



UNIVERSIDAD DE

VALLADOLID

E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS ESPECÍFICAS DE
TELECOMUNICACIÓN MENCIÓN EN INGENIERÍA TELEMÁTICA

Implementación de un Sistema de Desarrollo de Aplicaciones en Contenedores *Docker* y su Automatización a través de un Bot de *Slack*

Autor:

D. Eloy Redondo de Álvaro

Tutora:

Dña. Míriam Antón Rodríguez

Valladolid, 14 de Julio de 2017

TÍTULO: **Implementación de un Sistema de Desarrollo de Aplicaciones en Contenedores *Docker* y su Automatización a través de un Bot de *Slack***

AUTOR: **D. Eloy Redondo de Álvaro**

TUTORA: **Dña. Míriam Antón Rodríguez**

DEPARTAMENTO: **Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTA: **Dña. Míriam Antón Rodríguez**

VOCAL: **D. Mario Martínez Zarzuela**

SECRETARIO **D. David González Ortega**

SUPLENTE **D. Francisco Javier Díaz Pernas**

SUPLENTE **Dña. M^a Ángeles Pérez Juárez**

FECHA: **21/07/2017**

CALIFICACIÓN:

RESUMEN

Este trabajo trata del desarrollo y la implementación de un nuevo sistema de automatización para el desarrollo de aplicaciones en la empresa *MADISON Experience Marketing* en la que me encuentro realizando una beca.

El sistema de automatización empleará un sistema de desarrollo de aplicaciones en contenedores *Docker*. Gracias a este sistema, el desarrollo por parte de los distintos equipos de la empresa, podrá realizarse de una forma más ágil e independiente.

El sistema consistirá en tres elementos diferentes que se comunicarán entre ellos. El primero de estos elementos consiste en una *API REST*, encargada de la gestión de los nuevos contenedores *Docker* y de la realización de distintas operaciones comúnmente solicitadas por los desarrolladores, como, por ejemplo, creaciones de bases de datos. Otro de los elementos, consiste en un bot de *Slack* que recogerá distintos comandos introducidos por los usuarios y lanzará la ejecución de las tareas correspondientes en la *API REST*. Por último, el tercer elemento se trata de un servidor *Jenkins* que se utilizará para monitorizar y dejar constancia de la ejecución de las tareas lanzadas por los usuarios.

PALABRAS CLAVE

Automatización, contenedor, *Docker*, *Jenkins*, bot, *Slack*, *API*, *REST*.

ABSTRACT

This work deals with the development and implementation of a new automation system for the development of applications in the company *MADISON Experience Marketing* in which I find myself making a scholarship.

The automation system employs a Docker container application development system. Thanks to this system, the development can be carried out in a more agile and independent way by the different teams of the company.

The system will consist of three different elements that will communicate with each other. The first of these elements consists of a *REST API*, responsible for the management of the new *Docker* containers and the realization of different operations commonly requested by the developers, such as database creations. Another element consists of a *Slack* bot that will pick up different commands entered by the users and launch the execution of the corresponding tasks in the *REST API*. Finally, the third element is a *Jenkins* server that will be used to monitor and record the execution of the tasks launched by the users.

KEYWORDS

Automation, container, *Docker*, *Jenkins*, bot, *Slack*, *API*, *REST*.

AGRADECIMIENTOS

En primer lugar, tras todos los largos años juntos, quiero agradecer a todos mis amigos de la carrera el haber estado ahí conmigo, peleando tantas veces para conseguir superar esas difíciles asignaturas. Pero, sobre todo, por haber estado conmigo disfrutando de tantos momentos de compañía y ayuda mutua, forjando una amistad que dudará para siempre.

Tampoco puedo olvidarme de todos los compañeros de trabajo de *MADISON*, quienes, de mejor o peor forma, me aguantan día a día, y, durante todo el tiempo que hemos pasado juntos, me han hecho sentir como entre amigos de toda la vida.

En especial, quiero dar las gracias a Luis Miguel Sáez, quien ha sido mi compañero más cercano durante la realización de este TFG, por todo lo aprendido y de quien espero seguir aprendiendo, aunque tan sólo sea una mínima parte de su gran conocimiento.

Uno de mis mayores agradecimientos tiene que ir, sin duda, para mi jefe Iván Casado, una de las principales personas por las que este TFG ha podido siquiera existir. Aparte de sus grandes conocimientos, el talento para el liderazgo que demuestra día tras día, nunca ha dejado, ni dejará de sorprenderme. Sin duda es todo un referente para cualquiera que haya tenido la oportunidad de trabajar a su lado, yo incluido.

También tengo que agradecer, sin ninguna duda, todo el apoyo que he recibido por parte de Dña. Míriam Antón, mi tutora en la realización de este TFG, sobre todo por haberme brindado la posibilidad de llevarlo a cabo. No resulta sorprendente que la gran mayoría de los alumnos que han tenido la suerte de tener a Míriam como profesora, conserven un grato recuerdo imborrable, ya que, sin duda, en una de las mejores profesoras de toda la Escuela.

Por último, pero, como suele decirse, no menos importante, es difícil agradecer lo suficiente a toda mi familia, el haber estado ahí, apoyándome siempre y haciéndome sentir orgulloso de cada pequeño paso que consigo dar hacia delante. Por todo esto, no puedo sentirme más agradecido hacia mi abuela Alejandra, mi tía Ana, su pareja Juan Manuel y, en especial, a aquellos que conviven conmigo, mi hermana Iris y mi padre José Luis, quien, junto a mi madre María del Carmen, quien, por desgracia, ya no está con nosotros, han hecho de mí la persona que soy hoy.

Sin ninguna duda, llegue a donde llegue en un futuro sabré que todo ha sido gracias a todas las personas que han estado ahí para apoyarme incondicionalmente. A todos ellos, GRACIAS.

