Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Facultad de Informática

TRABAJO FIN DE GRADO

Desarrollo de una solución SaaS para una Enterprise Store

Autor: Camilo Pereira León

Director: Miguel Jimenez Gañán

MADRID, JUNIO DE 2014

Resumen

En el marco del proyecto europeo FI-WARE, en el CoNWet Lab (laboratorio de la ETSI Informáticos de la UPM) se ha implementado la plataforma *Web Wstore* que es una implementación de referencia del *Store Generic Enabler* perteneciente a dicho proyecto. El objetivo de FI-WARE es crear la plataforma núcleo del Internet del Futuro (IoF) con la intención de incrementar la competitividad global europea en el mundo de las TI. El proyecto introduce una infraestructura innovadora para la creación y distribución de servicios digitales en internet.

WStore ofrece a los proveedores de servicios la plataforma donde publicar sus ofertas y desde la cual los clientes podrán acceder ellas. Estos proveedores ofrecen servicios *Web*, aplicaciones, *widgets* y *data sets* del mismo modo que *Google* ofrece aplicaciones en la tienda online *Google Play* o *Apple* en el *App Store*. *WStore* está implementada actualmente como una plataforma *Web*, por lo que una organización que desee ofrecer el servicio de la store necesita instalar el *software* en un servidor propio y disponer de un dominio para ofrecer sus productos.

El objetivo de este trabajo es migrar *WStore* a un entorno de computación en la nube de manera que con una única instancia se ofrezca el servicio a las organizaciones que deseen disponer de su propia plataforma, de la cual tendrán total control como si se encontrase en su propia infraestructura. Para esto se implementa una versión de *WStore* que será desplegada en una infraestructura *cloud* y ofrecida como *Software as a Service*. La implementación incluye una serie de módulos de código que se podrán añadir opcionalmente en el proceso de instalación si se desea que la instancia instalada sea *Multitenant*. Además, en este trabajo se estudian y prueban las herramientas que ofrece *MongoDB* para desplegar la plataforma *Wstore Multitenant* en una infraestructura *cloud*.

Estas herramientas son *replica sets* y *sharding* que permiten desplegar una base de datos escalable y de alta disponibilidad.

Abstract

In the context of the European project FI-WARE, the CoNWeT Lab (IT Lab from ETSIINF UPM university) has been implemented the web platform WStore. WStore is a reference implementation of the Generic Enabler Store from FI-WARE project. The FI-WARE goal is to create the core platform of the Future Internet (IoF) with the intention of enhancing Europe's global competitiveness in IT technologies. FI-WARE introduces an innovative infrastructure for the creation and distribution of digital services over the Internet.

WStore offers to service providers a platform to publicate offerings and where customers can access them. The providers offer web services, applications, widgets and data sets in the same way that *Google* offers online applications on *Google Pla*y or *Apple* on *App Store* plataforms. WStore is currently implemented as a web platform, so if an organization wants to offer the store service, it need to install the software on it’s own serves and have a domain to offer their products.

The objective of this paper is to migrate WStore to a cloud computing environment where a single instance of the WStore is offered as a web service to organizations who want their own store. Customers (tenants) of the WStore web service will have total control over the software and WStore administration. The implementation includes several code modules that can be optionally added in the installation process to build a Multitenant instance. In addition, this paper review the tools that MongoDB provide for scalability and high

availability (*replica sets* and *sharding*) with the purpose of deploying multi-tenant WStore on a cloud infrastructure.

Índice

1 Introducción....5
1.1 Ámbito del trabajo....7
1.2 Objetivos....8
2 Estado del Arte....9
2.1 Cloud Computing....9
2.1.1 Modelos de servicio en la nube....9
2.1.1.1 Software as a Service (SaaS)....10
2.1.1.2 Platform as a Service (PaaS)....11
2.2. Tecnologías utilizadas....13
2.2.1 Python....13
2.2.2 Django....14
2.2.3 MongoDB....17
2.2.3.1 Replica sets....17
2.2.3.2 Sharding....18
2.3 FI-WARE....22
2.4 WStore....23
2.5 Proyectos similares....27
2.5.1 Django-tenant-schemas [2]....28
2.5.2 Django-db-multitenant [3]....28
2.5.3 Django-simple-multitenant [4]....29
3 Implementación de la funcionalidad de Multitenancy....30
3.1 Almacenamiento y aislamiento de los datos....30
3.2 Información de tenants....35
3.3 API para multitenancy....37
3.4 Django View....38
3.5 Django Middleware....41

3.6 Señales....44
3.7 Problemas 46
3.8 Manager....48
3.9 Interfaz web....50
3.10 Organización del código e instalación manual....51
4.0 Despliegue de una Base de Datos Distribuida....53
4.1 Escalado median te Sharding automático....54
4.1.1 Descripción del servidor de configuración....55
4.1.2 Descripción del servidor de consulta....55
4.1.3 Servidores de almacenamiento....56
4.1.4 Elección de la clave....57
4.2 Alta disponibilidad y tolerancia a fallos mediante Replica Set....60
5.0 Conclusiones....65
5.1 Conclusiones personales....66
5.2 Lineas futuras....66
6 Bibliografía y referencias 67

Índice de ilustraciones

Ilustración 1: Pirámide cloud.....11
Ilustración 2: Arquitectura Django.....16
Ilustración 3: interfaz Web del login de WStore.....27
Ilustración 4: Interfaz web WStore.....28
Ilustración 5: Bases de datos separadas.....32
Ilustración 6: Misma base de datos, esquemas diferentes.....33
Ilustración 7: Misma base de datos, mismo esquema.....34
Ilustración 8: Comparativa entre BBDD compartidas o separadas.....35
Ilustración 9: Ejemplo del contenido de Site.....40
Ilustración 10: middeware.....42
Ilustración 11: Interfaz web de solicitud de tenats.....52
Ilustración 12: Arbol de ficheros del código.....53
Ilustración 13: arquitectura para sharding.....55
Ilustración 14: Clave para Sharding.....59
Ilustración 15: Conexiones entre servidores del replica set.....62
Ilustración 16: Proceso de elección en el replica set.....64

1 Introducción

Hoy en día el concepto de *Cloud Computing* es tendencia dentro del mundo de la informática. En la Web e incluso los medios de comunicación encontramos numerosas definiciones del concepto de computación en la nube y sus ventajas. *Cloud Computing* es llevar el *software* al plano de los servicios tradicionales como pueden ser el agua o la electricidad. Si al usar una lavadora no nos preguntamos de que embalse proviene el agua ni de que central eléctrica se alimenta, ¿Por qué no hacer uso del *software* sin preguntarnos en que servidor está ejecutándose ni donde se encuentra físicamente dicho servidor?, ¿Por qué no pagar sólo por el uso que hacemos del *software* o de los recursos de computación?. El desarrollo de Internet hasta el punto que conocemos hoy nos abre un mundo de posibilidades que en décadas pasadas implicaría disponer físicamente de un supercomputador. La computación en la nube nos permite disponer de capacidad de cálculo casi ilimitada en cualquier parte donde se disponga de conexión a Internet, de ahí proviene el nombre de este paradigma, pues simboliza el poder disponer del *software*, de las plataformas o de la infraestructura como un servicio presente en todas partes, como las nubes.
La capa principal y de más alto nivel dentro del *Cloud Computing* la forma el modelo conocido como *Software as a Service (SaaS)*, esta es la que establece que la forma de distribuir y comerciar el *software* ya no es proporcionar los medios para que el *software* sea instalado en la infraestructuras de los consumidores, sino que este se ofrece como servicio desde infraestructuras pertenecientes al proveedor del *software*.

Según los datos de la empresa consultora y de investigación tecnológica Gartner, para el año 2015 el 85% del software será ofrecido bajo el modelo de *SaaS*. Algunas de las ventajas del uso de este modelo para las empresas son:

- Menor inversión ya que se puede hacer uso del *software* sin tener que invertir en infraestructura y *software*.
- Reducción de costes, al pagar por solo aquello que necesites y ahorrar en costes de mantenimiento al no disponer infraestructura propia.
- Actualizaciones, mantenimiento y nuevas funcionalidades de forma automática e inmediata, gestionadas por el proveedor del servicio.

Estas ventajas promueven que cada vez más las empresas que requieren de capacidad de cálculo y utilización de *software* suplan estas necesidades contratando los servicios que otras empresas que disponen de la infraestructura necesaria para ofrecer *software* como servicio. Al mismo tiempo cada vez más las empresas que desarrollan *software* cambian su modelo de negocio hacia el *SaaS* impulsadas por la demanda de dichos servicios. Actualmente las principales empresas tecnológicas ofrecen servicios *cloud* como pueden ser *Google* con *Google Apps*, *Amazon* con *Amazon WS* o *Microsoft* con *Windows Azure*. Ofrecer *software* como servicio no está limitado a grandes compañías y en este Trabajo de Fin de Grado se abordará la implementación de una solución *SaaS* para ofrecer como servicio la *WStore*, una plataforma web implementada en el CoNWeT Lab, laboratorio de la ETSI Informáticos de la UPM.
WStore es la implementación *open source* del *Store*, un *Generic Enabler* en el marco del proyecto europeo FI-WARE. FI-WARE es un proyecto de gran escala en el que participan las principales empresas y organizaciones académicas del ámbito tecnológico europeo. La implementación de la *store* en la que se basa este trabajo ofrece un servicio de tienda online como puede ser la *App Store* de *Apple* o *Google Play* si bien esto es sólo una parte de su funcionalidad. *WStore* permite monetizar servicios, aplicaciones y *widgets* con diferentes

formas de pago que se adaptan al servicio o *software* comercializado. *WStore* está implementada actualmente como una plataforma *Web*, por lo que una organización que

desee ofrecer el servicio de la *store* necesita instalar el *software* en un servidor propio y disponer de un dominio para ofrecer sus productos.

1.1 Ámbito del trabajo

En este trabajo se implementará una serie de módulos de *software* que podrán añadirse al *WStore* para dotar a la plataforma de la funcionalidad de *Multitenan*t, se pretende que este código pueda usarse en un entorno de producción real y por tanto durante la implementación se ha tenido especial cuidado de respetar las mejores prácticas de diseño y programación en el marco del proyecto europeo del que forma parte. El código ha sido probado minuciosamente en el intento de eliminar el mayor número de errores posible y debidamente comentado para que otros desarrolladores puedan entender los detalles de la implementación y beneficiarse de esta.
Por otra parte, este trabajo también contempla el despliegue de una una base de datos distribuida para el almacenamiento de los datos de *WStore Multitenant*. Para esto se ha realizado la configuración de la base de datos en un entorno de prueba basado en máquinas virtuales. Queda fuera del ámbito de este trabajo un despliegue de la plataforma en un *cluster* real para producción, con toda la infraestructura extra que esto implica.

1.2 Objetivos

El objetivo de este trabajo es migrar *WStore* a un entorno de *cloud computing* de manera que con una única instancia se ofrezca el servicio a las organizaciones que deseen disponer de su propia *store*, de la cual tendrán total control como si se encontrase en su propia infraestructura. Además, los datos de cada organización que haga uso de la *WStore* como *SaaS* no deben ser accesibles para el resto de organizaciones. El objetivo es implementar los módulos necesarios para ofrecer la *WStore* como servicio en la nube. Que una instancia de la *WStore* se ofrezca servicio *cloud* implica que esta debe compartir los recursos en el

entorno de ejecución de forma transparente a las diferentes organizaciones que hagan uso de este servicio. Para cumplir estos requisitos es necesario que la instancia que se instale para ofrecerse como *SaaS* disponga de las funcionalidades de *Multitenancy*, Escalabilidad y Alta disponibilidad que son los pilares para ofrecer *software* como servicio. Posteriormente se explican con más detalles estos conceptos.

La lista de objetivos está enfocada en los temas de seguridad y tecnologías cloud como se observa a continuación:

- Ofrecer la *WStore* actual como servicio *cloud*
- Dotar de Funcionalidad *Multitenant* a *WStore*.
- Hacer de *WStore* un servicio escalable.
- Hacer de *WStore* un servicio de alta disponibilidad.

2 Estado del Arte

2.1 Cloud Computing

El *National Institute of Standards Technology* define *Cloud Computing* como el modelo que permite el uso bajo demanda y a través de la red a un conjunto compartido de recursos computacionales configurables que pueden ser aprovisionado y liberado con el mínimo esfuerzo en administración y la menor interacción posible por parte del proveedor. [1]

Cloud computing es el paradigma de computación donde el *software* es ofrecido de forma ubicua y como servicio. Estos servicios son ofrecidos a través de internet permitiendo a los clientes disponer de capacidad de cómputo, almacenamiento y de todo tipo de *software*. Dentro de los tipos de *Cloud* existente, merece la pena diferenciar las *Cloud* públicas, ó *Public Clouds*, que ofrecen sus recursos como servicios al público en general y las *Cloud* privadas, ó *Private Clouds*, diseñados de forma exclusiva para proveer servicios a una organización, evitando los problemas inherentes al almacenamiento y manejo de datos sensibles fuera del control de la misma.

2.1.1 Modelos de servicio en la nube

Los servicios que ofrece una infraestructura cloud se pueden clasificar como Software as a Service (SaaS), Platform as a Service (PaaS) y Infrastructure as a Service (IaaS).

