UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería en Informática

TFG del Grado en Ingeniería Informática

Framework web de uso de sistemas de machine learning.

Presentado por Javier Martínez Riberas
en Universidad de Burgos — 1 de junio de 2017
Tutor: Dr. José Francisco Díez Pastor y Dr. César Ignacio García Osorio

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería en Informática

D. José Francisco Díez Pastor y D. César Ignacio García Osorio, profesores del departamento de Ingeniería Civil, área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. Javier Martínez Riberas, con DNI 71299495R, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado ” .

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 1 de junio de 2017

Vº. Bº. del Tutor:	Vº. Bº. del co-tutor:
D. José Francisco Díez Pastor	D. César Ignacio García Osorio

Resumen

Recientemente la mayoría de empresas que usan productos informáticos están intentando mejorar sus servicios con un enfoque .AI first", esto es más prevalente en los grandes del sector que tratan de aplicar el aprendizaje automático de manera más amplia.

La escalabilidad es un concepto con una definición clara pero muy abstracta, la mejor definición que se ha encontrado se atribuye a Bondi [Bondi 2000] y la define como la capacidad de un sistema para ser capaz de ampliarse para manejar mayor cantidad de trabajo.

La escalabilidad es una propiedad cambiante en el mundo software ya que existen varias fuentes de ampliación del sistema, la más común es la escalabilidad aumentando el numero de copias del software que se ejecutan, tanto en el mismo ordenador de forma paralela y concurrente como en distintos ordenadores (clusters, multiprocesadores y balanceadores de carga [load balancer]). Otras que han aparecido con el tiempo son descomponer el sistema en los subsistemas más pequeños posibles, lo que da origen a la arquitectura de microservicios. También existe una ultima que es separar los servicios en particiones distintas de manera que cada partición se encarga de dar servicio a parte de los datos.

Estas tres opciones de escalado se conocen en [ref The Art of Scaling] como el cubo de la escalabilidad.

Desde el punto de vista de mantenimiento un sistema escalable es aquel al que podemos mantener o incrementar su funcionalidad sin incurrir en una cantidad de deuda técnica demasiado grande.

En este trabajo se busca obtener un sistema escalable en los dos primeros ejes de escalado. El primer eje o dimensión lo vamos a conseguir con contenedores, el sistema que usaremos es docker. La segunda dimensión la obtendremos con la arquitectura de microservicios.

Esto busca dar un fácil acceso a un problema común en el campo del aprendizaje automático que es la clasificación de imágenes. Esto se hace mediante Deep Learning con el framework Tensorflow.

Con esto se pretende dar un ejemplo de una de las muchas maneras de evitar problemas de escalabilidad, un problema bastante común en el mundo del desarrollo software. Esta escalabilidad es tanto de rendimiento como de mantenimiento.

Se busca facilitar una arquitectura fácilmente escalable en ambos sentidos cuyo reflejo en código no es complejo para el futuro aprendizaje. También se intenta proporcionar un esqueleto mutable para que los futuros proyectos que quieran usar un control básico de usuarios puedan usar para centrarse en otras partes o temas de mayor interés.

Para exponer cómo se resuelve el añadir más servicios se ha expuesto un servicio que usa Deep Learning para clasificar imágenes sobre un conjunto de clases (Imagenet).

Descriptores

Mineria de datos, redes neuronales, clasificadores, aplicación web...

Abstract

A **brief** presentation of the topic addressed in the project.

Keywords

keywords separated by commas.

Índice general

Índice de figuras

Índice de tablas

Introducción

El uso mayoritario del aprendizaje automático o machine learning se puede ver en empresas como Google que ha pasado de buscar el termino exacto pedido por el usuario a intentar, con la información de que disponen de averiguar en que contexto se esta buscando por ejemplo si predice que eres programador y buscas R probablemente los primeros resultados sean del lenguaje de programación.

Este trabajo se orienta a conseguir un sistema actual y escalable. Se intentará que los métodos para conseguir este sistema sean lo más novedosos posibles. También se persigue adquirir los conocimientos necesarios para poder replicar este sistema sobre proyectos ya creados.

El sistema en cuestión será una página web, ya que es un sistema altamente accesible (desde casi cualquier plataforma) con mayor facilidad de mantenimiento que la mayor parte de aplicaciones especificas a un dispositivo concreto. Otra ventaja que proporcionan las páginas web es que tendremos acceso a todos los errores y fallos que surjan en ejecución.

