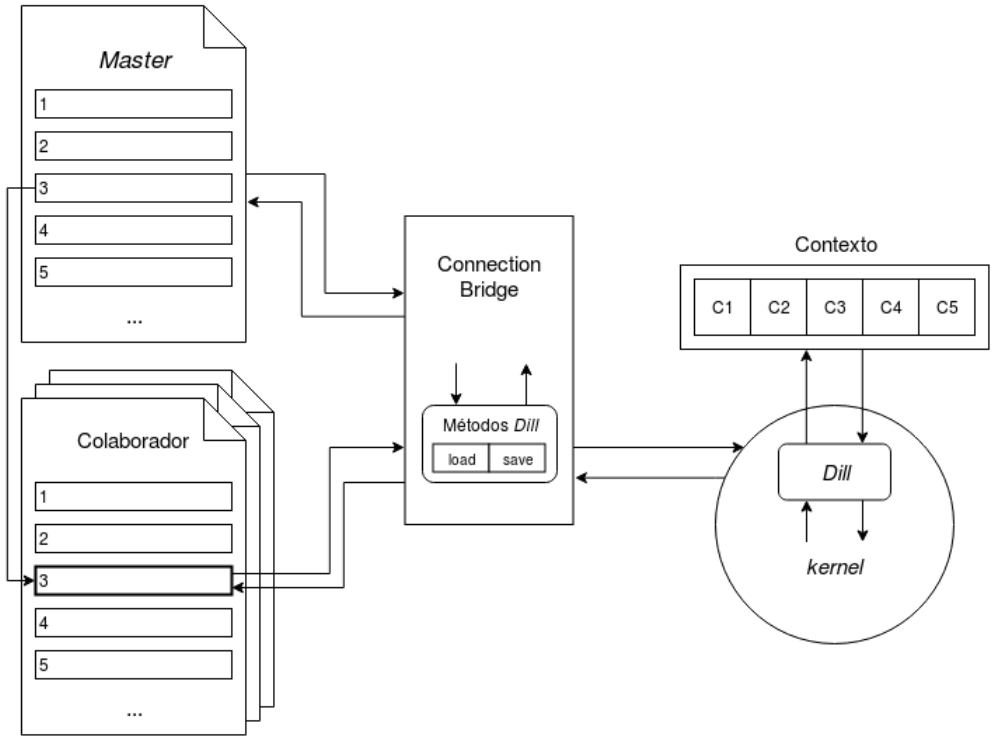

Figura 9.2: Esquema del nuevo modelo de ejecución

En la Figura 9.2 se puede ver un esquema de este nuevo modelo de edición y ejecución de nuestra plataforma. Podemos apreciar la versión máster del notebook así como los diferentes colaboradores. Tambien podemos ver que aparece un elemento llamado *connection bridge*, que será el encargado de interceptar los mensajes entre los notebooks y el *kernel* y añadir los métodos *Dill* [1] , que gestionarán los contextos, realizando su guardad y carga. El funcionamiento del sistema es el siguiente:

En primer lugar se tiene que realizar una ejecución del *notebook master* para generar los contextos. En el caso de que el *notebook* sea nuevo y esté vacío no es necesario, porque no habrá contexto disponible. Este modelo de ejecución va a seguir la lógica de que el *notebook* es lineal, es decir que las celdas inferiores siempre van a ejecutarse más tarde que la superiores. Al ejecutar el *notebook master* completo, se mandan las celdas a ejecución una a una y en orden. Cada vez que el connection bridge reciba el código de una celda, le insertará un método *dill* de guardado y lo mandará a ejecutar. De esta manera, se almacenará el contexto de cada celda en la array de contextos, que estará guardada en disco.

[1] https://pypi.org/project/dill/

Después, el administrador compartirá una celda, nombrando a un usuario editor. Cuando este usuario quiera realizar una ejecución de la celda, se mandará el código a el *connection bridge*, y éste añadirá el método de cargado *dill* de la celda anterior a la que se va a mandar a ejecutar. De esta manera, el *kernel* cargará el contexto correspondiente y la celda se ejecutará de forma consistente. El editor puede hacer pruebas, porque el contexto siempre va a ser el de la celda anterior en sus ejecuciones.