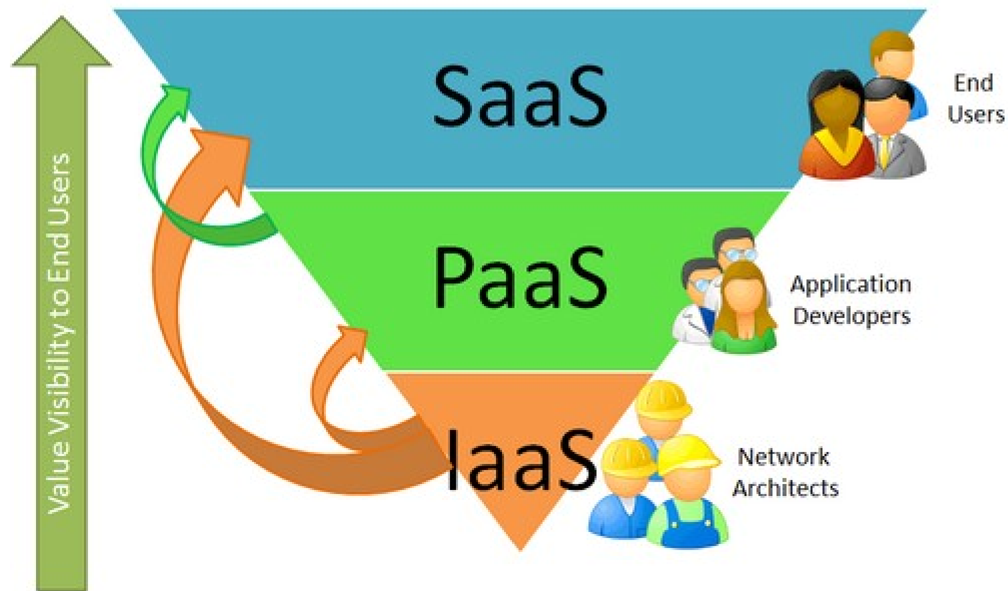


Ilustración 1: pirámide cloud

2.1.1.1 Software as a Service (SaaS)

Este es el modelo más utilizado en los entornos *cloud* y ofrece *software* bajo demanda que ejecuta en la infraestructura del proveedor de servicios. Debido a esto, el *software* es administrado por el proveedor y el cliente no puede realizar modificaciones sobre el mismo. Los beneficios que ofrece este tipo de servicios son amplios. Por un lado, el proveedor (y no el consumidor) es ahora el encargado de las actualizaciones y la gestión del servicio (escalabilidad, disponibilidad, mantenimiento, etc.). Por otro lado, el consumidor puede

reducir gastos debido al modelo de pago por uso que suelen ofrecer este tipo de servicios. Entre los servicios con mayor inversión dentro de este modelo encontramos *Suites*

ofimática como *Google Docs*, *Office 365*, plataformas de Inteligencia Empresarial (BPI), gestión de datos empresariales o CRM (*SalesForce*, etc.), plataformas de almacenamiento en la nube como *Drobox o Google Drive*, etc.

2.1.1.2 Platform as a Service (PaaS)

Este tipo de servicios permite a los clientes disponer de una plataforma computacional bajo demanda sin que estos la disponga de la infraestructura necesaria. Este tipo de servicios también reduce la complejidad a la hora de desplegar y mantener aplicaciones ya que el servicio gestiona automáticamente la escalabilidad usando más recursos si fuera necesario. Los clientes pueden centrar su esfuerzo en desarrollar sus propias aplicaciones que se ejecutarán en la nube. Todo esto permite la reducción de costes de mantenimiento y gestión. Ejemplos de *PaaS* son *Google App Engine* y *Microsoft Azure* que permiten a los clientes desarrollar aplicaciones sin disponer de la infraestructura física.

2.1.1.3 Insfrasctructure as a Service (IaaS)

Este servicio ofrece más control que con *PaaS*, aunque a cambio de eso el cliente debe encargarse de la gestión de infraestructura. *Amazon Web Service (AWS)* ofrece este tipo de servicios como EC2 que permite utilizar maquinas virtuales en la nube o S3 para almacenamiento. El consumidor elige qué tipo de instancias desea desplegar, pudiendo instanciar máquinas virtuales *Linux* o *Windows*, así como elegir la configuración de *software* adecuada (capacidad de memoria o procesador de cada una de las máquinas virtuales). Todo el *hardware* es controlado virtualmente por el cliente. La forma de pago de

este tipo de servicios está orientada al pago por uso, dependiendo de la cantidad de máquinas virtuales que se ejecutan, las características de *hardware* seleccionadas, la carga de trabajo de la *CPU*, tráfico de red, etc. Este modelo tiene grandes ventajas en coste frente a una infraestructura propia, tanto al requerir pocos recursos, puesto que se paga por uso,

como al necesitar grandes recursos de computación, puesto que la gestión que hacen los proveedores suele ser más económica que realizarla dentro de la propia empresa.

La arquitectura para la computación en la nube está representada por un modelo de capas:

- Capa *Hardware* donde se administran los recursos físicos del *cloud*
- Capa de infraestructura con un conjunto de recursos basados en tecnologías de virtualización. La virtualización es la tecnología que abstrae a través de *software* los recursos de un computador, permitiendo dividir estos en varios entornos de ejecución, como si de sistemas operativos en máquinas diferentes se tratase.
- Capa de plataforma, que consiste en el sistema operativo y los *frameworks* de aplicación.
- Capa de Aplicación, con las aplicaciones finales que ofrece el *cloud*. Esta es la capa de más alto nivel y en la que se centra este Trabajo de Fin de Grado.

WStore será ofrecida como *SaaS*, y se le quiere dotar de las siguientes características:

- *Multitenancy* es el concepto fundamental de todo *SaaS* y consiste en la capacidad para dar servicio de forma independiente a los clientes que consuman el servicio desde una única instalación del *software*. Esto implica compartir los recursos de todos los clientes (*tenants*) en el servidor pero dando la sensación a estos de que

disponen del servicio en exclusiva y con total control como si estuviera instalado en su propia infraestructura.

- La escalabilidad es la habilidad que tiene un sistema de soportar el crecimiento de la carga de trabajo sin perder calidad en los servicios que este ofrece. Esto es logra haciendo uso de nuevos recursos que pueden ser añadir recursos a los servidores (cantidad de memoria *RAM*, procesamiento, almacenamiento o red) o por otra parte añadir nuevos servidores con las mismas características.
- La alta disponibilidad es una de las características más importantes con las que se debe dotar una infraestructura *cloud* ya que al estar ofreciendo un servicio, este debe estar garantizado a los clientes en todo momento, evitando que estos se vean afectados por un corte del servicio.

2.2. Tecnologías utilizadas

A continuación se describen las tecnologías estudiadas y posteriormente aplicadas en el marco de este trabajo, tanto para ofrecer el servicio *Web*, como para el despliegue de una base de datos distribuida.

2.2.1 Python

Python es un lenguaje de programación interpretado multiparadigma de propósito general multiplataforma y con tipado dinámico. Este lenguaje ofrece una serie de ventajas de cara al desarrollo de aplicaciones, algunas de estas ventajas son:

- **Simple y potente**: La programación en *python* es eficiente y clara, siendo un lenguaje sencillo y compacto que produce código más facil de comprender. Todo

esto sin dejar de ser un lengua con una potencia increíble gracias al gran número de librerías y herramientas existentes.

- **Multiplataforma:** Permite el desarrollo de *software* multiplataforma que puede ser ejecutado en la mayoría de los sistemas operativos.
- **Comunidad**: *Python* está soportado por una gran comunidad de usuarios por lo que existe infinidad de documentación y foros de ayuda para aprender el lenguaje y resolver problemas en el proceso de implementación.
- **Framework para desarrollo web:** Es posible programar aplicaciones *Web* con python gracias al *framework Django*, que ofrece todas las herramientas para construir plataformas *Web* robustas y complejas.

2.2.2 Django

Django es un framework gratuito y abierto para desarrollo *Web*. *Django* está escrito en *Python* y se basa en el patrón de Modelo-Vista-Controlador, o MVC. A continuación se describen los elementos del patrón MVC para django:

- **Modelo**: modelos de datos definidos en clases *Python*. El framework contiene un *mapper* que permite convertir estas clases en tablas dentro de la base de datos de forma automática. En este trabajo se utiliza la versión de *Django Non-Rel*, que utiliza *MongoDB* como sistema de base de datos. Esto implica que se realiza un mapeo de los modelos de datos hacia colecciones de *MongoDB*. Posteriormente se explica el funcionamiento de *MongoDB* y sus peculiaridades como sistema de bases de datos no relacionales.

- **Vista**: un sistema de plantillas utilizado para dar respuesta a cada una de las peticiones recibidas. *Django* contiene un fichero *views.py* donde se definen funciones y clases encargadas de procesar las peticiones *HTTP* y ejecutar código en consecuencia.

- **Controlador**: Procesa las peticiones al servidor, relacionando *URL* con funciones *Python* que realizan determinadas acciones para dar respuesta a estas peticiones. El diseño de estas *URLs* es ampliamente flexible al estar basado en expresiones regulares.

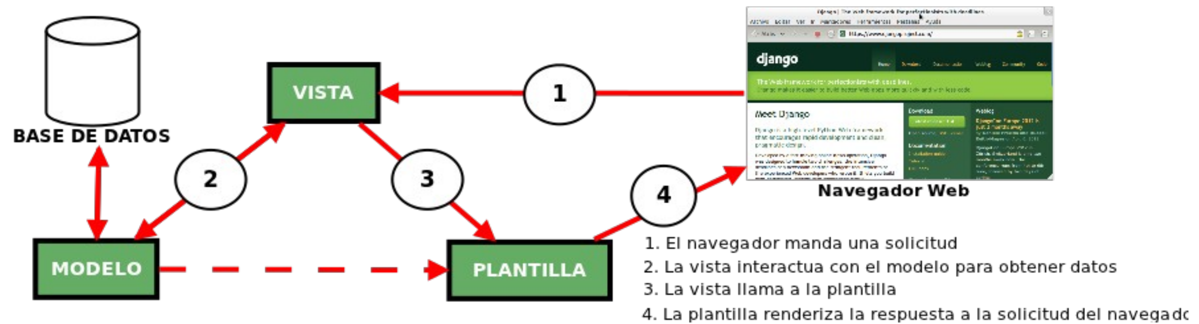


Ilustración 2: Arquitectura Django

Django ofrece una serie de características por las cuales es una de las mejores opciones para el desarrollo de plataformas *Web*. Algunas de estas características son:

- Simplifica el desarrollo al abstraer la gestión de la base de datos, permitiendo el acceso a los datos desde el código en forma de objetos *Python* sin la realización de consultas explícitas a la base de datos por parte del programador.
- Separa el código de la lógica de la aplicación de la parte de visualización, la cual abstrae con un motor de plantillas para *HTML*, que permite visualizar el contenido de la *Web* con los datos generados por la lógica de la aplicación.

- Permite añadir nueva funcionalidad por medio de *Apps* que pueden ser reutilizadas en diferentes proyectos *Web*. Por ejemplo, la funcionalidad de *Multitenancy* implementada en este trabajo se añade través del *App Multitenancy*.

- El núcleo de *Django* es flexible y permite modificar el comportamiento de las aplicaciones *Web* por medio de señales y *middleware*. Las señales ejecutan funciones facilitadas por el usuario cuando se activan dichas señales según ciertas condiciones como puede ser el acceso a la base de datos. Así mismo, Los *middlewares* sirven para interceptar peticiones al servidor y ejecutar código consecuentemente, e incluso modificar los datos de las peticiones interceptadas antes de ser procesadas por el servidor.

2.2.3 MongoDB

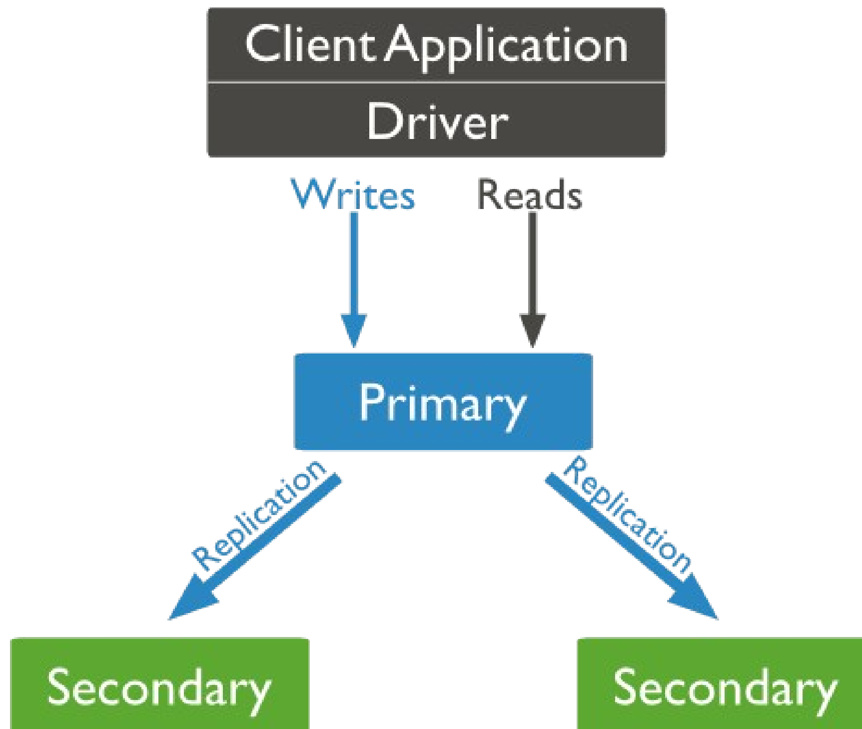
MongoDB es un sistema de base de datos *NoSQL* orientado a documentos. Esto significa que los datos no se organizan en tablas como en las bases de datos *SQL*, sino que se almacenan en ficheros llamados colecciones. Las colecciones contienen documentos, que son estructuras en formato *JSON* donde se almacenan los datos sin ningún esquema o formato predefinido. *MongoDB* dispone de una serie de características beneficiosas a la hora de desplegar una infraestructura cloud como son *Replica sets* y *sharding*. A continuación se describen ambos.