Para darle al sistema algún uso más específico se ha decidido exponer un servicio que usa Deep Learning para clasificar imágenes sobre un conjunto de clases correspondientes con las clases de Imagenet [ref].

Se usarán tecnologías puntas para conseguir estos objetivos. Como docker para la escalabilidad y reducción de deuda técnica gracias a la arquitectura de microservicios que facilita.

La metodología que se usará será la integración continua ya que permite reducir la deuda técnica derivada de la integración de distintos servicios entre sí.

Objetivos del proyecto

A continuación se indican los objetivos, tanto teóricos marcados por los requisitos como objetivos que se persiguen con el proyecto. También se incluyen ciertos requisitos que no tienen por que ser obvios pero que en la actualidad se esperan de cualquier aplicación web.

2.1. Objetivos principales

- Conseguir un sistema actual y escalable.
- Seguir principios de desarrollo actualizados.
- Proporcionar control de usuarios a aquella persona que lo necesite y su proyecto encaje con el que aquí se muestra.

2.2. Servidor web y página web

- Arquitectura MVC (Model View Controller)
- Facilidad de uso
- Internacionalización
- Sistema responsivo

2.3. Servicio de Minería

- Capacidad de predicción
- Capacidad de re-entrenamiento

2.4. Proceso de desarrollo

Los puntos siguientes se ven como necesidades fundamentales en cuanto al desarrollo de proyectos actualmente a los que quizá no se de suficiente importancia.

- Sistema para el control de dependencias
- Despliegue de el proyecto
- Sistema de control de versiones
- Tener un proceso de CI (Continuous integration)
- Programar con agilidad
- Programar siguiendo el 'Manifesto for Software Craftsmanship'

2.5. Objetivos personales

- Aprender arquitecturas actuales, las cuales se pueden usar tanto en industria como en academia.
- Avanzar mis conocimientos a partir de los obtenidos en la carrera, sobre todo aquellos que tienen importancia real y quizá no se hayan estudiado suficiente en la carrera
- Profundizar en el entorno de Python, ya que en los últimos años se ha incrementado su importancia tanto para Data Scientists como para desarrolladores web, que son las dos profesiones encuentro muy interesantes.

Conceptos teóricos

Algunos conceptos teóricos tanto de la parte técnica como de la parte de procesos de software

3.1. Deuda técnica

La deuda técnica es una metáfora para el coste que conlleva el dejar código que no tiene un buen diseño en el proyecto. Si seguimos manteniendo el proyecto esta deuda cada vez costará más el arreglarla.

Las razones de incremento de la deuda con el paso del tiempo son que simplemente se le olvide al programador como funciona exactamente cada linea del código y que se incremente la cantidad de código dependiente del anterior (ya que aunque tenga bajo acoplamiento puede dar problemas).

Este tipo de deuda se ve como necesaria, según algunos autores se debe incurrir en deuda técnica ya que es beneficioso para el desarrollo. Esto se debe a que a corto plazo es imposible prever cómo se va a desarrollar el proyecto y unos cambios demasiado tempranos pueden ser un mal diseño a medio o largo plazo. La deuda 'buena' o beneficiosa se suele considerar como aquella que como mucho dura una semana, una vez existe por más tiempo pasa a ser cada vez más costosa de solucionar o 'pagar'.

Una característica importante de esta deuda es que una vez dejas de soportar el mantenimiento de un proyecto la deuda técnica desaparece, al contrario que la deuda financiera.

Se puede idealizar la deuda técnica y considerar que todas las semanas en las que se dedica una cantidad de tiempo a solucionarla esta deuda técnica desaparece, pero esto no corresponde con la realidad. Normalmente la deuda aunque se solucione suele ir incrementando poco a poco debido a la complejidad de un sistema mantenido a lo largo de una gran cantidad de tiempo.

Figura 3.1: Deuda técnica: Beneficiosa y perjudicial.