ÍNDICE DE CONTENIDOS

CAPÍTULO 1. INTRODUCCIÓN	17
1.1. Introducción	17
1.2. Motivación y objetivos	17
1.3. Estructura	18
CAPÍTULO 2. CONCEPTOS PREVIOS	21
2.1. Entornos de desarrollo	21
2.2. Contenedores Docker	21
2.3. Servicios de contenedores	24
2.3.1. Introducción	24
2.3.2. Swarm	25
2.3.3. AWS	26
CAPÍTULO 3. DISEÑO DEL SISTEMA	29
3.1. Componentes	29
CAPÍTULO 4. TECNOLOGÍAS UTILIZADAS	35
4.1. Introducción	35
4.2. API REST	35
4.3. Bot de Slack	37
4.4. Jenkins	40
CAPÍTULO 5. DESARROLLO DEL SISTEMA	45
5.1. Introducción	45
5.2. Descripción técnica	46
5.2.1. Introducción	46
5.2.2. Creación de aplicaciones	53
5.2.3. Operaciones con contenedores	55
5.2.4. Subidas de nuevas versiones de una aplicación	56
5.2.5. Creación de bases de datos	56
5.2.6. Operaciones en bases de datos	57
5.2.7. Comandos de solicitud de información	57
5.2.8. Comunicaciones generales	58
5.3. Manual de usuario	62
5.3.1. Creación de contenedores	62
5.3.2. Arranque de contenedores	65
5.3.3. Eliminación de contenedores	65
5.3.4. Ejecutar comandos en los contenedores	65

5.3.5. Clonación de contenedores	65
5.3.6. Subidas a preproducción y producción	66
5.3.7. Creación de bases de datos	67
5.3.8. Importación de bases de datos	67
5.3.9. Ejecución de ficheros sql	67
5.3.10. Listado completo de comandos	68
5.4. Pruebas y ejemplos	71
5.4.1. Introducción	71
5.4.2. Creación de un contenedor de desarrollo	71
5.4.3. Arranque de un contenedor	74
5.4.4. Creación de un servicio en preproducción	76
5.4.5. Redespigüe de aplicaciones	79
5.4.6. Información sobre los servicios	81
5.4.7. Ejecución de comandos en un contenedor	83
5.4.8. Creación de bases de datos	85
5.4.9. Importar bases de datos	86
CAPÍTULO 6. CONCLUSIONES Y LÍNEAS FUTURAS	91
6.1. Conclusiones	91
6.2. Líneas futuras	91
REFERENCIAS	93

ÍNDICE DE ILUSTRACIONES

Figura 1: Esquema del sistema de automatización. Fuente: Propia _____	18
Figura 2: Comparación de la estructura de un contenedor Docker y una máquina virtual. Fuente: [4] _____	22
Figura 3: Ejemplo de Dockerfile. Fuente: Propia _____	23
Figura 4: Ejemplos de la estructura de distintos contenedores. Fuente: [7] _____	24
Figura 5: Diferencia entre la estructura de un contenedor Docker y un servicio Swarm a través de una representación empleando el logo de Docker. Fuente: [3] _____	25
Figura 6: Esquema del sistema de automatización. Fuente: Propia _____	29
Figura 7: Sistema de automatización con diferentes elementos encargados de recopilar los datos de entrada. Fuente: Propia _____	30
Figura 8: Interfaz de SwaggerUI. Fuente: Propia _____	31
Figura 9: Estructura detallada en SwaggerUI de un servicio web. Fuente: Propia _____	32
Figura 10: Ejemplo de servicio web Slim de PHP. Fuente: Propia _____	35
Figura 11: Ejemplo de implementación de un comando en el bot de Slack. Fuente: Propia _____	37
Figura 12: Interfaz de Slack. Fuente: Propia _____	38
Figura 13: Pantalla de configuración de un bot de Slack. Fuente: Propia _____	39
Figura 14: Inicialización del proceso Nodejs del bot de Slack. Fuente: Propia _____	40
Figura 15: Interfaz principal del servidor Jenkins implementado. Fuente: Propia _____	41
Figura 16: Menú de gestión de plugins del servidor Jenkins. Fuente: Propia _____	42
Figura 17: Esquema de las comunicaciones que se realizan a través del sistema de automatización. Fuente: Propia _____	45
Figura 18: Ejemplo de configuración de una tarea en Jenkins (1). Fuente: Propia _____	48
Figura 19: Ejemplo de configuración de una tarea en Jenkins (2). Fuente: Propia _____	50
Figura 20: Ejemplo de configuración de una tarea en Jenkins (3). Fuente: Propia _____	51
Figura 21: Ejemplo de llamada para lanzar una tarea en Jenkins. Fuente: Propia _____	52
Figura 22: Ejemplo de función del bot que devuelve un mensaje en un formato JSON reconocible por Slack. Fuente: Propia _____	58
Figura 23: Servicio web que se encarga de transmitir al bot de Slack el tiempo estimado de una tarea. Fuente: Propia _____	60
Figura 24: Servicio web que se encarga de transmitir al bot de Slack el aviso de la finalización de una tarea. Fuente: Propia _____	61
Figura 25: Servicio web del bot de Slack para comunicar mensajes a un chat determinado. Fuente: Propia _____	61
Figura 26: Chat grupal del historial de tareas. Fuente: Propia _____	62
Figura 27: Ejemplo del uso del comando “crear contenedor” (1). Fuente: Propia _____	72
Figura 28: Ejemplo del uso del comando “crear contenedor” (2). Fuente: Propia _____	73
Figura 29: Ejemplo del uso del comando “mostrar contenedores”. Fuente: Propia _____	74
Figura 30: Ejemplo del uso del comando “arrancar contenedor”. Fuente: Propia _____	75
Figura 31: Ejemplo del uso del comando “mostrar contenedores” para listar los contenedores de desarrollo. Fuente: Propia _____	76
Figura 32: Ejemplo del uso del comando “crear contenedor pre”. Fuente: Propia _____	77
Figura 33: Ejemplo del uso del comando “mostrar contenedores” para listar los contenedores de varios entornos. Fuente: Propia _____	78