2.2.3.1 Replica sets

Replica sets son el conjunto de servidores donde se almacenan los datos en *MongoDB*. Estos servidores contienen exactamente los mismos datos y dotan a *MongoDB* de redundancia y alta disponibilidad. Con múltiples copias de los datos en diferentes

servidores la replicación protege a la base de datos de la pérdida de datos en un servidor. La replicación también permite recuperarse de fallos en el *hardware* e interrupciones del servicio.

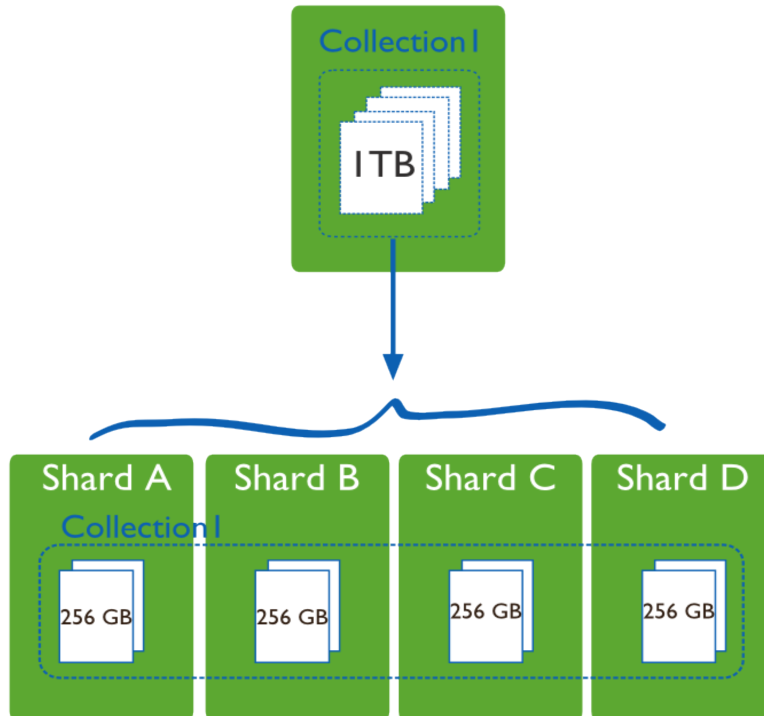
Con copias adicionales de los datos también puede mejorarse la capacidad de la base de datos para realizar operaciones ya que se dispone de más servidores a los que realizar peticiones de lectura y escritura dentro de la base de datos.



2.2.3.2 Sharding

Sharding es el método mediante el cual *MongoDB* almacena los datos en diferentes servidores. La totalidad de los datos se divide en porciones más pequeñas del mismo tamaño, respecto a una clave para ser distribuida en los diferentes servidores (cada servidor de los que componen la red de *sharding* es en realidad un *replica set*). Por ejemplo, si se

usa el identificador de usuario como clave para el *sharding* los datos de cada usuario quedarían separados en particiones, una por cada usuario existente en la base de datos.
Los sistemas de bases de datos como *MongoDB*, que gestionan grandes cantidades de datos no son posibles de desplegar en un solo servidor. Muchas peticiones a la base de datos pueden agotar la capacidad de la *CPU* de un solo servidor. Grandes volúmenes de datos pueden sobrepasar la capacidad de almacenamiento de una sola máquina. Finalmente, trabajar con conjuntos grandes de datos requieren de una cantidad de *RAM* y capacidad de I/O superior a lo que un único servidor puede ofrecer.
Estos problemas de escalabilidad pueden resolverse mediante el escalado vertical, que consiste en añadir más recursos hardware a los servidores, o por el contrario, resolverlos mediante el escalado horizontal, que se trata de añadir más servidores con semejantes características, formando así una red de nodos conectados, que individualmente no son excesivamente potentes pero que en conjunto forman una gran red de procesamiento de datos. El escalado horizontal es el más indicado puesto que no tiene límite a la hora de escalar, y cuando se necesite más capacidad de cómputo basta con añadir más nodos a la red. *MongoDB* permite el escalado horizontal mediante el *sharding*, donde cada nodo contiene una base de datos independiente donde se almacena una porción de los datos del sistema.



Resumiendo, *MongoDB* ofrece una serie de características diferentes respecto a las bases de datos tradicionales *SQL*:

- Modelo de datos basado en documentos que son estructuras con formato de diccionario (pares clave-valor), que permiten el almacenamiento de estructuras complejas de datos como pueden ser arrays y otros documentos.
- *MongoDB* usa un lenguaje dinámico para consultas sobre documentos del mismo modo que SQL para bases de datos relacionales.

- *MongoDB* no requiere una definición estática de los datos, como puede ser definir las columnas de una tabla en una base de datos *SQL*. Esto significa que en una colección los documentos pueden tener campos diferentes a diferencia de las bases de datos relacionales, donde cada entrada en una tabla tiene los mismos campos (columnas).

Estas características hacen de *MongoDB* la mejor alternativa para la capa de persistencia en una plataforma *Web* que maneje grandes cantidades de datos con relaciones complejas. Almacenar datos relacionados en un mismo documento permite que las consultas que solicitan estos datos puedan ser más rápidas que en una base de datos relacional, donde los datos relacionados se encuentran en tablas diferentes, lo que requiere operaciones costosas como los *joins* para obtener estos datos. También se benefician las aplicaciones que manejan datos desestructurados, los cuales no pueden almacenados en tablas con campos fijos. Disponer de estos esquemas dinámicos también permite que se desarrollen plataformas *Web* de forma ágil, ya que a diferencia de las bases de datos relacionales, en *MongoDB* no es necesario especificar las propiedades que tendrán los datos (columnas de las tablas) y por tanto es posible añadir nuevas propiedades al modelo de datos sin tener que migrar la totalidad de la base de datos a este nuevo esquema.
Las aplicaciones web con *MongoDB* se benefician también de las facilidades de escalado que este ofrece, permitiendo el escalado lineal añadiendo más nodos al *cluster*. Gracias al autosharding el escalado se realiza sin interrupciones en el servicio.
En la actualidad existen clusters con *MongoDB* que han escalado desde cientos de nodos hasta el orden de miles. También existen bases de datos con *MongoDB* que almacenan grandes cantidades de datos del orden de los petabytes en miles de millones de documentos.

2.3 FI-WARE

Los avances en las tecnologías de la información, como pueden ser el surgimiento del cloud computing, el considerable aumento del ancho de banda disponible para el acceso a internet mediante fibra óptica y las redes inalámbrica 4G, conducen a una nueva era conocida como Internet del futuro, con nuevas oportunidades de negocio donde todo se realiza a través de Internet, donde cada vez hay más dispositivos conectados que generan y procesan cantidades de datos inimaginables.
El objetivo de FI-WARE es crear la plataforma núcleo de este Internet del Futuro (IoF) con la intención de incrementar la competitividad global europea en el mundo de las TI. Esto pretende conseguirse mediante la introducción de una innovadora infraestructura para la creación y distribución de servicios digitales versátiles, con un coste eficiente y ofreciendo una alta calidad de servicio y garantías de seguridad. Así se estimulará la creación de un ecosistema sostenible donde los proveedores de servicios desarrollarán nuevas aplicaciones que satisfarán las necesidades de áreas de negocio emergentes o ya establecidas. FI-WARE será así un sistema abierto basado en unos elementos denominados *Generic Enablers* (GE) y que ofrecen funciones reusables y compartidas, útiles en diversos sectores, en contraste con los Specific Enablers, que sólo son útiles únicamente en áreas concretas. Los *Generic Enablers* que forman la plataforma núcleo de FI-WARE se encuentran dentro de alguna de las siguientes categorías:

- *Cloud Hosting*: es la capa fundamental que ofrece capacidades de computación, almacenamiento y red, sobre la que los servicios serán desplegados y gestionados.
- *Data/Context Management*: facilidades para acceder, procesar y analizar grandes volúmenes de información transformándolos en información útil disponible para las aplicaciones.

- *Internet of Things (IoT) Services Enablement*: puente entre los servicios y los dispositivos que compondrán el *Internet of Things*.

- *Interface to Network and Devices (I2ND)*: interfaces abiertas a redes y dispositivos para satisfacer las necesidades de comunicación de los servicios distribuidos a través de la plataforma.
- *Security*: mecanismos que aseguran que la entrega y el uso de los servicios es confiable y cumple los requisitos de seguridad y privacidad.
- *Applications/Services Ecosystem and Delivery Framework*: infraestructura para crear, publicar, gestionar y consumir servicios a lo largo de su ciclo de vida teniendo en cuenta los aspectos técnicos y de negocio. *WStore* es una implementación del *Store Generic Enabler*, perteneciente a esta categoría.

2.4 WStore

Como se menciona al final del apartado anterior, *WStore* es la implementación de un *Generic Enabler* dentro de la categoría *Applications/Services Ecosystem and Delivery Framework*. Se trata de una plataforma *Web* donde los proveedores de servicios publican sus ofertas y los clientes acceden a ellas. Los usuarios pueden registrarse en esta plataforma como proveedores o consumidores y forman parte de organizaciones. Las organizaciones son la forma en la que se organizan los usuarios dentro de la *WStore* y son éstas las que publican ofertas. Otras organizaciones pueden adquirir estas ofertas y sus miembros tendrán

acceso a los servicios adquiridos. Por otra parte, existen una serie de usuarios con permisos especiales, responsables de la administración de la plataforma.
Para que un usuario de *WStore* pueda ofrecer servicios en la plataforma debe solicitar el rol de proveedor y el administrador deberá aprobar esta petición. Los usuarios con el rol de proveedor pueden publicar ofertas, que incluyen información relevante como el modelo de pago, condiciones legales, acuerdo de nivel de servicio, etc. Las ofertas publicadas por provider pueden incluir recursos descargables, por ejemplo aplicaciones.

Un administrador puede registrar monedas soportadas, unidades de precio, registrar *WStore* en un *Marketplace*, etc. Un *Marketplace* es un instrumento para facilitar el comercio entre productores y consumidores. La principal funcionalidad de éstos es ofrecer una interfaz uniforme para descubrir ofertas de aplicaciones y servicios publicados en diferentes plataformas. En el *Marketplace* se pueden encontrar ofertas de diferentes instancias de *Wstore* permitiendo al cliente poder comparar ofertas y determinar la que mejor encaja con sus requerimientos.

Los datos con los que trabaja *WStore* son los siguientes:

- **Offering**: contiene información de las ofertas de la *Store*. Entre esta información cabe destacar el nombre de la oferta, la versión o la organización que la ofrece. Se almacenará el estado además de cierta información adicional.
- **Resource**: contiene información de los distintos recursos registrados para ser vinculados a alguna oferta. A parte del nombre y la versión, también se almacenará el tipo o la dirección de acceso.
- **Usuario**: contiene la información del usuario como puede ser e el nombre del usuario, las organizaciones a las que pertenece, sus roles y la dirección de facturación.

- **Purchase**: contiene información de las distintas compras realizadas. Entre otros datos se almacenará la fecha de la compra o el estado del pago.
- **MarketPlace**: contiene información de los *MarketPlaces* en los que la Store ha sido registrada con la intención de que los proveedores puedan decidir en qué *MarketPlace* publicar su oferta. Para cada *MarketPlace* se almacena su host y su puerto.
- **Repository**: contiene información de los repositorios en los que la *Store* ha sido registrada con la intención de que los proveedores puedan decidir en qué repositorio almacenar los *USDLs* que describen al servicio. De la misma manera que para los *MarketPlaces*, se almacenará el host y el puerto de cada repositorio.

- **Comment**: contiene información de los comentarios hechos por los usuarios para cada una de las ofertas. De esta forma se almacenará el comentario, la fecha del mismo y la puntuación dada a la oferta.