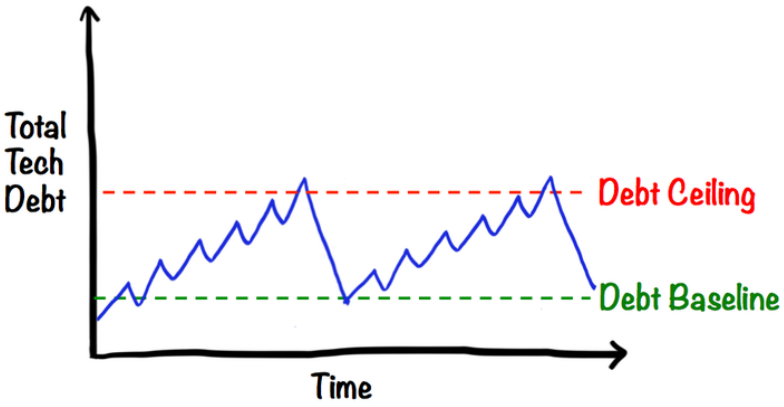

Figura 3.2: Deuda técnica: Techo y base

La manera de tener esto en cuenta es tener un 'techo de deuda' este techo debería ser lo suficientemente alto como para que no se alcance todos los meses pero no demasiado alto como para que cuando se llegue el proyecto sea directamente un fracaso o demasiado caro de pagar.

La opinión de algunas personas parece ser que se debe llegar al techo cada 6 meses, desde mi propia inexperiencia creo que deberíamos evaluar la deuda a los 4 meses para ver si la re-estructuración del proyecto es viable e intentar buscar un punto de tiempo en el que el pagar la deuda técnica sea algo más simple ya que si esperamos hasta los 6 meses probablemente no podamos elegir un momento óptimo.

Me gustaría añadir que la deuda técnica tiene otros razonamientos para tener que mantenerse baja: al que menos importancia parece darse es que la deuda técnica es algo difícilmente cuantificable hasta que se refactoriza.

3.2. Integración continua (Continuous Integration)

Normalmente acortado con las siglas CI, es un método habitual para reducir la deuda técnica. Es la práctica de ejecutar y testear conjuntamente todas los servicios que deban ir relacionados una vez se inicie el despliegue en producción. Esto nos asegura que van a funcionar en producción la mayoría de las veces, la minoría habrá problemas por la escalabilidad... Un ejemplo es si ejecutamos java y los test no fallan pero al ponerlo en producción le damos a la JVM más de 200 GB de memoria RAM, esto causa comportamientos inesperados.

Entrega continua (Continuous Delivery)

Es un término asociado a CI, consiste en un proyecto que dispone de CI y buena cantidad de test de calidad, una vez ya tengamos un sistema a punto podemos empezar a modificar o mejorar funcionalidades añadiéndolas directamente a producción si pasa los test, requiere que hagamos los test casi a la vez que el código. Esto fomenta y premia técnicas como extreme programming (XP) que se basan en TDD (Test Driven Development), una forma de programar que pide que hagamos los test antes que el resto del código.

3.3. DevOps

DevOps es un acrónimo inglés de: 'software **Dev**elopment and information technology **Op**eration**s**' es un termino que engloba un conjunto de prácticas de colaboración y comunicación entre desarrolladores software y técnicos informáticos. Los objetivos de esta comunicación y colaboración son una construcción de software más consistente y confiable. Este proceso heredero de las técnicas ágiles se basa en una *cadena de herramientas*. Esta cadena de herramientas es algo que no esta completamente definida pero más o menos la podemos concretar, cabe tener en cuenta que esta cadena cambia según a quien le preguntes.

La 'cadena de herramientas' de DevOps

Esta cadena de herramientas se basa en siete procesos con sus correspondientes herramientas:

1. Plan: Consistente en determinación de métricas, requerimientos... y una vez pasemos de la primera iteración ha de tener en cuenta el feedback del cliente.

2. Creación: Es el proceso de programar y crear el software, las herramientas en este proceso es el software de control de versiones que vayamos a usar.

3. Verificación: Proceso de comprobación de la calidad del software. Normalmente consiste en hacer test de diversos tipos (Aceptación, seguridad...)

4. Preproducción o empaquetación: En esta fase se piden aprobaciones de los distintos equipos y se configura el paquete.

5. Lanzamiento: En este punto se prepara el horario de lanzamiento y se orquesta el software para poder ponerlo en el entorno de producción objetivo.