Figura 34: Ejemplo del uso del comando “redespargar aplicacion” para subir una nueva versión. Fuente: Propia _____	79
Figura 35: Ejemplo del uso del comando “redespargar aplicacion" para volver a una versión anterior. Fuente: Propia _____	80
Figura 36: Ejemplo del uso del comando “mostrar info app”. Fuente: Propia _____	82
Figura 37: Ejemplo de uso del comando “ejecutar comando”. Fuente: Propia _____	84
Figura 38: Ejemplo del uso del comando “crear db”. Fuente: Propia _____	85
Figura 39: Ejemplo del uso del comando “importar db”. Fuente: Propia _____	86
Figura 40: Ejemplo del uso del comando “mostrar tarea” para comprobar el estado de ejecución de una tarea. Fuente: Propia _____	87

CAPÍTULO 1

INTRODUCCIÓN

CAPÍTULO 1. INTRODUCCIÓN

1.1. Introducción

Este TFG tratará del trabajo que he realizado a lo largo del curso lectivo 2016/2017 como becario en la empresa *MADISON Experience Marketing* [1] y, más concretamente, en su departamento de Infraestructura y Telecomunicaciones.

Este departamento es el encargado de gran variedad de funciones dentro de la empresa, que pueden ir desde proporcionar un puesto de trabajo a un nuevo empleado, hasta asegurarse del correcto funcionamiento de cada uno de los servidores existentes. Esto hace que cada uno de los compañeros del departamento tengan asignadas unas responsabilidades determinadas, pero sin dejar de lado la comunicación y cooperación para la mayoría de estas.

MADISON Experience Marketing ofrece distintos tipos de servicios a sus clientes. Debido al trabajo que he llevado a cabo, el servicio en el que me centraré es el del desarrollo de aplicaciones web. Este servicio es llevado a cabo por distintos equipos de desarrolladores pertenecientes a varios departamentos de la empresa.

Desde un inicio mi trabajo ha sido el de proporcionar a estos grupos de desarrolladores una nueva metodología que les permita llevar a cabo su trabajo de una forma más cómoda e independiente, gracias al desarrollo de un bot de *Slack*.

Slack [2] es una herramienta de comunicación para grupos, que cuenta con cliente *web*, de escritorio y de dispositivos móviles. La gran ventaja de esta herramienta reside en una integración total con multitud de herramientas del ámbito profesional.

1.2. Motivación y objetivos

Este bot permitirá a los desarrolladores de los distintos departamentos de la empresa, realizar tareas que, de una forma tradicional, necesitarían solicitar a un miembro del departamento de Infraestructura y Telecomunicaciones, además de tener que esperar a que su petición fuese atendida.

De esta forma se consiguen tres beneficios a tener en cuenta:

1. Acelerar el desarrollo de aplicaciones, gracias a la independencia que se les da a los desarrolladores a la hora de realizar tareas recurrentes y necesarias para avanzar en su trabajo diario.
2. Descongestionar el departamento de Infraestructura y Telecomunicaciones al liberarle de la realización de dichas tareas, permitiendo así el trabajo en nuevos proyectos que aporten innovación y mejoren la calidad de trabajo.
3. Estandarización de los métodos de desarrollo, al hacer que todos los departamentos vayan migrando el desarrollo de sus aplicaciones a un nuevo sistema más centralizado y controlado.

Este nuevo sistema que se estandarizará para todos los equipos, tendrá su base en la utilización de contenedores *Docker* [3] para el desarrollo de aplicaciones, cuyas diferencias respecto al desarrollo de forma convencional se explican en el siguiente capítulo.

El funcionamiento del bot de *Slack* se basará en la realización de llamadas a una *API REST*, la cual se encargará de realizar las tareas solicitadas por los desarrolladores de la empresa a través del bot de *Slack*. Además, las tareas solicitadas serán controladas y almacenadas gracias a un servidor *Jenkins*. Por lo tanto, este TFG tratará sobre el desarrollo e implementación de estos tres componentes, que juntos forman el sistema de automatización.

En la siguiente figura puede observarse un esquema muy general de la comunicación entre las distintas partes que forman parte del sistema.

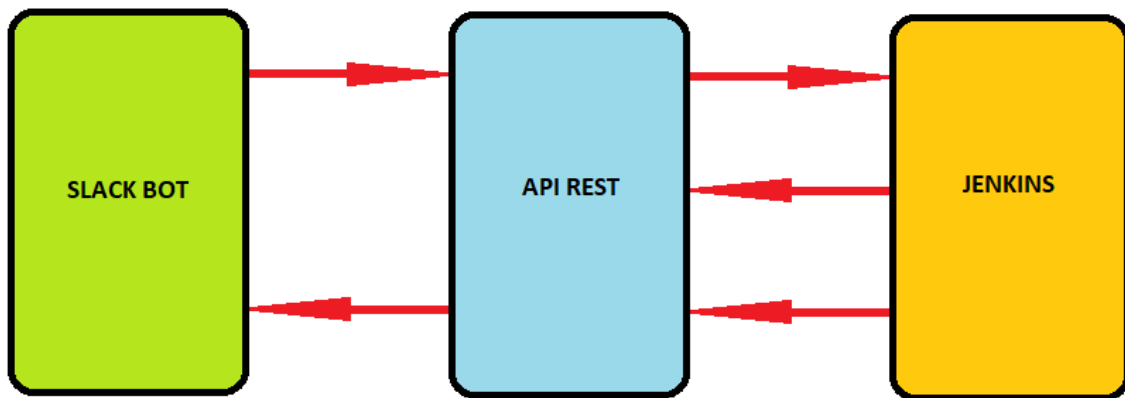


Figura 1: Esquema del sistema de automatización. Fuente: Propia

A lo largo de este documento se explicarán de forma más detallada, las distintas comunicaciones entre los elementos del sistema, además de profundizar en las características de dichos elementos.