Ilustración 3: interfaz Web del login de WStore

FI-WARE
WStore

kmilinho
•••••
Login

Email
Password
Sign up for WStore

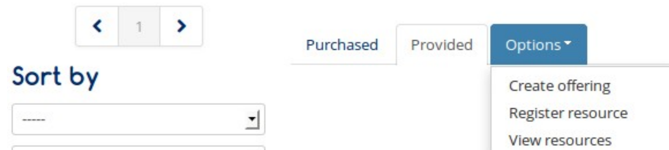


Ilustración 4: Interfaz web WStore

2.5 Proyectos similares

La necesidad de crear aplicaciones *Multitenant* en *Django* no es nueva, por lo que existen proyectos que implementan esta funcionalidad. El problema que encontramos al intentar buscar una solución *Multitenant* que encaje con los requisitos de *WStore* es que estas están implementadas para casos particulares de plataformas con una serie de características que difieren de las deseadas en *WStore*. Despúes de analizar varios proyectos se ha decidido implementar *software* específico a modo de *plugin* permitiendo dotar a *WStore* de la funcionalidad de *Multitenancy* sin necesidad de modificar la implementación original. Aunque estos proyectos no han sido adoptados para la solución si han servido como guía para el diseño de *WStore Multitenant*. Algunos de estos proyectos son:

2.5.1 Django-tenant-schemas [2]

Esta aplicación permite el desarrollo de aplicaciones en *Django* con soporte para *multitenancy* mediante el uso de esquemas en *PostgreSQL*. Con esta aplicación es posible ofrecer plataformas *Web* a diferentes clientes sobre una misma instancia de *software*, soportando datos privados o compartidos entre tenants y enrutamiento de peticiones al *tenant* correspondiente para el acceso a sus datos.
Los esquemas facilitan la creación de particiones dentro de una base de datos, estos pueden ser visto como directorios de un sistema operativo. Cada directorio (esquema) contiene sus propios ficheros (tablas de la base de datos). Esto que varias tablas tengan el mismo nombre al estar en esquemas diferentes. El uso de esquemas proporciona aislamiento entre los datos de cada *tenant* ya que estos se encuentran su propio conjunto de tablas independientes al resto.
Django-tenant-schemas identifica los tenants por su nombre de equipo. La información de cada tenant y su nombre de equipo se encuentra en una tabla global. Cuando se realiza una petición a la base de datos, el nombre de host se usa para redirigir la consulta al esquema correspondiente de la base de datos.
La ausencia de esquemas en *MongoDB* impide el uso de esta aplicación.

2.5.2 Django-db-multitenant [3]

Esta aplicación permite el desarrollo de aplicaciones en *Django* con soporte para *multitenancy* mediante el uso de bases de datos independientes para cada tenant. Esta aplicación obtiene el tenant al cual van dirigidas las peticiones y se almacena como variable global. Esto se hace mediante una función de mapeo que deberá escribir el usuario, implementando la interface *TenantMapper* de dicha aplicación. Para seleccionar la base de

datos del tenant correspondiente en cada petición la aplicación incluye un *wrapper* del *backend* de la base de datos, que es el código que controla el acceso a esta desde *Django*. Este *wraper* selecciona la base de datos mediante el comando USE DB de SQL, donde DB es la base de datos del tenant actual, almacenada globalmente como mencionamos anteriormente. La aplicación es útil en este trabajo solamente a nivel teórico ya que está pensada para ser usada sólo con *Django* y *MSQL*.

2.5.3 Django-simple-multitenant [4]

Esta aplicación implementa la funcionalidad de *Multitenancy* en una base de datos única. Las tablas que contienen datos privados tienen una columna con una *foreign key* a la tabla de tenants que indica a cual de estos pertenece dicha entrada. Para crear los modelos en *Django* con soporte para *multitenancy* basta con heredar de la clase *TenantModel* que ofrece dicha aplicación. Las consultas a la base de datos se filtran mediante un *Manager* que solo devuelve los datos del *tenant* asociado al usuario que realiza la consulta a la base de datos. En el perfil del usuario se almacena a que tenant pertenece.

3 Implementación de la funcionalidad de Multitenancy

La peculiaridad del desarrollo es que en ningún caso debe modificarse el código existente de *WStore* por lo que las funcionalidades que se añadan deben ser a modo de *plugins*, sobrescribiendo funciones y mediante herencia de clases existentes. El diseño basado en *plugins* permite que en tiempo de instalación, se pueda elegir el modo en que se instala *WStore*, que puede ser la clásica plataforma *Web* para uso particular de la organización o bien una instalación multitenant que será ofrecida como servicio por dicha organización. La funcionalidad de *multitenancy* implica que los datos de los clientes (o *tenants*) sean almacenados en de forma independiente. En este apartado se analizan las diferentes opciones disponibles para proveer esta independencia de los datos. También hay que tener en cuenta que los usuarios en ningún caso deben ser conscientes de esta compartición y sólo ellos podrán acceder a sus datos. Es necesario por tanto implementar ciertos mecanismos para la correcta gestión de los datos de cada *tenant*.

3.1 Almacenamiento y aislamiento de los datos

Los datos son el principal activo de cualquier negocio. Las compañías almacenan y procesan datos acerca de productos, clientes, empleados, proveedores, etc. Los proveedores de *Software as a Service* ofrecen servicio a estas compañías para que estas dispongan de acceso a sus datos de manera centralizada y a través de la red. Pero para que las organizaciones se beneficien de las facilidades que ofrece el *SaaS*, estas deben delegar el control de sus propios datos, confiando en que el proveedor del servicio mantenga la integridad, disponibilidad y confidencialidad de los datos.

La primera decisión a tomar y la más importante es cómo abordar el aislamiento de los datos entre tenants. Se trata de determinar cómo se van a almacenar los datos de cada tenant, de manera que los datos sólo sean accesibles por los usuarios de dicho tenant. Además, el mecanismo elegido debe ser transparente al *tenant*, cada cliente debe ver el recurso, en este caso la base de datos, como propio en su totalidad, ignorando que se comparte con otros *tenants*. Las opciones pueden ser bases de datos separadas para cada *tenant*, misma base de datos con esquemas diferentes para cada *tenant* o bien la misma base de datos y el mismo esquema para todos los *tenants*. Esta decisión es la primera que se debe tomar ya que de ella depende la forma en la que se realiza la implementación .

Bases de datos separadas

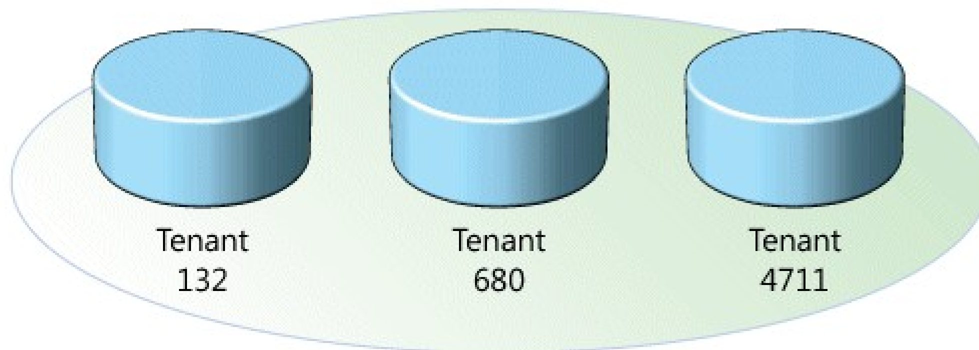


Ilustración 5: Bases de datos separadas

El uso de bases de datos separadas puede parecer el enfoque más adecuado ya que provee de aislamiento real en los datos de los *tenants*. Sin embargo, este modelo es el más ineficiente puesto que no tiene en cuenta que diferentes tenants pueden tener diferentes

volúmenes de datos, lo que supone un problema de eficiencia a la hora de asignar recursos a los tenants. Tener diferentes bases de datos implica que cada una de estas pueda crecer de forma independiente y con diferentes necesidades de *hardware*. La creación de un nuevo tenant implicaría añadir un nuevo servidor de bases de datos aún cuando el resto de servidores pueden estar infrautilizados.

Misma base de datos, esquemas diferentes

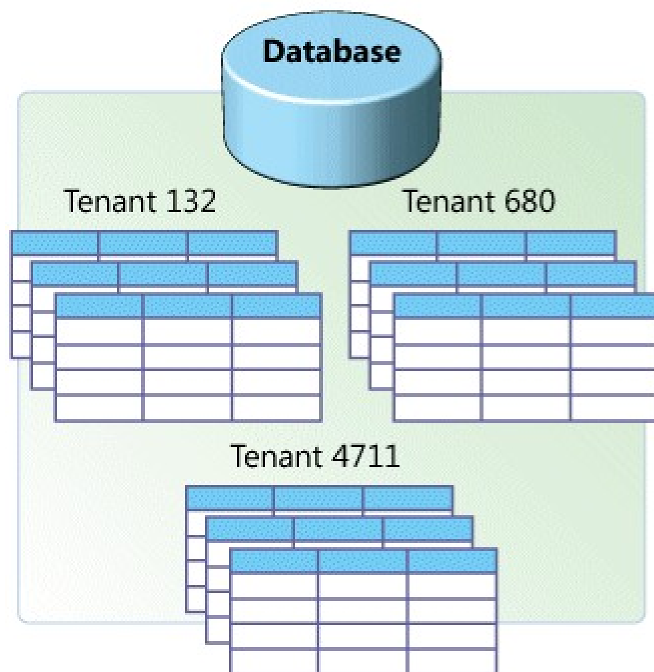


Ilustración 6: Misma base de datos, esquemas diferentes

Esta alternativa se implementa sobre bases de datos relacionales, las cuales disponen de esquemas. Cada tenant dispone de su esquema propio que contiene el conjunto de tablas donde se almacenan sus datos. Esta alternativa no se considera en este trabajo puesto que *MongoDB,* el motor de bases de datos de *WStore* no dispone de soporte para esquemas.

Misma base de datos, mismo esquema

TenantID	CustName	Address

TenantID	ProductID	ProductName

TenantID	Shipment	Date
4711	324965	2006-02-21
132	115468	2006-04-08
680	654109	2006-03-27
4711	324956	2006-02-23

Ilustración 7: Misma base de datos, mismo esquema

En este caso los datos de los diferentes *tenants* estarán almacenados en una única base de datos. Este es el enfoque que mayor esfuerzo de implementación requiere pero una vez que ha sido desplegado el servicio, esta implementación favorece la compartición de recursos, reduciendo costes de mantenimiento y facilitando las labores de administración. Gracias al *sharding* este enfoque también permite el escalado horizontal. *MongoDB* permite, mediante *sharding*, dividir en porciones una base de datos y distribuirla dinámicamente según va creciendo entre los diferentes servidores (*shards*) que conforman el sistema de base de datos distribuido. Este es el modelo más eficiente ya que por ejemplo, si cada tenant requiere de 1.5 servidores, para dar servicio a 2 tenants con bases de datos separadas se

necesitan 4 servidores mientras que con el enfoque de una sola base de datos se requieren solo 3 servidores.

Teniendo en cuenta lo anteriormente descrito, se ha tomado la decisión de usar una única base de datos la cual contendrá los datos de todos los tenants. Esta decisión si bien es la más eficiente y escalable, también es la que implica mayor complejidad en el desarrollo para asegurar el aislamiento entre tenants y la seguridad del sistema. A continuación se el proceso de desarrollo para hacer de *WStore* un servicio *multitenant*.

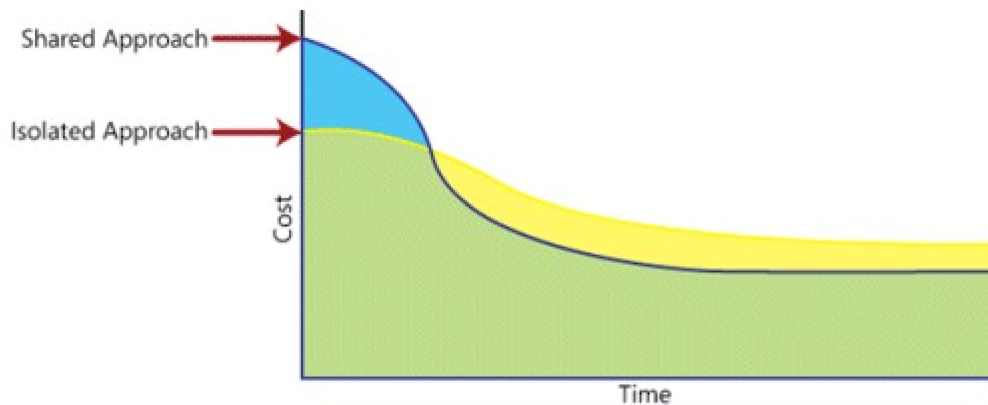


Ilustración 8: Comparativa entre BBDD compartidas o separadas

3.2 Información de tenants

Lo primero de todo es necesario disponer de la información de los tenants existentes en el sistema. Para esto se ha añadido un nuevo modelo a los ya existentes en *WStore*. Los modelos en *Django* representan la información que contendrá la base de datos y en definitiva los datos con los que trabaja la aplicación. Se puede considerar los modelos de

Django como las definiciones de las tablas de una bases de datos *SQL* o las colecciones de la base de datos *MongoDB.*

Los modelos más importantes definidos en *WStore* son:

- **Organization**, para representar organizaciones en la plataforma. Estas almacenan información cómo puede ser nombre, dirección, administradores, ofertas compradas e información de pago.
- **UserProfile**, que representa el perfil de cada usuario registrado y contiene información tal como usuario, nombre completo, dirección, organizaciones a la que pertenece el usuario ofertas compradas y publicadas e información de pago.
- **Offering**, que contiene información de ofertas como el nombre, organización que ha publicado dicha oferta, versión, estado, imágenes, comentarios y fecha de publicación.
- **Resource**, recursos que incluye una oferta, con nombre, versión, proveedor, enlace de descarga, oferta a la que pertenece, etc.
- **Review**, con la información de las valoraciones sobre ofertas de los usuarios. Contiene datos tales como usuario, organización, oferta, fecha valoración o comentarios.
- **Contract**, que representan contratos con la información del modelo de pago del contrato, pagos realizados o pagos pendientes.