6. Configuración: Una vez el software esta desplegado toda la parte de la infraestructura y configuración de la misma se incluye en esta categoría, como por ejemplo las bases de datos, configuración de las mismas...

7. Monitorización: Tras entregar el software se mide su rendimiento en la infraestructura objetivo y se mide la satisfacción del usuario final. Se recogen métricas y estadísticas

3.4. Microservicios

Los microservicios son una arquitectura y un patrón de diseño, que no se ha inventado en un momento dado, si no que ha surgido como una tendencia o patrón del diseño de sistemas en el mundo real.

Un microservicio es un servicio pequeño y autónomo que puede trabajar junto a otros, es importante que este centrado en hacer una cosa bien.

Esto esta reforzado por el concepto del Principio de responsabilidad única de C. Martin: "Juntar aquellas cosas que cambian por la misma razón y separar aquellas que cambian por razones diferentes."

El tamaño de un microservicio es algo muy discutido, pero generalmente se requiere que sea mantenible por un equipo 'pequeño' (6-10 personas), y se pueda reescribir en 2 semanas.

Esto se debe conseguir mediante una interfaz de programación de aplicación (API: application programming interface), que permita a lso clientes acceder al servicio sin causar acoplamiento, algo más difícil de hacer que de decir.

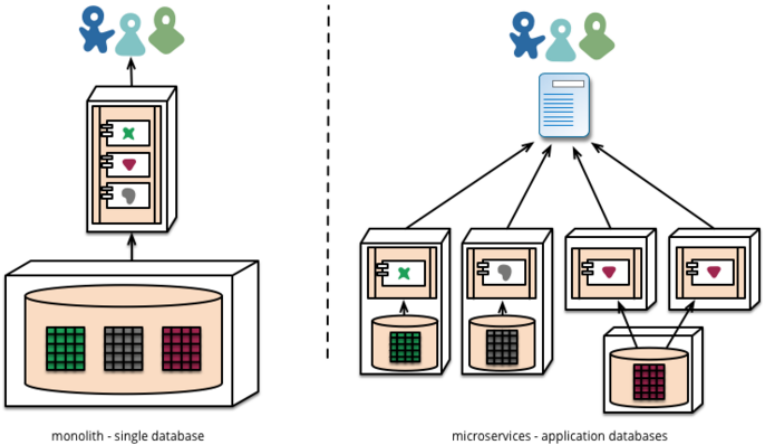

Figura 3.3: Arquitecturas monolítica y microservicios.

Los principales beneficios son:

1. Sistemas heterogéneos: Facilita usar distintas tecnologías en distintos sistemas para usar la mejor herramienta en cada ocasión.
2. Resiliencia: Si falla un microservicio el resto del sistema se puede mantener levantado, aislando los problemas y facilitando la alta disponibilidad.
3. Escalabilidad: En caso de que una parte del sistema necesite mayor número de recursos podemos crear una nueva instancia sin cambiar el funcionamiento del resto del sistema.
4. Facilidad de despliegue: Los cambios se mantienen más contenidos.
5. Replazabilidad: Un microservicio que pasa a ser obsoleto puede ser reemplazado o eliminado en vez de tener que mantenerlo por que al eliminarlo se rompe todo el sistema, algo que no debería pasar pero que es común en los sistemas legados.
6. Test a nivel de servicio: Cada microservicio se puede testear por su cuenta propia de manera que aunque el sistema aumenten su complejidad podamos controlar cada microservicio por separado.

Estos beneficios por supuesto vienen con desventajas:

1. Distribución: Raramente se van a ver microservicios en una arquitectura que no este distribuida, esto hace que sea más difícil de programar y tengamos que tener en cuenta otros errores.

2. Seguridad: Es más difícil asegurar la seguridad de un microservicio ya que debemos protegernos contra más puntos de vulnerabilidad, a no ser que estemos ejecutando en un clúster seguro.

3. Complejidad de operación: Hace falta un equipo de operaciones con madurez de manera que se puedan redistribuir los microsistemas regularmente.

4. Test a nivel de sistema: Los tests a nivel de sistema incrementan algo su complejidad ya que vamos a tener que usar microservicios falsos como mocks o similares casi siempre a no ser que estemos en un sistema pequeño.

Por ultimo vale la pena hablar del coste en productividad, esto no es una ventaja ni un perjuicio, es un trade-off, un intercambio.