1.3. Estructura

A continuación, se detalla la estructura de la que consta este documento. En el capítulo 2, se introducirán los conceptos previos a tener en cuenta de la infraestructura sobre la que funcionará el sistema de automatización, como la metodología a seguir durante el proceso de desarrollo en la empresa o el concepto de contenedores *Docker*. El capítulo 3 se centra en la arquitectura general del sistema y las decisiones de diseño que ha conllevado. En el capítulo 4, se detallarán las distintas tecnologías empleadas para el desarrollo del sistema de automatización junto con las razones de la elección de las mismas respecto a otras posibles alternativas. El capítulo 5 tratará sobre el funcionamiento de todo el sistema en conjunto, incluyendo el proceso llevado a cabo para su desarrollo. También se incluye el manual de usuario para la realización de las distintas tareas y ejemplos de funcionamiento del sistema. Finalmente, el capítulo 6 corresponde con las conclusiones finales del TFG junto con líneas futuras a tener en cuenta.

CAPÍTULO 2

CONCEPTOS PREVIOS

CAPÍTULO 2. CONCEPTOS PREVIOS

2.1. Entornos de desarrollo

El proceso que sigue una aplicación desde que se empieza a desarrollar hasta que está funcionando y accesible al cliente, consta de tres pasos, divididos en tres entornos de desarrollo separados. Estos son el entorno de desarrollo, el de preproducción y el de producción.

Cada uno de estos entornos tiene unas necesidades que el sistema basado en contenedores *Docker* que se implementará, debe atender de forma adecuada:

- Entorno de desarrollo: En este entorno, la aplicación será modificada constantemente, por lo que deben permitirse realizar cambios instantáneos que puedan verse reflejados lo más rápido posible.
- Entorno de preproducción: Este entorno se utiliza para realizar pruebas sobre las nuevas funcionalidades que se van implementado en la aplicación. Las modificaciones en este entorno no son tan comunes y deben ser una copia casi exacta de lo que sería la aplicación final. En este entorno también se realizan distintas pruebas de rendimiento a la aplicación, para determinar los requisitos de *hardware* que esta puede tener.
- Entorno de producción: Es el paso final del proceso. En este entorno, la aplicación ya está siendo usada por el cliente, por lo que los cambios en ella son muy poco comunes.

Como se muestra, el nuevo sistema debe presentar distintas características en función del entorno. Por ejemplo, para el entorno de producción es vital proporcionar un sistema cuanto más resistente a fallos imprevistos mejor. Por lo tanto, es primordial la existencia de sistemas de respaldo, mientras que, por ejemplo, en el entorno de desarrollo es más importante priorizar la velocidad a la hora de realizar cambios.

2.2. Contenedores *Docker*

Uno de los principales problemas a los que un administrador de sistemas suele enfrentarse, es el de la gestión de las versiones de *software* distribuidas en las máquinas de la empresa, ya sean puestos de trabajo o servidores.

Debido a esto, es muy poco aconsejable que los empleados utilicen como entorno de desarrollo su propia máquina local, ya que supondría tener instaladas las versiones del *software* que el desarrollador necesite en cada momento. Además, teniendo en cuenta que estos suelen, casi con toda seguridad, trabajar en grupos y no en solitario, esto supondría el tener que asegurarse que todos los miembros del grupo poseen y mantienen las mismas versiones de todo el *software* en cada momento, para evitar futuros problemas.

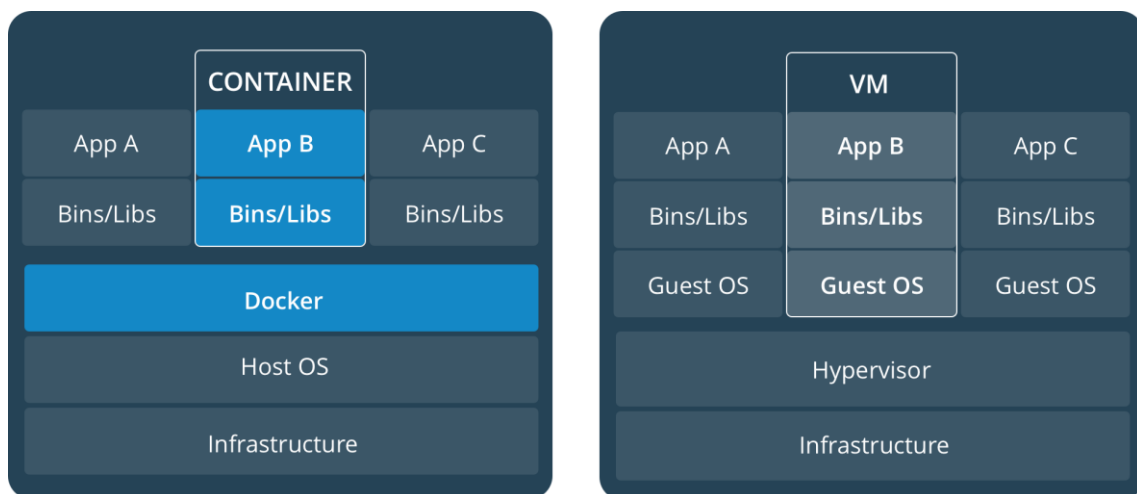
Esto sería muy complicado de controlar, además de que supondría una carga de trabajo adicional para mi departamento, debido al tiempo que se invertiría en solucionar los problemas que puedan surgir debido a que algún empleado haya cambiado algo en su máquina local.

Por eso mismo, es preferible que los desarrolladores trabajen a través de máquinas virtuales sobre servidores de la empresa. Esto evita en gran medida los problemas que pudiesen surgir de trabajar en su máquina local, ya que pueden crearse máquinas virtuales iguales para cada miembro del grupo de desarrollo.