- **Purchase**, con información de compras tal como la oferta comprada, el comprador, la organización que ha publicado la oferta comprada o el nombre.

Se añade un nuevo fichero con la definición del modelo *TenantProfile.*

```
class TenantProfile(models.Model):
    REQUEST_STATUS = (
        ('P', 'Pending'),
        ('R', 'Rejected'),
        ('A', 'Approved'),
    )
    tenant_id = models.CharField(max_length=30)
    tenant_admin = models.ForeignKey(User)
    tenant_status = models.CharField(max_length=1,choices=REQUEST_STATUS)
```

Esta contiene los siguientes campos:

- El *tenant_id* es el identificador único del *tenant*, que será una cadena de caracteres con el nombre de dicho *tenant*. Este identificador será el nombre de subdominio del *tenant*, por ejemplo, si el dominio de la *store* es http://mystore.com, el tenant con ID=upm será accesible desde el subdominio http://upm.mystore.com.
- El *tenant_admin* es el usuario con permisos de administrador, a este tipo de usuarios se les llama indistintamente "*staff*" o administrador a lo largo de este trabajo.
- El *tenant_status* puede ser 'A','P' ó 'D'. Un *tenant* no estará disponible hasta que el administrador apruebe la solicitud. 'A' significa que la solicitud de *tenant* ha sido aprobada, 'D' significa que ha sido denegada y 'P' que está pendiente de que un administrador tome la decisión de aprobar o denegar la solicitud.

Por defecto, existe un tenant principal, el cual no requiere un documento en la colección *TenantProfile*. Los usuarios registrados en este tenant principal podrán solicitar un tenant propio en la interfaz de la plataforma. Si administrador aprueba la solicitud del nuevo *tenant* este será creado y un nuevo documento será añadido a *TenantProfile*. El usuario del *tenant* principal que realizó la solicitud automáticamente dispondrá de una cuenta en el nuevo tenant creado con permisos de administrador.

3.3 API para multitenancy

La solicitud, aprobación y acceso a los tenants se realiza a través de un *API REST*. Antes de analizar el *API* cabe destacar como *Django* procesa las peticiones *HTTP*. *Django* relaciona cada petición al servidor con una vista (*Django view*), que se trata de código que se ejecuta al recibir la petición. Mediante el fichero *url.py*, *Django* mapea URLs con métodos contenidos en las diferentes vistas. El fichero de urls admite expresiones regulares y parámetros en la url que se pasan a las funciones de vista como parámetros. En este caso, las entrada correspondiente al API es la siguiente:

```
url(r'^api/multitenancy/tenantrequest/?
$',multitenancy_view.TenantRequest(permitted_methods=('POST','GET','PUT')
)),
```

Se accede al *API* mediante peticiones *POST*, *GET* y *PUT* a la url http://example.url/api/multitenancy/tenantrequest . *POST* para crear un nuevo tenant, *GET* para obtener los tenants pendientes de aprobación y *PUT* para aprobar o denegar solicitudes de nuevos *tenants*.

3.4 Django View

En este apartado se explican las vistas que han sido definidas para *WStore Multitenant*. Estas vistas procesar las peticiones recibidas a las diferentes *URLs* de la plataforma para darle soporte *multitenant*.

Para las peticiones al API mencionadas en el apartado anterior se ha definido la clase *TenantRequest* que heredea de la clase *Resource*, implementada en *WStore*. Esta clase permite utilizar vistas como *API REST*. Para esto se define una clase en el fichero *views.py* que herede de *Resource* y que defina los métodos *create, update y read*. Estos métodos se ejecutarán al recibir peticiones *POST*, *PUT* y *GET* respectivamente.

Create crea un nuevo tenant en la plataforma. Para esto se comprueba que la petición viene desde el *tenant* principal ya que sólo los usuarios de este tenant pueden solicitar nuevos. Posteriormente se añade el nuevo tenant a la colección *TenantProfile* con estado ‘P’, indicando que está pendiente de aprobación. A continuación se notifica via *email* al administrador de que una nueva solicitud está pendiente de procesar. Mediante la interfaz de administración de tenants el administrador aprobará o denegará dicha solicitud.

Cuando el administrador accede al panel de administración de tenants verá la lista de solicitudes para la creación de nuevos tenants. Esta lista de tenants pendientes de aprobación se obtiene a través del método **read** del *API REST* que procesa las peticiones *GET* de *HTTP*. Este método primero comprueba que la petición la realiza el administrador del tenant principal y posteriormente construye la respuesta *HTTP* con la lista de solicitudes pendientes que se mostrarán en la interfaz.

El método **update** procesa la petición asociada a la acción de aprobar o denegar la solicitud de nuevo *tenant*. Si la solicitud ha sido rechazada el documento del *tenant* es eleminado. Si la petición fue aprobada, el documento de dicho *tenant* es modificado, estableciendo el estado del *tenant* como aprobado. Cada *tenant* tiene asociado un documento en la colección *Site*, por lo que al crear un nuevo *tenant* se añade un nuevo documento a dicha colección.
Site almacena información acerca de los sitios web que *Django* administra. Este modelo contiene la *URL* principal del sitio y un nombre. Por ejemplo, desde la consola de administración de *MongoDB* podemos inspeccionar que contiene actualmente la colección *django_site* en el entorno de pruebas actual.

```
> db.django_site.find()
{ "_id" : ObjectId("5388b0b67c2d3012acc2a36d"), "domain" : "http://mystore.com:8000", "name" : "mystore" }
{ "_id" : ObjectId("5388b3e37c2d30137eba99b4"), "domain" : "http://upm.mystore.com:8000", "name" : "upm" }
{ "_id" : ObjectId("5388b53a7c2d3013a7e10d5c"), "domain" : "http://euw.mystore.com:8000", "name" : "euw" }
> 
```

Ilustración 9: Ejemplo del contenido de Site

El campo *domain* contiene la *URL* y *name* el identificador del *tenant*.

Tanto si se aprueba o se deniega la solicitud, el usuario responsable recibe un *email* informando de la decisión tomada por el administrador.

A parte del *API REST* se define también una vista para el panel de administración y de solicitud de tenants. Esta vista muestra el documento *HTML* de la interfaz correspondiente.

Otra de las vistas implementadas es *ProviderRequest* que sobrescribe el *API* para la solicitud del rol de proveedor de los usuarios de *WStore*. El motivo por el cual se sobrescribe es que para la plataforma *multitenant* el comportamiento es diferente. En

WStore el administrador de la plataforma es el encargado de aprobar dicha solicitud de nuevo rol, pero en *WStore Multitenant* cada *tenant* tiene su propio administrador. Por tanto,

esta vista se asegura que la notificación llegue al administrador del *tenant* donde se realiza la solicitud de nuevo rol.

Por último, ha sido necesario también añadir a las vistas la clase *MultitenantForm* que hereda de la clase *RegistrationForm*. Mediante esta clase *Django* procesa el formulario de registro, comprobando que el *email* y el usuario son correctos para registrarse en la plataforma. Se ha creado una nueva vista que utiliza esta clase para validar el *email* y el usuario en el registro ya que en *WStore* se comprueba que el *email* y el usuario no existan en la base de datos pero en *WStore Multitenant* podrán existir *emails* y nombres de usuarios iguales, siempre y cuando estén registrados en *tenants* diferentes. Otra alternativa que se consideró previamente fue tener un espacio de cuentas de usuario común a todos los *tenants* manteniendo así el código de registro existente, pero esta opción fue descartada para ofrecer el mayor aislamiento posible de los usuarios entre *tenants*, como si de plataformas *Web* distintas se tratase.

3.5 Django Middleware

Django ofrece la posibilidad de definir clases *Middleware* con código que se ejecuta antes de que las peticiones *HTTP* recibidas sean procesadas por el servidor. El fichero de configuración *settings.py* de *Django* contiene la lista de *middlewares* que procesan las peticiones al servidor en el orden en que aparecen en dicha lista. Esto es importante tenerlo en cuenta ya que algunos *middleware* dependen de que otros se ejecuten previamente por lo que se debe prestar atención al orden en que se declaran en el fichero de configuración.

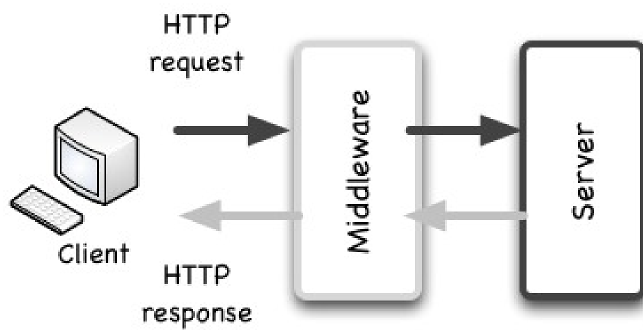


Ilustración 10: middeware

Cada *middleware* es responsable de realizar alguna función específica. Por ejemplo, *Django* incluye el *middleware AuthenticationMiddleware* que asocia usuarios con peticiones durante las sesiones. Las clases *middleware* deben implementar el método

process_request el cual recibirá el objeto *request* de la petición HTTP. Se ha añadido una serie de *middlewares* que interceptan las peticiones y ejecutan código para hacer de *WStore* una aplicación *multitenant*. A continuación se describen los *middlewares* implementados y el objetivo de los mismos:

```
MIDDLEWARE_CLASSES=('wstore.multitenancy.middleware.multitenant_middlewar
e.SetGlobalTenantNameMiddleware','wstore.multitenancy.middleware.multiten
ant_middleware.SignUpInTenant','wstore.multitenancy.middleware.multitenan
t_middleware.ProviderRequestTenat',....
```

- *SetGlobalTenantNameMiddleware* obtiene el *tenant* al que va dirigida la petición mediante la *url* ya que el identificador de *tenant* coincide con el subdominio de este. El primer diseño separaba las *URLs* por puntos y esperando que fueran del tipo http://tenant.dominio.com y http://dominio.com para el *tenant* principal. De esta forma, si la *URL* contenía un solo punto se determinaba que la petición se realizaba al *tenant* principal. En caso de tener más de un punto en la *URL* se obtenía el identificador de *tenant* entre las barras y el primer punto de la *URL*. Actualmente se ha adpotado una solución menos restrictiva ya que la anterior impide que *WStore Multitenant* se instale en un subdominio, con una *URL* del tipo http://wstore.fi.upm.es con *tenants* del tipo http://tenant.wstore.fi.upm.es. El nuevo *middleware* no determina que se trata de una petición al *tenant* principal si la *url* contiene un solo punto, en su lugar primero busca en la colección *django_site* la *url* del *tenant* principal y comprueba si concuerdan. El *tenant_id* obtenido por este *middleware* se almacena en una variable global a la conexión (*_thread_local*). Por tanto, en cada conexión

con el servidor se establece la variable *tenant_id* dependiendo de a que *tenant* ha accedido el usuario. Posteriormente se analizan algunos casos particulares cuando hay modelos que requieren de *tenant_id* pero la variable global que contiene este

valor aún no está definida ya que la instanciación de estos modelos no están asociadas a ninguna conexión.

- *SignUpInTenan* intercepta las peticiones que van dirigidas a la vista de registro de *WStore*. Estas peticiones *HTTP* llevan la url http://dominio_tenant/signup/ en su cabecera y el *middleware* se encarga de modificarla por http://dominio_tenant/multitenant_reg/. Esta modificación permite que *Django* ejecute el código de registro para *WStore Multitenant* que se explicó en el apartado 3.1.4 para responder a la petición de registro.

- *ProviderRequestTenat* intercepta las peticiones que van dirigidas al api de *WStore* para solicitar el rol de *proveedor*. Estas peticiones *HTTP* llevan la url http://dominio_tenant/api/provider/ en su cabecera y el *middleware* se encarga de modificarla por http://dominio_tenant/multitenant_providerRequest/. En *WStore* cuando un usuario solicita el rol de *provider* para publicar ofertas se le notifica al administrador de la plataforma para que este apruebe la solicitud. En *WStore Multitenant* en cambio, cada *tenant* tiene su propio administrador, por lo que esta modificación permite que se notifique al administrador del *tenant* correspondiente cuando se ha realizado una petición de nuevo rol.
- Además, en este módulo se define una función que devuelve el objeto *Site* del *tenant* principal. Esta función es utilizada en *SetGlobalTenantNameMiddleware* como se explicó anteriormente y además es llamada desde el módulo de señales ques se tratará a continuación.

3.6 Señales

Django permite configurar señales asíncronas que ejecutan código en determinadas situaciones. *WStore Multitenant* utiliza las señales para interceptar los accesos a la base de datos para de esta manera añadir el *tenant_id* a los documentos previa modificación de la estructura de las colecciones que contendrán el atributo *tenant_id*. Algunas de estas señales son:

- *pre_init*, que se activa cuando se instancian los modelos en *Django*.
- *pre_save*, que se activa antes de realizar una escritura en la base de datos.
- *class_prepare*, que se activa cuando se crea un modelo por primera vez, lo que permite modificar la estructura de una colección antes de establecerla en la base de datos.