Por último, cuando un editor termine con las modificaciones en su celda, el administrador del *notebook* tiene que decidir si mantiene los cambios o los descarta. En el caso de descartar los cambios propuestos, no se tiene que realizar ninguna ejecución, porque los contextos se mantienen. Sin embargo, en el caso de aceptar estas modificaciones, se tienen que recalcular los contextos de las celdas posteriores, porque dependen de éste. En el momento en el que se acepten los cambios, se ejecutará desde la celda cambiada hasta el final del documento. En la primera celda que se ejecute, el *connection bridge* insertará el método *dill* de carga al principio y el de guardado al final, para ejecutar las celdas en base al contexto correcto. El resto de celdas se ejecutarán con el método *dill* de guardado al final. Para finalizar, se actualizarán los notebooks de los editores con un mensaje, avisando de que la versión *master* ha cambiado, para que lo tengan en cuenta.

Este sistema funciona con un solo *kernel*, por lo que el consumo de recursos es bajo. Sin embargo, en el caso de haber muchos editores y querer ejecutar sus celdas a la vez, algunos usuarios puede que tengan que esperar. Esto se debe a que el *kernel* solo puede ejecutar una celda a la vez, por lo que el *connection bridge* tiene que implementar una cola, mandando una celda a ejecutar cuando el *kernel* le haya devuelto un resultado y esté disponible. Por otro lado, se podría plantear un modelo con un *kernel* por editor, cargando su contexto al principio. De esta manera el *connection bridge* serviría para dirigir cada celda a su *kernel* asociado. Se debe valorar si interesa un sistema con poco tiempo de espera y poco consumo de recursos o si se prefiere un tiempo de respuesta menor con un coste computacional más elevado. De esto dependerá la topología elegida para el sistema mejorado.

Glosario

A | B | C | F | H | J | L | N | P | S

A

array

Estructura de datos en forma de vector unidimensional que contiene una serie de elementos del mismo tipo. 10

B

bug

Problema en un programa de computador o sistema de software que desencadena un resultado indeseado. 3

C

callback

Cualquier código ejecutable que se pasa como argumento a otro código, que lo llamará de vuelta (*call back*) y ejecutará el argumento en un momento dado. 20

cookie

Pequeña información enviada por un sitio web y almacenada en el navegador del usuario, de manera que el sitio web puede consultar la actividad previa del navegador. 14

F

flag

Uno o más bits que se utilizan para almacenar un valor binario o código que tiene asignado un significado. 26

H

handler

Rutina que opera de forma asíncrona y maneja los eventos recibidos en un programa. 10

hash

Función que tiene como entrada un conjunto de elementos, que suelen ser cadenas, y los convierte en un rango de salida finito, normalmente cadenas de longitud fija. 15

HTTP

acrónimo de *Hypertext Transfer Protocol* es el protocolo de comunicación que permite las transferencias de información en la *World Wide Web*. 10

J

jquery

biblioteca multiplataforma de *JavaScript* que permite simplificar la manera de interactuar con los documentos *HTML*. 26

JSON

Acrónimo de *JavaScript Object Notation*, es un formato de texto ligero para el intercambio de datos. Más información en `https://es.wikipedia.org/wiki/JSON`. 1

JWT

Estándar abierto basado en *JSON* propuesto para la creación de *tokens* de acceso que permiten la propagación de identidad y privilegios o claims en inglés. 19

L

link

Elemento de un documento electrónico que permite acceder automáticamente a otro documento o a otra parte del mismo. 38

N

notebook

Documentos que contienen tanto código como elementos de *texto rico*, como figuras o tablas. Los notebooks son documentos tanto legibles como ejecutables.. 1

P

polling

operación de consulta constante para crear una actividad sincrónica sin el uso de interrupciones. 37

S

scroll

Movimiento en 2D de los contenidos que conforman el escenario de un videojuego o la ventana que se muestra en una aplicación informática. 25

Bibliografía

[1] Erik Sandewall. Programming in an Interactive Environment: the Lisp Experience. *ACM Computing Surveys (CSUR)* 1978.

[2] F. Pérez, B.E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering* 21-29, 2007

[3] T. Kluyver et al. Jupyter Notebooks – a publishing format for reproducible computational workflows *20th International Conference on Electronic Publishing*, Enero 2016

[4] Anind K. Dey, Gregory D. Abowd. Towards a Better Understanding of Context and Context-Awareness. *HUC '99 Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing* 304-307, 1999

[5] Brian Hayes. Thoughts on Mathematica. *Pixel* 28-35, Enero-Febrero 1990

[6] Wolfram S. *Mathematica*. Cambridge University Press; 1996.

[7] *The wire protocol*, el protocolo de mensajería cliente-*kernel* de *Jupyter*. Consultado el 19/06/2018 en http://jupyter-client.readthedocs.io/en/stable/messaging.html