En este punto es donde la inclusión de contenedores *Docker* [4], supone una especie de evolución respecto a las máquinas virtuales. Ambos sistemas tienen beneficios similares respecto a lo que el aislamiento y asignación de recursos se refiere, pero internamente funcionan de forma diferente. Las máquinas virtuales son una abstracción del *hardware* del servidor donde están construidas, convirtiendo un servidor en varios a través de un hipervisor. Cada una de las máquinas virtuales que se creen sobre un servidor incluirán una copia completa de un sistema operativo, aplicaciones, ficheros binarios y librerías, terminando por ocupar varios *gigabytes* de almacenamiento.

Los contenedores por su parte, son una abstracción en la capa de aplicación, que empaqueta código y dependencias juntas [5]. Además, cuando varios contenedores están creados en una misma máquina, estos pueden compartir el sistema operativo del *kernel*, corriendo cada uno de ellos como procesos aislados. Esto hace que, si creamos varios contenedores iguales, sólo uno de ellos ocupa espacio en el servidor, ocupando además cantidades del orden de *megabytes* en lugar de *gigabytes*.

Con las siguientes imágenes podemos ver una comparación entre estos dos sistemas.



*Figura 2: Comparación de la estructura de un contenedor Docker y una máquina virtual.
Fuente: [4]*

Como puede verse, las máquinas virtuales necesitan instalar todo un sistema operativo que se comunica con el del servidor a través del hipervisor que virtualiza el *hardware* de la máquina. Por el contrario, los contenedores son procesos *Docker* dentro del propio sistema operativo de la máquina, que aíslan la aplicación junto con sus dependencias.

Una vez vistas las diferencias entre máquinas virtuales y contenedores, se puede concluir que, debido a la gran diferencia de tamaño, los contenedores son mucho más manejables. Otras ventajas pueden ser, por ejemplo, el tiempo de arranque. Mientras

que los contenedores son capaces de arrancar de forma instantánea, las máquinas virtuales no. Otras ventajas son la escalabilidad y el control sobre las versiones de software que se irá viendo a continuación.

Los contenedores *Docker* se crean a partir de imágenes, las cuales se crean a partir de ficheros de configuración llamados *Dockerfiles*. En los *Dockerfiles* se describe todo el software que encapsularán los contenedores. Estos ficheros pueden heredar unos de otros de forma que puede crearse una imagen a partir de un *Dockerfile* que incluya distintos módulos de *PHP* y que herede de otra imagen que, a su vez incluya otros módulos. Un ejemplo de *Dockerfile* puede verse a continuación.

```
1 FROM registry.madisonmk.local:5000/apache24
2
3 RUN apk update
4     && apk add php5-mssql php5-iconv php5-gd php5-pdo php5-pdo_mysql php5-xcache php5-mcrypt ...
5         php5-imap php5-phpmailer php5-dom php5-ctype --no-cache
6
7 #Config PHP
8 RUN sed -i -e 's/^date.timezone.*/date.timezone = CET/g'
9         -e 's/^(max_execution_time = \).*/\1120/g'
10        -e 's/^(memory_limit = \).*/\164M/g'
11        -e 's/^(^extension=.*.dll.*\)$;/\1/g'
12        -e 's/^(max_execution_time = \).*/\160/g'
13        -e 's/^(expose_php = \).*/\10ff/g'
14        /etc/php5/php.ini
15
16 #Composer install
17 RUN php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
18     && php -d memory_limit=16M composer-setup.php --install-dir=/usr/bin --filename=composer
19     && php -r "unlink('composer-setup.php');"
```

Figura 3: Ejemplo de *Dockerfile*. Fuente: Propia

Los *Dockerfiles*, por defecto, buscan las imágenes de las que heredan en el *Docker Hub* [6], que es el repositorio oficial de imágenes *Docker*. En el ejemplo del anexo, se puede ver que el *Dockerfile* hereda de otro que no se encuentra en el *Docker Hub*, sino en un registro privado, que se ha configurado para almacenar las imágenes que se utilizarán más a menudo. De esta forma no hace falta descargar estas dependencias cada vez que se quiere modificar una imagen, sino que basta con traerlas desde este registro interno de la empresa.

Para el caso de la implementación de nuestro sistema de desarrollo, las imágenes han sido creadas, en la medida de lo posible, incluyendo como sistema operativo, *Alpine Linux* [7]. Esta es una distribución de *Linux* muy ligera, obteniendo imágenes que consiguen llegar a ocupar unas pocas decenas de *megabytes*.

Como vemos en la siguiente imagen, los contenedores que pueden construirse son muy variados y dependen de nuestras necesidades.

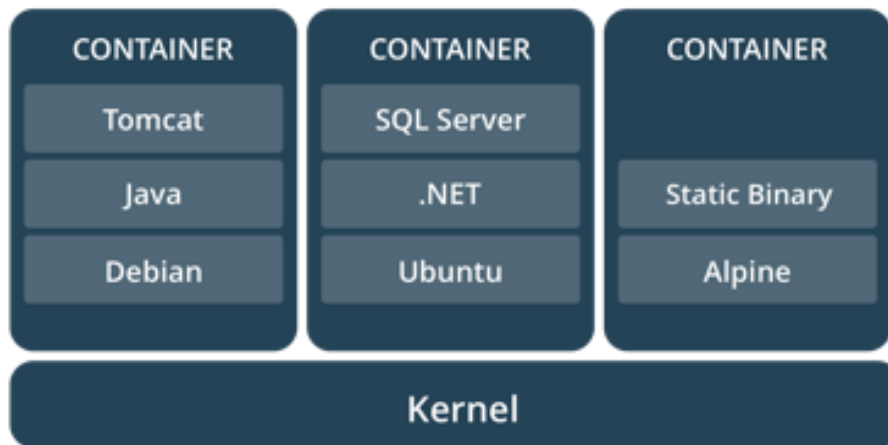


Figura 4: Ejemplos de la estructura de distintos contenedores. Fuente: [5]

Por regla general, los ejemplos que se utilizan en este documento utilizarán contenedores creados sobre una imagen que contiene *Apache 2.4* y *PHP 5.6* entre otros módulos de utilidad para los desarrolladores. Todo esto sobre un *Alpine Linux*. Por lo tanto, los contenedores creados a partir de esta imagen serán servicios *web*. El motivo de tomar esta imagen como ejemplo es que muchos de los desarrolladores de la empresa crean aplicaciones *web* con estos requisitos, por lo que es la imagen más usada.