La señal que se utiliza es pre_save, y se declara en el fichero *__init.py__* . El fichero *__init.py__* se ejecuta antes que cualquier otro fichero de *WStore Multitenant* por lo que la señal estará disponible desde el inicio de la ejecución.

```
pre_save.connect(tenant_save)
```

Al ejecutarse esta instrucción quedará asociada la función *tenant_save* para ejecutarse cada vez que se realicen escrituras en la base de datos. El código de *tenant_save* es el encargado de modificar las colecciones de la base datos para que contengan el campo *tenant_id* con el valor del *tenant* al que pertenece dicha colección. El funcionamiento de la señal es el siguiente:

- Se recibe la señal antes de crear o modificar un documento en la base de datos. La señal ejecuta la función *tenant_save*.

- La función comprueba que se trate de la inserción de un nuevo documento, si se trata de la modificación de uno existente esta función no hace nada.
- La función añadirá el campo *tenant_id* sólo a las colecciones que contendrán datos exclusivos de cada *tenant*, como por ejemplo las colecciones *Offering*, *UserProfile*, etc. Un caso particular es cuando el modelo que dispara la señal es *User*, en este caso no se añade *tenant_id* sino que se modifica el nombre de usuario que se va a guardar en la base de datos. La razón por la que se implementa esto será explicada en el siguiente apartado.
- Se añade el campo *tenant_id* a la colección gracias a la función *add_to_class* de *Django* que permite modificar el objeto que representa a la colección para añadirle este atributo.
- Se añade el *manager* con soporte para *multitenancy*. En el siguiente apartado se explica con más detalle esta acción.
- Se añade al documento el valor de *tenant_id*. Este valor se obtiene gracias al *middlware SetGlobalTenantNameMiddleware* explicado en el apartado anterior.

3.7 Problemas

La colección *auth_user* no pertenece a *WStore*, sino que es usada por *Django* para administrar los usuarios y su autenticación en las plataformas web que soporta. Al ser un modelo propio de *Django* este no puede ser alterado (añadir *tenant_id)* por la función *tenant_save* que se ha implementado y que se explicó anteriormente (el *manager* de este modelo si puede sobrescribirse). Por tanto, se trata de una colección global, que contiene todas las cuentas de usuario del sistema. Que la estructura de la colección *auth_user* no pueda ser modificada supone un problema ya que cada documento de *auth_user* tiene asociado un documento *UserProfile*, que extiende la tabla *auth_user* con más información de usuario. *UserProfile* contiene información exclusiva de cada *tenant*. Para seguir este convenio en *WStore Multitenant* un usuario tendrá asociado un documento en *auth_user* (y por tanto también un documento *UserProfile*) por cada cuenta que tenga en los diferentes *tenats*. El problema surge cuando se crean usuarios con el mismo nombre de usuario en diferentes *tenants.* Esto debe permitirse debido a la independencia entre *tenants* pero *Django* no permite dos documentos en esta colección con el mismo nombre de usuario y además como explicamos anteriormente, la estructura de *auth_user* no puede ser modificada.
La solución al problema descrito anteriormente ha sido añadirle un sufijo a todos los nombres de usuario de los documentos que se guarden en *auth_user*. El prefijo se construye de la siguiente forma: nombre_de_usuario$tenant_id. Un nombre de usuario que se guarda en un usuario registrado en el *tenant* principal es de la forma nombre_de_usuario$ mientras que este mismo usuario si se registra en el tenant “upm” tendrá asociado otro documento en *auth_user* con el nombre de usuario nombre_de_usuario$upm. Este sistema de nombrado es transparente al usuario y se añade de forma automática. De esta forma la colección *auth_user* contendrá tantos documentos como cuentas tenga el usuario en los *tenants* y por tanto, el usuario podrá tener un documento *UserProfile* para cada *tenant*.

También ha habido un problema con la colección *Context*. Esta colección contiene información de la plataforma, exclusiva de cada *Site* (cada *Context* tiene una *foreign key* del *Site* al que pertenece). Al realizar las pruebas se descubrió que los documentos *Context* de cada *site* tenían como *tenant_id* el *tenant* principal. Esto se debe a que estos datos se escriben en la base de datos antes que se reciban conexiones a la plataforma, que es cuando se guarda el valor *tenant_id* como variable global a dicha conexión. Por tanto en la función *tenant_save* se trata el modelo *Context* de forma particular para establecer el *tenant_id* al que pertenece. La variable *tenant_id*, global a las conexiones no está disponible en el momento que se debe guardar en la base de datos el *tenant* al que pertenecen los documentos *Context* (ya que no se realiza dentro de ninguna conexión) por lo que el *tenant_id* se obtiene del documento *Site* asociado a cada *Context*

Un problema similar al anterior ocurría con con el modelo *UserProfile*. *WStore* automáticamente crea un documento *UserProfile* con el perfil de usuario en el momento en que se añade un nuevo usuario a *auth_user*. Como vimos anteriormente en el apartado del *API* para solicitud de *tenants*, cuando se aprueba un *tenant* se crea un nuevo usuario con permisos de administración en el nuevo *tenants* creado. La aprobación de los nuevos *tenants* las realiza el administrador del *tenant* principal, por lo que la variable global *tenant_id* contiene el identificador del *tenant* principal. Esto provoca que el campo *tenant_id* del documento *UserProfile* asociado al nuevo usuario no pueda obtenerse de dicha variable global. En este caso, el *tenant_id* se obtiene del nombre de usuario del documento *auth_user* asociado ya que este es de la forma nombre_de_usuario$*tenant_id* como vimos anteriormente. Este problema también ocurre en el modelo *Organization* puesto que *WStore* también crea una organización privada para el nuevo usuario registrado.

Un problema de diseño que fue corregido en este módulo fue la elección de la señal *class_prepare* para añadir el *tenant_id* a las colecciones con documentos privados a cada *tenant*. Esta señal se activa cuando un modelo es instanciado por primera vez. El problema

descubierto en la utilización de esta señal es que la creación de los modelos en algunos casos se produce antes que se declare esta señal. Como *Django* no asegura que en el momento de la instanciación de los modelos todas las señales están activas, no podemos considerar esta alternativa puesto que no podemos asegurar que funcione en todos los casos.

3.8 Manager

El *manager* de un modelo es el objeto encargado de procesar las peticiones a la colección que representa. Este objeto define las funciones que se ejecutan con las consultas a la base de datos. Por ejemplo, para obtener los datos de un usuario de la base de datos se ejecuta la función *get* del *manager* del modelo *User*. *Django* permite utilizar *manager* personalizados que hereden de la clase *Manager* original.
Esta funcionalidad se usa para filtrar los datos en las respuestas de la base de datos por *tenants*. De esta manera cuando en el código de *WStore* realice lecturas en la base de datos, los resultados obtenidos serán los correspondientes al *tenant* desde el que se solicitan. Por ejemplo, en el proceso de *login* la plataforma comprueba el nombre de usuario y la contraseña en la base de datos para permitir el acceso. Por esta razón el *manager* debe filtrar los resultados de la consulta, descartando las posibles cuentas que tenga ese usuario en otros *tenants*.

Para *WStore Multitenant* se han definido los siguientes *managers*:

- *MultitenantManager,* que es usado por todos los modelos excepto *Profile*. En este manager se ha sobrescrito la función *get_query_set*. Esta función se ejecuta en todas las lecturas de la base de datos y se modifica para que filtre el conjunto de datos que conforman la respuesta, descartando los documentos que no pertenezcan al *tenant* desde que se realiza la petición.

- *UserMultitenantManager* se ha implementado para sobrescribir el *manager* de la colección *auth_user* de *Django* (Modelo *User*) y permitir que el *login* en *WStore Multitenant* funcione de forma correcta. Como explicamos anteriormente, el nombre de usuario queda registrado en *Django* con el sufijo $*tenant_id* por lo que en el proceso de *login* el usuario debe acceder con el nombre de usuario en este formato. La adición de este sufijo es transparente al usuario y este *manger* se encarga de que así sea. Se sobrescriben las funciones *get*, *filter*, *update*, *exclude*, *get_or_create* y *create* que son las funciones del *manager* de_*User* que podrían recibir como parámetro el nombre de usuario. Al sobrescribir estas funciones se comprueba si entre los parámetros de llamada se encuentra un filtrado por nombre de usuario y se modifica el valor de este. Por ejemplo, si se realiza una consulta a la base de datos para obtener el documento cuyo nombre de usuario es “pepe”: **`User.objects.get(`*`username=pepe`*`)`** se modifica la condición de la consulta para que esta sea **`User.objects.get(`*`username=pepe$tenant_id`*`)`**. De esta forma todas las peticiones que se realizan a la base de datos se harán con el formato correcto nombre _de_usuario$*id_tenant*. Este *manager* también sobrescribe la función *create_user* para que el API de solicitud de *tenants* cree el usuario administrador en los nuevos *tenants* aprobados. La función clona el documento del usuario que ha solicitado el *tenant* modificando el nombre de usuario para añadirle

el sufijo $*tenant_id* con el identificador del *tenant* que se ha creado y declara la cuenta como activada y con permisos de administración.

3.9 Interfaz web

En el lado del cliente también se han realizado modificaciones para añadir la opción de que el usuario solicite un nuevo *tenant*. El formulario para dicha solicitud requiere que se establezca en identificador del *tenant*. Con esta información se realiza una petición *POST* al *API* de solicitud de *tenants*. Las llamadas al *API* se realizan por medio de *AJAX* desde el código *Javascript* del cliente.
Posteriormente el administrador podrá acceder al panel de administración para aprobar o denegar la solicitud de nuevo *tenant*.

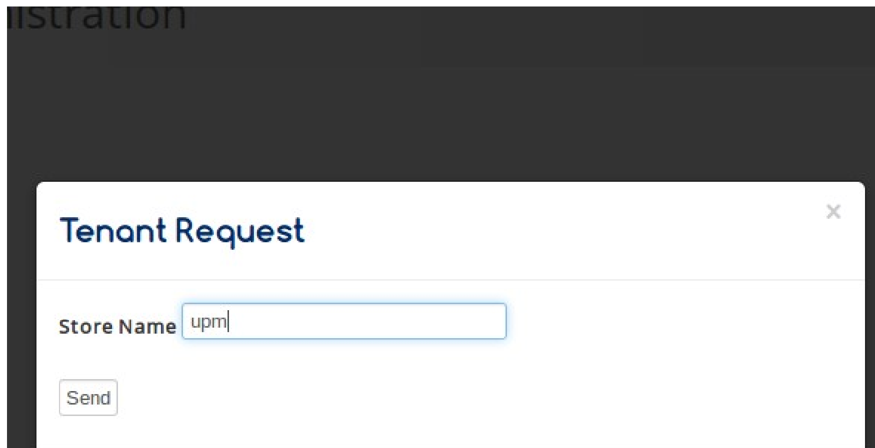


Ilustración 11: Interfaz web de solicitud de tenats

3.10 Organización del código e instalación manual

A lo largo de los apartados anteriores se ha explicado el funcionamiento de los módulos desarrollados para *WStore Multitenant*. El código que forman estos módulos está ubicado dentro del árbol de directorios de *Django*, por lo que para la instalación de *WStore Multitenant* es necesario que estos ficheros sean colocados en el lugar adecuado. *Django* organiza la funcionalidad de las plataformas desarrolladas en directorios llamados *Apps*.

Dentro del directorio del proyecto se encuentran los ficheros *models.py*, con los modelos de datos, *views.py* con las vistas (código que se ejecuta dependiendo de la URL de la petición recibida por la plataforma), un directorio *Static* para el código de la interfaz web (HTML, CSS y Javascript) y otra serie de ficheros pertenecientes a *WStore*. Además, cada *App* tiene su carpeta en este directorio con la definición de sus propios modelos y vistas. Se añade una nueva *App* mediante el directorio *Multitenancy* y añadiendo la ruta de dicho directorio a la lista de *Apps* instaladas en la configuración de *Django:*

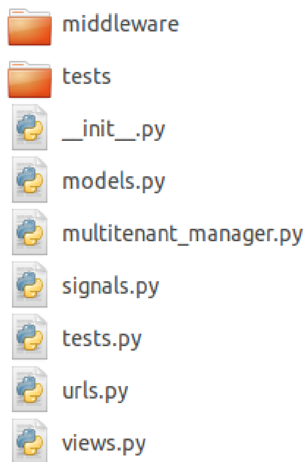


Ilustración 12: Arbol de ficheros del código

El directorio *middleware* contiene el fichero *multitenant_middleware.py* con la funcionalidad descrita en el apartado *Django Middleware*. El fichero *__init__.py*, primero en ejecutar contiene la declaraciones de la señal usada y la asignación del *manager* implementado para el modelo *User* (Colección *auth_user*). El fichero *models.py* contiene el

modelo de datos que representa los *tenants* (*TenantProfile*) . En *multitenant_manager.py* implementada la funcionalidad del apartado *Manager*, al igual que *signals.py* contiene la implementación de la señal *tenant_save* con sus funciones auxiliares. El fichero *tests.py* contiene pruebas unitarias por lo que no ofrece ninguna funcionalidad. En *url.py* está la relación entre las *URL* de las peticiones y el código de *views.py*, lo que permite ejecutar el código adecuado según la petición que se realice. Entre el código que contiene *views.py* se encuentra el *API* desarrollada para la gestión de solicitudes de *tenats* y el renderizado de la plantilla para la interfaz web de gestión de *tenants*.