En un sistema monolítico tenemos menos coste en productividad hasta que llegamos a un punto crítico en el que la complejidad aumenta demasiado o necesitamos escalar.

En los microservicios tenemos menos coste a la hora de seguir iterando aumentando la complejidad de un sistema pero a cambio tenemos mayor barrera de entrada hasta tener el primer prototipo funcional y los desarrolladores tienen que aprender cómo programar bajo esta arquitectura.

Técnicas y herramientas

En esta parte de la memoria se detallan tanto técnicas y herramientas como metodologías y librerías usadas.

4.1. Metodología

Al no haber una metodología concreta que se adapte al trabajo de una sola persona se ha buscado un conjunto de metodologías que se pudiesen conjuntar para dar una metodología conjunta más aplicable a la situación particular que se tiene.

De **DevOps** se ha extraído la toolchain (cadena de herramientas) para minimizar riesgos totales del proyecto y se ha buscado reducir la deuda técnica nacida de la necesidad de desplegar el producto y no estar preparado para ello, algo que resulta ser más habitual de lo que debería ser.

De **Scrum** recogemos las herramientas como los sprint y las reuniones cada sprint tanto para concretar el siguiente sprint como para hacer una retrospectiva del anterior.

Se ha estudiado también **extreme programming** (XP) y se ha visto que no es viable para el proyecto por la cantidad de esfuerzo que requiere para una sola persona hacer el trabajo de dos como es el **pair programming**.

De todos los métodos ágiles se ha adquirido la mentalidad de no intentar planificar todo desde el principio e intentar planear las cosas con el conocimiento adquirido en cada iteración. Se intenta decidir en todo lo más tarde posible para intentar conseguir el máximo conocimiento posible para mejorar las decisiones.

Se ha decidido que la recopilación de información de los sprints y este tipo de documentación se va a hacer en un tablero **kanban**, la herramienta que nos proporciona este sistema va a ser **Zenhub**, se usa por la facilidad y el nivel de integración que nos da con otros sistemas también elegidos.

4.2. Patrones de diseño

Se ha intentado usar los patrones de diseño para conseguir una arquitectura mejor y más simple para nuevos desarrolladores. También hay que tener en cuenta que no todos los patrones tienen sentido en un lenguaje de programación como python por su naturaleza dinámica.

Model-View-Controller

El patrón modelo vista controlador nos ayuda a simplificar mucho la estructura de los archivos de una página web o aplicación que dependa de vistas. Esto es especialmente beneficioso cuando se introduzca a un miembro nuevo al proyecto ya que permite conocer rápidamente como funciona mucha cantidad de código, de manera que si necesitase realizar cambios lo tuviese fácil.

4.3. Control de versiones

El control de versiones es una necesidad hoy en día tanto en sistemas compuestos de uno o múltiples desarrolladores, esto nos permite poder dejar ciertas features que empecemos a implementar y se dificulten sin terminar para seguir con otras que den más valor al cliente y luego continuar con las dificultades o dejarlas de lado según beneficie al cliente.

Este solo es un ejemplo de las ventajas entre las que se cuentan: recuperación de versiones estables, visualización de los cambios que han dado resultado a un bug...

El sistema de hosting lo usaremos para facilitar el acceso a la información contenida en el sistema, también, al ser externo (no estar en el mismo ordenador que se use) nos servirá de sistema de copias de seguridad.

Usaremos **git** por razones de documentación y conocimiento ya adquirido, aunque existen otros sistemas como subversion.

Las alternativas principales de sistema de hosteo de git son: Bitbucket, Github, Gitlab.

Se ha elegido Github por una mezcla de razones históricas con razones de integración con otros sistemas como el sistema kanban (Zenhub).

Históricamente se conocía Github y se sabe que los proyectos open source no tienen ninguna limitación, la fácil integración con otros sistemas como **travis** y **slack** si hiciese falta.

4.4. Integración continua

La integración continua es el método por el cuál intentamos integrar todos nuestros productos continuamente para ver si funcionan correctamente en conjunto. Nos va a ayudar a minimizar el riesgo de fallo del proyecto sobre todo si se mantiene en desarrollo un largo periodo de tiempo.

La herramienta que se va a usar es **travis**, esto se debe a la gran documentación, facilidad de integración con git y gihub, y otras integraciones que nos ayudaran con otras secciones. Otra ventaja es que al ejecutar tu build y tests en sus servidores no te tienes que preocupar de casi nada.