2.3. Servicios de contenedores

2.3.1. Introducción

Como se ha visto previamente, los distintos entornos del proceso de desarrollo de una aplicación, tiene diferentes necesidades y vimos que en los entornos de preproducción y producción a diferencia del de desarrollo, era importante conseguir una mayor robustez del sistema frente a fallos, ya que es de vital importancia mantener las aplicaciones en funcionamiento de cara a los clientes. Por eso mismo, no es posible emplear un solo servidor *Docker* para cada uno de estos entornos, ya que dependeríamos de que este no sufra ningún fallo para que las aplicaciones se mantengan en funcionamiento.

En nuestro caso se decidió crear unos entornos de preproducción y producción locales, empleando el hardware disponible en la empresa, y otros en la nube, a través de los sistemas que ofrece *Amazon Web Services* [8].

El motivo tras esta decisión es la gran diferencia de necesidades que tienen las distintas aplicaciones que son desarrolladas en la empresa. Por ejemplo, si una aplicación concreta necesita ser accesible desde fuera de la red de la empresa, es preferible crearla en la nube de *Amazon Web Services* ya que la configuración del *firewall* y resolución *DNS* se simplifican bastante.

Por otro lado, si una aplicación necesita acceder a recursos locales de la empresa, es preferible mantenerla en los servidores propios en lugar de subirla a la nube, de nuevo por motivos de facilidad en la configuración y la seguridad.

De todas formas, por supuesto no existen dos tipos concretos de aplicaciones, sino que es posible que algunas tengan ambos requisitos. En estos casos, el sistema que se utilizará dependerá del caso concreto de cada aplicación, siempre tratando de priorizar la utilización de los servicios de *Amazon Web Services* a los locales, debido a que son servicios más especializados y preparados. Además, esto permite que, en un futuro, las necesidades de *hardware* local vayan disminuyendo o, al menos, no aumentando, lo que supondría un ahorro a la empresa. Este ahorro ya no sería sólo respecto al dinero invertido en *hardware*, sino también al coste de mantenimiento que ello conlleva, espacio, etc.

2.3.2. Swarm

Para conseguir un sistema escalable y resistente a fallos local basado en contenedores en donde implementar los entornos de preproducción y producción, se recurrió a *Swarm* [9].

Swarm es un clúster de nodos *Docker* donde pueden desplegarse servicios. Este clúster está formado por nodos trabajadores (*workers*) y un nodo administrador (*manager*), que también puede ser a su vez trabajador.

Los servicios son la parte fundamental del funcionamiento del sistema *Swarm*. Un servicio se crea a partir de una imagen de contenedor como las que se han mencionado con anterioridad. *Swarm* crea una tarea (*task*) para el servicio en cada uno de los nodos trabajadores que forman el clúster. Estas tareas contienen un contenedor *Docker* junto a los comandos a ejecutar dentro de este contenedor. En el momento de la creación del servicio, el nodo administrador asigna tareas a los trabajadores en función del número de réplicas elegido.

De esta forma, cuando una aplicación está encapsulada dentro de un servicio del *Swarm*, las peticiones a esta pueden ir a cualquiera de los contenedores que forman parte del servicio, por lo que, si falla un nodo completo, la aplicación puede seguir funcionando en el resto de nodos. En el caso de disponer únicamente de un nodo, la aplicación también es resistente frente a fallos en los contenedores, ya que, si uno de ellos se para por algún motivo, el servicio volverá a crear otro para mantener el número de réplicas que se seleccionó al inicio.

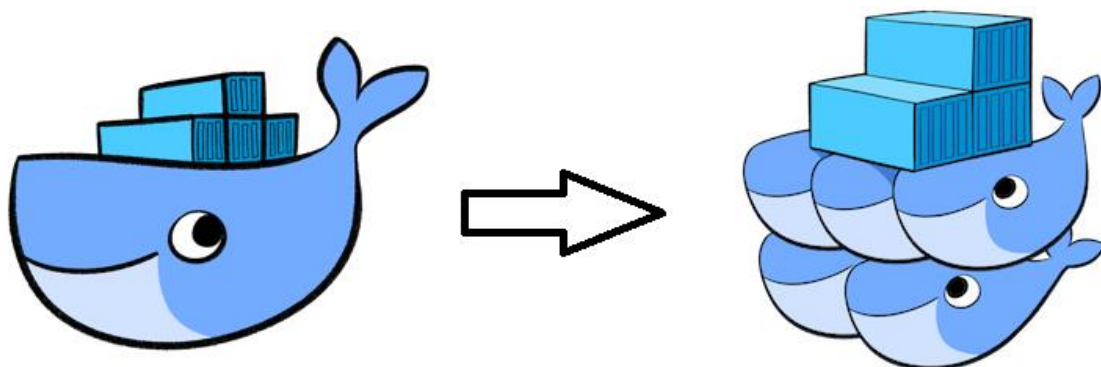


Figura 5: Diferencia entre la estructura de un contenedor Docker y un servicio Swarm a través de una representación empleando el logo de Docker. Fuente: [3]

En un servicio *Swarm* de contenedores *Docker* las peticiones de una aplicación pueden realizarse a cualquiera de los nodos que forman parte del clúster. Esto es debido a que, si un nodo recibe una petición correspondiente a una aplicación que no se encuentre en un contenedor de ese nodo, la petición se reenviará de forma automática a otro nodo del clúster hasta encontrar la aplicación. En la infraestructura de la empresa, las peticiones se realizan a un servidor que actúa de balanceador, reenviando la petición de forma aleatoria a uno de los nodos del clúster *Swarm*.