Las plantillas *HTML*, el *CSS* y el código *Javascript* de la interfaz web va situado dentro del directorio *Static* de *WStore*, en una carpeta llamada *Multitenancy* junto a los ficheros estáticos de *WStore.*

4.0 Despliegue de una Base de Datos Distribuida

En el apartado anterior se implementó el *software* necesario para ofrecer la funcionalidad de *multitenancy* de cara a ofrecer *WStore* como un servicio web. Que la plataforma se convierta en un servicio implica que será usada por un número mayor de clientes, un número que potencialmente irá aumentando con el tiempo. Por tanto, la base de datos debe estar preparada para el aumento de la carga de trabajo y el volúmen de los datos. Ofrecer un servicio supone una responsabilidad de cara al cliente, que espera que sus datos conserven la confidencialidad, integridad y la disponibilidad. Por este motivo, no basta con que la aplicación sea *multitenant* sino que la instancia que se ofrezca como servicio en Internet debe contar con una capa de persistencia de datos escalable y de alta disponibilidad. En este apartado se estudia la funcionalidad que ofrece *MongoDB* de cara al despliegue de *WStore Multitenant*.

4.1 Escalado median te Sharding automático

La arquitectura propuesta para la base de datos es la siguiente:

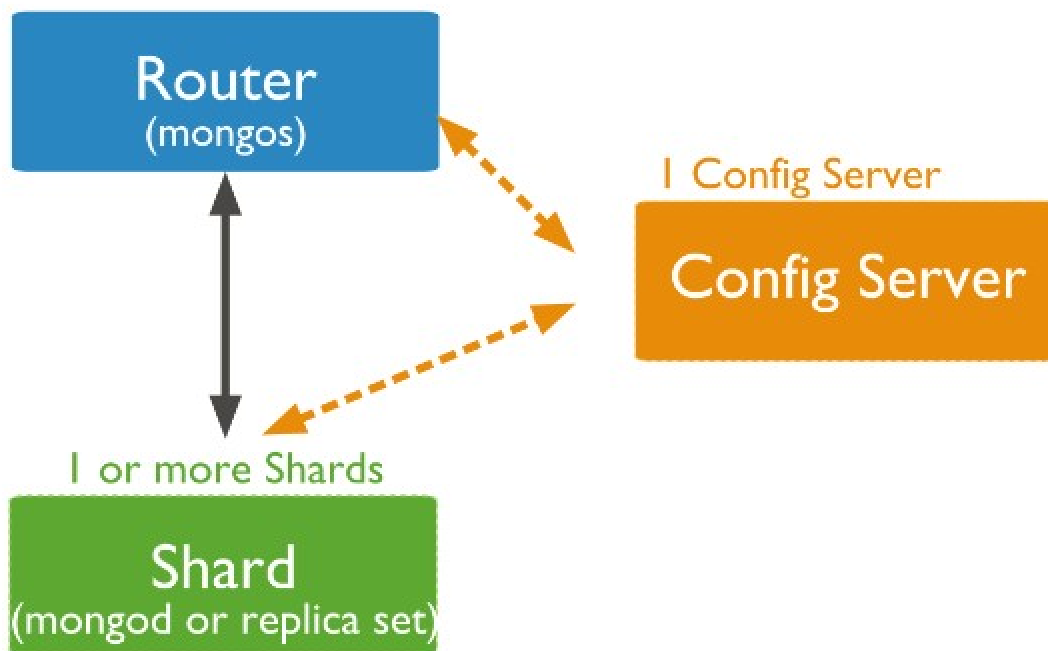


Ilustración 13: arquitectura para sharding

La arquitectura implementada como entorno de pruebas consta de 6 máquinas virtuales, 3 de las cuales son servidores *Shard*, que almacenan los datos, una es el servidor *Query Router* que procesa los accesos a la base de datos y una es el *Config Server* que contiene los metadatos del sistema. En el siguiente apartado se han realizado pruebas con 11 máquinas virtuales, sustituyendo cada uno de los servidores de *sharding* por *replica sets* de tres servidores. Los servidores están en la misma red local ya que deben comunicarse entre ellos.

4.1.1 Descripción del servidor de configuración

El servidor de configuración o *Config Server* almacena los metadatos de la base de datos distribuida. Este servidor es el encargado de relacionar los datos con el *Shard* correspondiente por lo que el servidor *Query Router* consulta estos metadatos para dirigir las operaciones de la base de datos sobre el *shard* adecuado. Este servidor debe ser el primero en ejecutar y permanecerá escuchando peticiones en el puerto 27019. Para ejecutar el *Config Server* se ejecuta el comando: **`mongod --configsvr`**

Para un entorno de producción se recomienda disponer de 3 servidores de configuración ofreciendo redundancia al *cluster* y evitando interrupciones en el servicio a causa de la caída de un servidor de configuración.

4.1.2 Descripción del servidor de consulta

El servidor de consulta o *Query Router* es la interfaz con la aplicación cliente, por lo que recibe las peticiones a la base de datos. Este servidor enruta las operaciones a los shards correspondiente y devuelve los resultados al cliente. Para ejecutar el *Query Router* en el servidor de base de datos se ejecuta el comando: **`mongos --configdb`**

config-server:27019, donde **config-server** es el nombre del servidor de configuración que arrancamos anteriormente en el puerto 27019.

4.1.3 Servidores de almacenamiento

Los servidores de almacenamiento o *Shard*s almacenan las distintas particiones de los datos. *MongoDB* particiona las colecciones de la base de datos según una clave. Por ejemplo, si utilizamos el índice como clave en la colección de usuarios, se creará una partición para cada usuario. Estas particiones son repartidas automáticamente entre los tres servidores *shard*. Si en el futuro se añade un nuevo *shard*, *MongoDB* automáticamente redistribuye las particiones entre los cuatro servidores.

Ejecutamos el comando: `mongod --shardsvr` en cada servidor de almacenamiento. De esta forma tenemos todos los servidores de la base de datos distribuida en ejecución. A continuación se debe configurar los parámetros de la base de datos. Para la configuración se accede al servidor *query-router* por *ssh* y se procede a añadir todos los servidores *shard* al *query-router* desde la consola de administración de *MongoDB.* De esta forma se se establece la red de almacenamiento distribuida. Los servidores *shards* se añaden escribiendo el siguiente comando en la consola de administración de *MongoDB* del *query-router*:

```
sh.addShard(nodeX:27018)
```

donde nodeX es el nombre de equipo o dirección *IP* de cada servidor de *sharding* y 27018 es el puerto por defecto que estos utilizan para recibir conexiones.

Posteriormente se debe habilitar el *sharding* desde la consola de administración de *MongoDB* con el comando:

```
db.runCommand({enableSharding: “wstore_multitenant_db”});
```

donde wstore_multitenant_db es la base de datos de *WStore*. Luego deben seleccionarse las colecciones de la base de datos que van a ser distribuídas entre los *shards* con el comando:

```
sh.shardCollection(“wstore_multitenant_db.colecciónX”,“clave_de_sharding”})
```

donde colecciónX son las colecciones que participarán en el proceso de *sharding*. Las colecciones de *WStore* que serán distribuidas son: *Application, Code, Token, Organization, Contract, Unit, Purchase, Review, Offering, Resource, UserProfile, Context y Profile* ya que son las que contendrán datos independientes en cada *tenant* y por tanto serán las que aumentan considerablemente de tamaño a medida que se preste el servicio a más clientes. En el siguiente apartado se analiza el proceso de selección de la clave de *sharding.*

Además se debe configurar *Django* para que pueda hacer uso de la misma para el almacenamiento de los datos. En el fichero de configuración *settings.py* se debe establecer el *host* y el puerto de la base de datos con la *IP* y puerto del *Query Router*.

4.1.4 Elección de la clave

Otro aspecto importante es la elección de la clave para el *sharding*. Esta clave debe seleccionarse cuidadosamente pues una mala elección puede ocasionar que el sistema no funcione de forma adecuada.

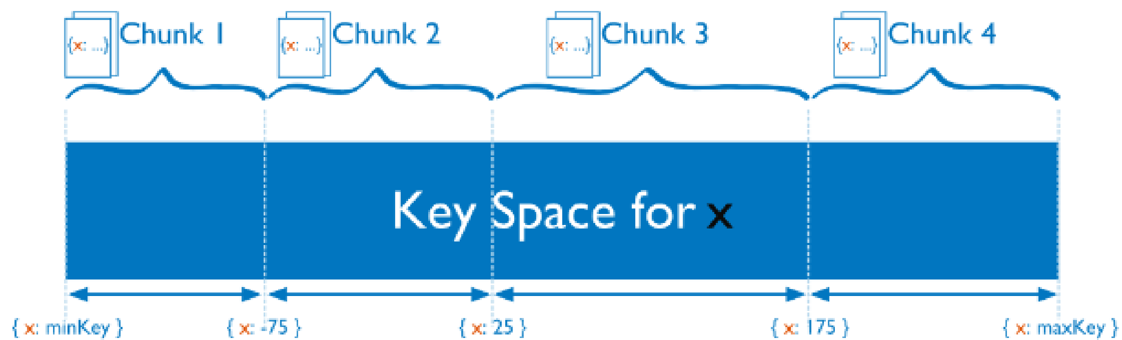


Ilustración 14: Clave para Sharding

MongoDB divide los datos según los posibles valores que tome la clave proporcionada y asigna de forma uniforme las particiones a los diferentes servidores de *sharding*. Esto es todo lo que asegura automáticamente el sistema, la elección de la clave condiciona tres importantes aspectos de la base de datos:

- Distribucion de lecturas y escrituras a través de los servidores. Esta distribución es uno de los aspectos más importantes ya que si el sistema siempre realiza la mayor parte de las escrituras en un *shard*, este estará sobrecargado mientras que otros servidores permanecerán ociosos. Esto es un problema de escalabilidad que se debe evitar eligiendo una clave adecuada. Por ejemplo, no se deben seleccionar claves que crezcan uniformemente como puede ser un *timestamp* ya que siempre se estarán añadiendo datos en el último *shard*.
- Tamaño de las particiones. Este es un parámetro importante puesto que las particiones muy grandes pueden sobrecargar el servidor donde se encuentren. *MongoDB* divide las particiones grandes en trozos más pequeños para intentar mantener la uniformidad entre los *shards*, pero solo divide las particiones que tienen

claves diferentes. Si un valor de la clave contiene grandes cantidades de datos, *MongoDB* no podrá dividirla. Esto implica que no debe usarse una clave que no asigne uniformemente sus valores entre todo su espacio, como por ejemplo el *tenant_id*. Asignar el *tenant_id* como clave implica que cada tenant tenga asociado un *shard* ypuede provocar que si los datos de un *tenant* aumentan, *MongoDB* no sea capaz de dividirlo para balancear la carga entre los servidores y por tanto saturen el *shard* donde se encuentran mientras otros servidores están infrautilizados.

- Proximidad de los datos. La eficiencia del sistema depende del tiempo de respuesta de la base de datos para procesar una solicitud. Si se recibe una petición que requiere consultar varios servidores, la latencia de esta operación impactará negativamente en el tiempo de respuesta. Por tanto, es deseable que una consulta a la base de datos pueda resolverse internamente en un *shard*, evitando el mayor número de conexiones entre servidores del sistema. Por tanto la utilización de una clave *hash* dispersa de forma aleatoria los datos a través de todos los *shards*, lo que tendría un impacto negativo en el rendimiento del sistema. Lo ideal en este caso es que cada *tenant* tenga sus datos en un solo servidor ya que las consultas que un cliente de este *tenant* realice podrán ser resueltas sin tener que acceder a otro *shard*.

La solución adecuada en este caso es usar una clave compuesta por el *tenant_id* y el índice del documento. El índice del documento es único y permite que los datos de un *tenant* puedan ser divididos si la partición de sus datos creciera demasiado. De esta manera se logra un equilibrio entre mantener la proximidad de los datos y el tamaño de las particiones que realiza *MongoDB*.

4.2 Alta disponibilidad y tolerancia a fallos mediante Replica Set

La arquitectura de pruebas expuesta en el apartado anterior se puede mejorar sustituyendo los servidores *shards* por *replica sets* para de esta forma ofrecer un servicio de alta disponibilidad, tolerante a fallos y preparado para posibles fallos en los servidores de almacenamiento. Esta característica es esencial para una plataforma que ofrece un servicio web y donde los datos son la parte más importante del mismo, como es el caso de *WStore Multitenant*. A continuación se describe el entorno de pruebas que ha sido implementado para probar esta funcionalidad. La configuración de los *replica sets* le da robustez a la arquitectura escalable basada en *sharding*

Para la alta disponibilidad se utilizan *replica sets* de 3 servidores en cada *shard*. Esto implica que los datos de las particiones asignadas a cada *shards* están replicados y sincronizados en los estos servidores. La redundancia de los datos, protegiendo al servidor de posibles fallos en el *hardware* garantiza la alta disponibilidad en *WStore Multitenant* reduciendo el tiempo que la plataforma deja de ofrecer el servicio debido a fallos de *hardware* en alguno de los servidores de almacenamiento.