Otras opciones que se podrían usar son Jenkins que como ventaja es Open Source y tiene gran cantidad de plugins.

4.5. Quality Assurance

Dado que no tenemos departamento de QA como deberíamos para llevar una toolchain como la especificada en DevOps usaremos herramientas automatizadas, que aunque no tengan la calidad de una revisión humana es lo mejor que disponemos.

La herramienta que se ha elegido es **CodeClimate**, se ha usado por que no ha sido demasiado complicado el incluir esta herramienta en la linea de producción que ya teníamos (Github+travis). Se ha intentado usar SonarQube pero no ha sido tan sencillo.(Se intento introducir dentro de la toolchain automática, no manualmente).

4.6. Tests

Los tests se han realizado con **pytest**. Esta herramienta es muy parecida a unittest (la herramienta de tests en la biblioteca standard de python). Tiene ventajas, facilita el debug al decirte exactamente que ha fallado y con que valores, y los métodos como setup y teardown. Se integró esta herramienta con travis.

Otras librerías pueden ser unittest o nose2.

Recubrimiento

La forma más sencilla para ver el recubrimiento de un test en Python es coverage, tiene varias opciones como report para salida en consola o html para un html con el cubrimiento bien señalado. Se consiguió integrar esta herramienta con travis, pytest y CodeClimate de manera que se ejecuta con la integración continua.

4.7. Dependencias

Para el control de dependencias (seguridad, últimas versiones y licencias) se ha usado VersionEye, integrado con Github, esta herramienta se adhiere mediante un webhook a Github y nos dice si se ha descubierto alguna brecha de seguridad en nuestras herramientas, cuales son sus últimas versiones y si las estamos usando y si las licencias del proyecto son compatibles con la que tenemos.

4.8. Comunicación

La comunicación se ha hecho de diferentes maneras: Email, Slack y de manera física. Email se ha usado para casi toda la comunicación a distancia. Slack se ha usado para integrar herramientas y notificar sobre estas. La comunicación física es el medio que más se ha usado, esto no se debe a ningún motivo en particular simplemente ha sido el medio más natural para todos.

4.9. Documentación

La documentación se ha basado en dos sistemas: en código y fuera de código.

En código

En el código se ha usado la documentación recomendada por la comunidad de python en el PEP 287, este recomienda usar **reStructuredText** (rst).

Se han seguido las guidelines (guías) de pocoo, esto es para poder usar Sphinx (generador de documentación) para transformar esas cadenas de documentación en una documentación tanto en html o en Latex si hiciese falta.

Fuera de código

Para la documentación que se esta leyendo se ha usado la plantilla oficial de la ubu para crear documentos Latex aunque se ha considerado usar rst al igual que en el código ya que si hiciese falta se podría transformar a Latex o a html siendo mucho más flexible. No se ha hecho por que no existe una plantilla y la barrera de entrada es algo alta.

4.10. Editor de texto

Se ha mirado tanto en editores de texto más simples como vim, atom, sublime... Estos cuentan con suficientes plugins que acaban siendo una IDE.

Las IDEs también se han mirado y se han considerado tanto Spyder, Lyclipse como Pycharm.

De todos estos no hay ventajas en usar unos u otros ya que todos acaban teniendo la misma capacidad gracias a plugins, se ha decidido usar Pycharm por que era la única IDE que todavía no se había probado.

4.11. Bibliotecas

Se ha usado Flask como microframework ya que sirve tanto para implementar el patrón MVC como para crear un sistema altamente escalable.

Para facilitar ciertas funcionalidades como control de usuarios, seguridad... se han usado las extensiones: flask-babel y babel(internacionalización), flask-login (control de usuarios), flask-sqlalchemy y psycopg2(acceso a la base de datos), flask-wtf y WTForms (Formularios html) y flask-bcrypt y Bcrypt para seguridad.

Como podemos ver muchas de estas extensiones de flask tienen el mismo nombre pero con flask añadido, esto se debe a que son wrappers de la librería en cuestión pero facilitados para inicializarse con flask (generalmente añaden un constructor y un constructor lazy).

Se ha usado el modulo paramiko para simplificar el uso de SSH.

Bibliografía