2.3.3. AWS

Docker tiene una gran integración con muy diversas herramientas de infraestructura y *Amazon Web Services* está entre ellas. Esto permite establecer una infraestructura muy similar a la del *Swarm*, pero más personalizable y compleja [10].

La infraestructura que se ha creado consiste en dos instancias *EC2* [11] para cada entorno de cada departamento. Estas instancias se crean a partir de un grupo de autoescalado, el cual permite que cuando las instancias empiecen a quedarse sin recursos, se cree otra nueva instancia preparada para contener nuevos contenedores.

También, en *AWS* se ha creado un balanceador de carga para cada entorno de cada departamento, en el que se irán creando *listeners* que apuntarán a los servicios de contenedores subidos a la nube.

Aunque el sistema implementado del que trata el TFG hace uso de esta infraestructura, esta no ha sido configurada por mí, sino por un compañero, por lo que no se entrará en los detalles de los diferentes elementos de los que dispone la infraestructura de *AWS*.

CAPÍTULO 3

DISEÑO DEL SISTEMA

CAPÍTULO 3. DISEÑO DEL SISTEMA

3.1. Componentes

El trabajo realizado se compone del desarrollo de una *API REST* que ejecuta operaciones sobre los distintos entornos de desarrollo mencionados previamente. Para que los desarrolladores de la empresa puedan realizar estas operaciones a través de llamadas a esta *API REST*, también se ha desarrollado conjuntamente, un bot de *Slack*. Este bot se comunica con los desarrolladores de forma que recoge las peticiones de los usuarios y, en función de ellas, ejecuta distintas llamadas a la *API REST* que controla el sistema basado en contenedores *Docker*. Finalmente, se ha implementado un sistema de control de tareas con *Jenkins* [12], que permite tener un registro de las peticiones realizadas por los usuarios a través del bot de *Slack*.

La estructura final del sistema puede verse representada en la siguiente figura.

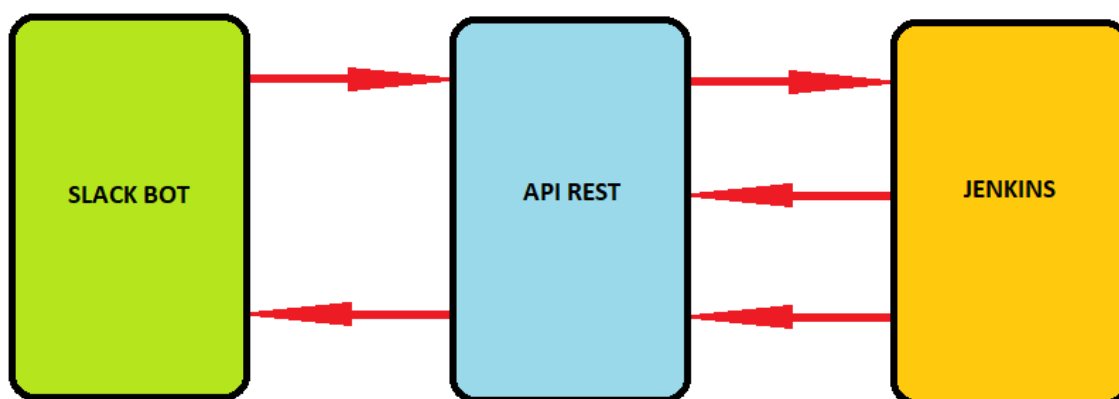


Figura 6: Esquema del sistema de automatización. Fuente: Propia

Como se puede observar, el sistema está dividido en tres partes, las cuales voy a explicar brevemente:

1. Bot de *Slack*: es el punto inicial desde donde el usuario introducirá los datos necesarios para lanzar tareas a través de comandos de texto.
2. *API REST*: su funcionalidad puede verse dividida en dos partes.
Por un lado, es la encargada de recibir peticiones de ejecución de tareas desde el bot de *Slack*, reenviando, a su vez, esta petición al servidor *Jenkins*.
Por otro lado, es la parte encargada de la ejecución, propiamente dicha, de estas tareas, tras recibir las correspondientes llamadas desde el servidor *Jenkins*.
3. Servidor *Jenkins*: es el encargado de controlar la ejecución de las tareas solicitadas por los usuarios. Este servidor recibe peticiones para realizar determinadas tareas, las ejecuta, y devuelve una confirmación al usuario a través de la *API REST*.

Una de las decisiones de gran importancia tomadas respecto al diseño del sistema, es el de ver el bot de *Slack* y la *API REST* como elementos diferenciados. Ambos elementos se encargan de lanzar llamadas a los servicios de otro servidor, por lo que podrían

haberse desarrollado como una sólo unidad. La decisión de separarlos se ha tomado para tener claramente separados el núcleo del sistema, encargado del funcionamiento en sí, de lo que puede verse como la interfaz de usuario, que en este caso sería el chat con el bot de *Slack*.

De esta forma, podemos decidir en cualquier momento, añadir otra forma de realizar las tareas o modificar la existente. Por ejemplo, se podría añadir una forma de lanzar las tareas a través de un formulario *HTML* en una interfaz *web*. También podría añadirse un bot de *Skype*, que lance las distintas tareas para sustituir al bot de *Slack* o complementarlo.

Como podemos ver en la siguiente imagen, estos cambios no variarían en absoluto la arquitectura del sistema, lo que podría ahorrar mucho tiempo y esfuerzo en un futuro si se tuviese que cambiar la “interfaz de usuario”.