Los servidores que forman el *replica set* ejecutan el demonio *mongod*, que es el encargado de recibir las peticiones a la base de datos. En nuestro entorno de pruebas tenemos 3 servidores, es muy importante dejar claro que esta configuración no es la recomendada en entornos de producción, se trata de la más sencilla posible al estar realizando pruebas con máquinas virtuales. Lo recomendado para ofrecer la alta disponibilidad de la base de datos es crear un replica set mayor (el máximo de servidores que soporta un *replica set* es 11) distribuyendo los servidores en situaciones geográficas diferentes.

Los servidores que forman el *replica sets* desplegado son:

- Un servidor primario, el único en el *replica set* encargado de procesar las operaciones de escritura. Las operaciones quedan registradas en un *log*. Por defecto, las operaciones de lectura también van dirigidas a este servidor pero el sistema se puede configurar para modificar este comportamiento, permitiendo distribuir las lecturas entre todos los servidores.

- Dos servidores secundarios que replican los datos del servidor primario a través del *log* de operaciones que éste mantiene.

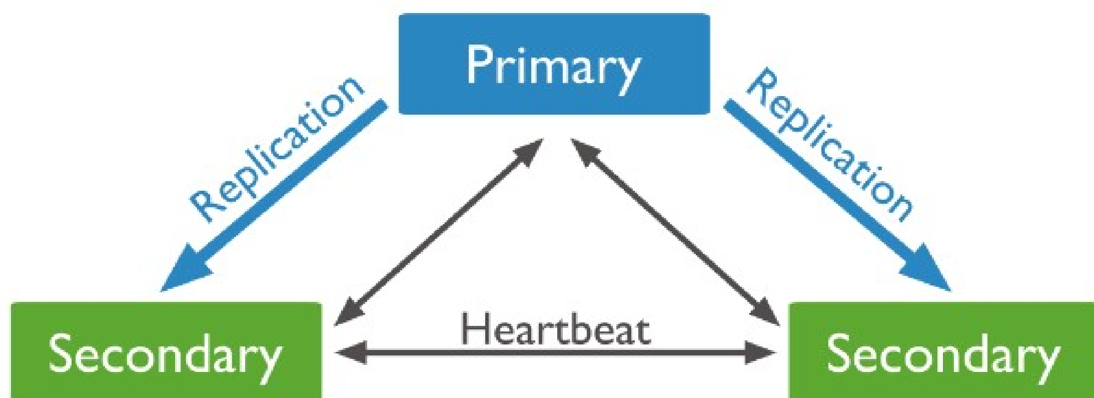


Ilustración 15: Conexiones entre servidores del replica set

Los servidores secundarios mantendrán una copia exacta de los datos del servidor primario, si el servidor primario falla, automáticamente se inicia el proceso de elección del nuevo servidor primario. Es necesario que en todo momento exista un servidor primario ya que es el encargado de procesar las peticiones de escritura en la base de datos.

Los servidores del *replica set* se enviarán *pings* los unos a los otros cada dos segundos, si alguno no responde en menos de 10 segundos, este será marcado como inaccesible. Si el servidor primario está inaccesible se inicia un proceso de elección. El proceso de elección provoca que se elija un nuevo servidor primario, encargado de recibir las peticiones de escrituras en la base de datos. Las diferentes situaciones que provocan el inicio del proceso de elección son las siguientes:

- La iniciación de un nuevo *replica set*.
- Un miembro secundario pierde contacto con el primario.
- El miembro primario es obligado a convertirse en secundario.
- Si un secundario es elegible para elecciones y posee un mayor indice de prioridad.

La elección de un nuevo miembro primario se basa en las prioridades de los miembros elegibles, esta prioridad es por defecto 1, esto para darles a todos los miembros la posibilidad de ser elegidos ya que se tratan de 3 servidores iguales. Los servidores con prioridad 0 no podrán ser elegidos (no existe ninguno de prioridad 0 actualmente) y los servidores preferirían votar por los miembros de mayor prioridad. El proceso de elección no se llevará a cabo si el servidor primario tiene mayor prioridad. Si un servidor secundario tiene mayor prioridad y tiene sus datos actualizados con un margen de 10 segundos se puede iniciar el proceso de elección para permitir al servidor de mayor prioridad convertirse en primario. Para que un servidor pueda convertirse en primario, este debe poder conectar con la mayoría de servidores. En la arquitectura propuesta en este trabajo, el sistema podrá elegir un primario sólo si dos servidores pueden conectarse entre ellos. Si se

da la situación en el que sólo un servidor está funcionando, si se trata del primario pasará a ser secundario y si es secundario permanecerá así al no poder conectar con la mayoría de los servidores del *replica set*.

Con este comportamiento se dota a cada *shard* de tolerancia a fallos y alta disponibilidad en un entorno real, donde los miembros del *replica set* se encuentran en situaciones geográficas diferentes.

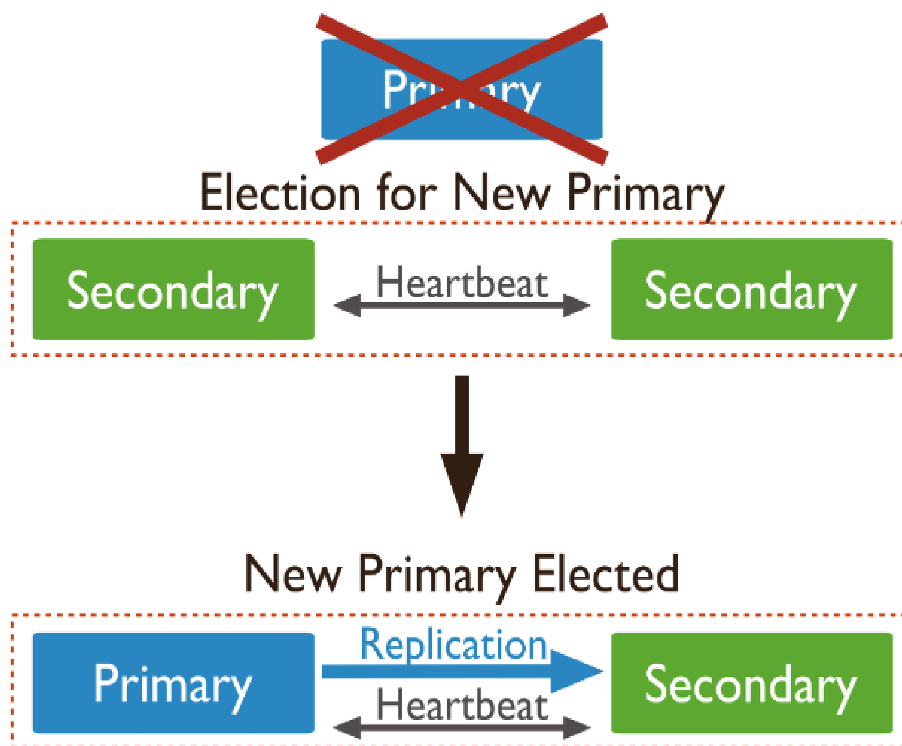


Ilustración 16: Proceso de elección en el replica set

Por defecto, las lecturas a la base de datos van dirigidas al servidor primario del *replica set*. Se ha modificado la configuración, para que todos los miembros puedan aceptar escrituras. Esto balancea la carga de trabajo entre los componentes del *replica set*, lo que disminuye la latencia de las operaciones.

Para el despliegue de los *replica set* se han seguido los siguientes pasos:

- Añadir los servidores a la red local del *replica set*
- Instalar *MongoDB* en los servidores e iniciar el proceso mongod con el comando **`mongod -- replSet "nombredeRS"`.**
- Ejecutar *MongoDB* con el comando ***mongo*** y una vez dentro de la consola de administración se inicializa el *replica set* con el comando **`rs.initiate()`.**
- Posteriormente se añaden los servidores al *replica set* con el comando **`rs.add('nodeX:puerto')`** donde ***nodeX*** es la dirección *IP* de los otros 2 servidores que formarán el *replica set* y ***puerto*** es el puerto en el que escucha las peticiones el demonio *mongod.*
- Al finalizar los pasos anteriores el *replica set* ya se encuentra configurado y automáticamente realiza el proceso de elección para determinar cual será el servidor primario.

5.0 Conclusiones

Este Trabajo de Fin de Grado ha resultado en la implementación de los módulos necesarios para dotar a la plataforma *WStore* con la funcionalidad de *Multitenancy* y de escalabilidad.

Con respecto a la primera de ellas, se ha tenido especial cuidado en que esta funcionalidad se ofrezca a modo de *plugin* por lo que la implementación se ha realizado en ficheros independientes y sin realizar modificaciones en el código original de *WStore*. Esta decisión permite dejar invariante el software original, pero ha añadido complejidad a la realización de este TFG, sin embargo la flexibilidad de *Python* y *Django* ha permitido cumplir con este objetivo. Gracias a este trabajo, en el momento de desplegar una instancia de *WStore* se podrá escoger entre una instalación como plataforma web o una instalación como servicio *Web*, disponible para el uso de cualquier organización que desee disponer de su propia plataforma. Del mismo modo, durante el proceso de diseño e implementación de la solución *multitenant* han surgido una serie de problemas a los cuales se les ha dado solución, obteniendo como resultado una serie de conocimientos útiles para cualquier trabajo que requiera dotar una una plataforma *Django* de la funcionalidad de *multitenancy*.

Además en este trabajo se ha estudiado las herramientas que ofrece *MongoDB* para desplegar una base de datos distribuida escalable y de alta disponibilidad, introduciendo una arquitectura básica para pruebas que es la base para el despliegue de la plataforma en un entorno de computación en la nube real.

5.1 Conclusiones personales

Personalmente valoro positivamente la realización de este trabajo ya que me ha permitido trabajar con tecnologías que no son enseñadas en la carrera. He mejorado mi capacidad para enfrentarme a problemas de los que no se conoce la solución y que requieren de búsqueda de información técnica específica e ingenio de nuevas soluciones. Me ha dado la experiencia de trabajar dentro de un proyecto a nivel europeo con un entorno de trabajo educativo y de investigación. Además, he puesto en práctica muchos de los conocimientos adquiridas a lo largo de la carrera como pueden ser el diseño de algoritmos, programación orientada a objetos, ingeniería de *software*, sistemas distribuidos, bases de datos, concurrencia, *middleware*, entre otras.

5.2 Lineas futuras

El proyecto en el cual se enmarca este trabajo continuará desarrollandose, quedando pendiente el diseño de una arquitectura de producción para un *cluster* desde el que se ofrezca *WStore Multitenanant* como servicio, basándose en las pruebas realizadas en este trabajo.

Queda pendiente también para trabajos futuros hacer de *WStore Multitenant* un servicio elástico, que sea capaz de regular el consumo de recursos de forma dinámica de acuerdo a la carga de trabajo que tenga en cada momento.

También puede mejorarse la forma en la que se instala la funcionalidad de *Multitenancy*, pudiéndose ofrecer alguna herramienta para realizar la instalación de forma automática.

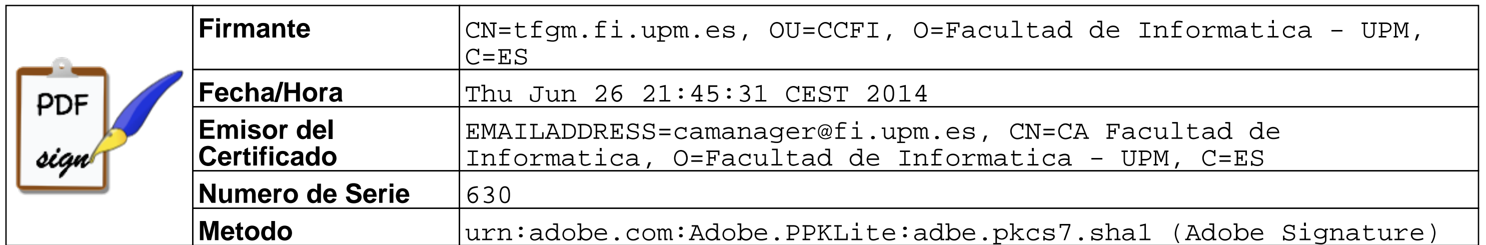
6 Bibliografía

- Cloud computing: state-of-the-art and research challenges Qi Zhang, Lu Cheng Scale Hacking: Cloud Computing, Software and System Performance, Moshe Kaplan. http://top-performance.blogspot.com.es
- Multi-Tenant Data Architecture, Frederick Chong, Gianpaolo Carraro, and Roger Wolter: http://msdn.microsoft.com/en-us/library/aa479086.aspx
- Manual de MongoDB: http://docs.mongodb.org/manual/
- The Django Book: http://www.djangobook.com/en/2.0/index.html
- Manual de Python: https://docs.python.org/2.7/
- Manual de Django: https://docs.djangoproject.com/en/1.6/

7 Referencias

- [1] The NIST Definition of Cloud Computing, http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf
- [2] Aplicación Django-tenant-schemas https://github.com/bernardopires/django-tenant-schemas
- [3] Aplicación Django-db-multitenant https://github.com/mik3y/django-db-multitenant
- [4] Aplicación Django-simple-multitenant https://github.com/pombredanne/django-simple-multitenant

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Thu Jun 26 21:45:31 CEST 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)