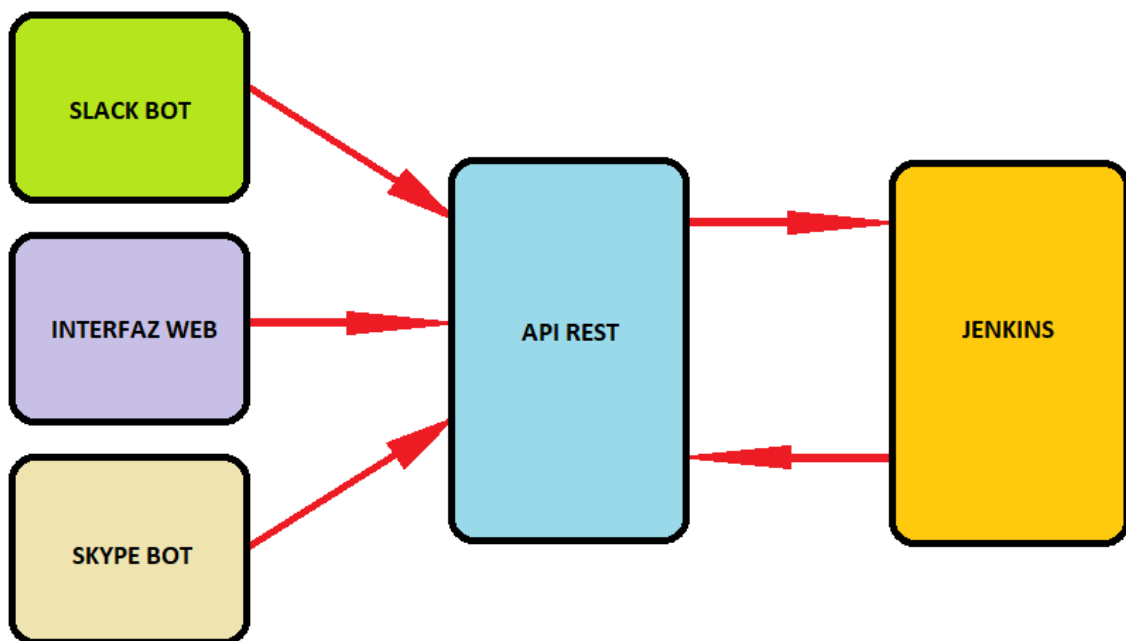


Figura 7: Sistema de automatización con diferentes elementos encargados de recopilar los datos de entrada. Fuente: Propia

Para facilitar este sistema en el que la interfaz de usuario podría ser modificada en un futuro, la *API REST* ha sido documentada a lo largo de su desarrollo con *Swagger* [13].

Swagger es una herramienta que permite describir, producir y visualizar servicios web *RESTful* a través de la especificación *OpenAPI* [14].

Swagger proporciona un editor web [15] en el que se realiza la documentación en formato *YAML*.

Además de *YAML*, también es posible emplear el formato *JSON* para realizar esta documentación [16]. Una vez finalizada, el editor web de *Swagger* nos permite exportar

la documentación como fichero *YAML* o *JSON* para ser cargada por una interfaz de usuario como *SwaggerUI* [17].

En la siguiente imagen se muestra la interfaz del *SwaggerUI*.

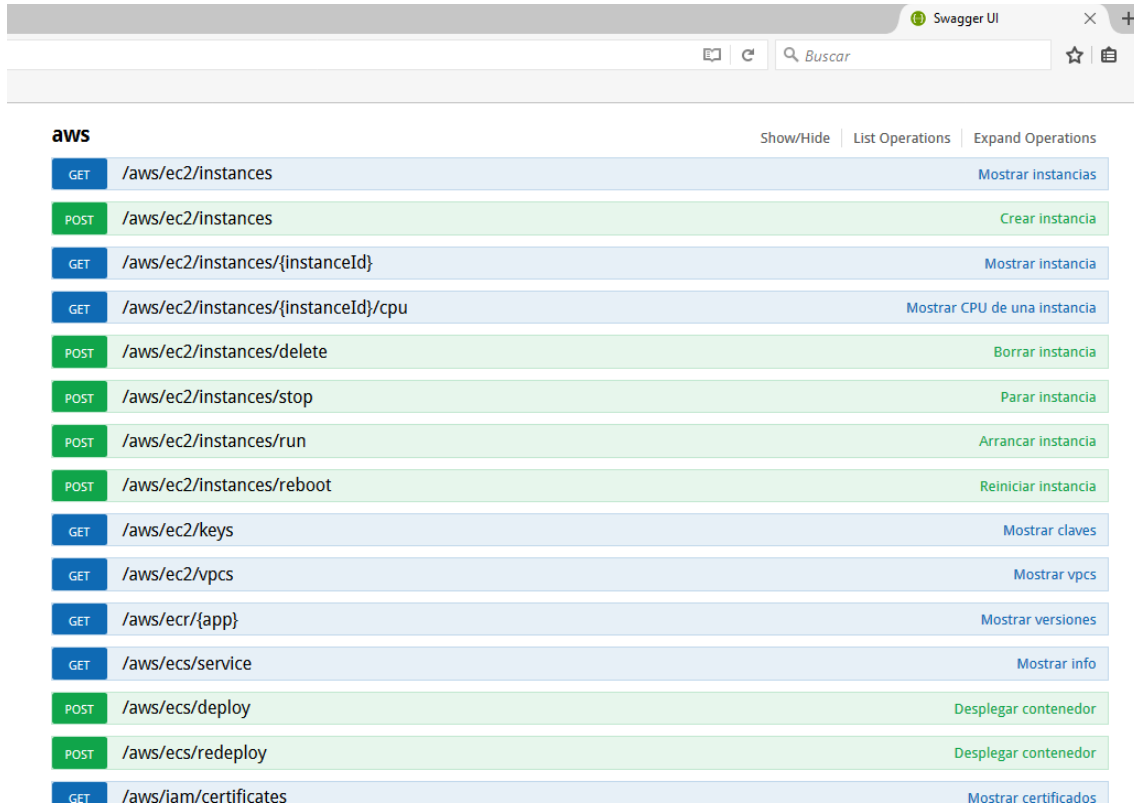


Figura 8: Interfaz de SwaggerUI. Fuente: Propia

Como puede verse en la imagen, a través de *SwaggerUI* podemos ver de forma ordenada la documentación generada previamente en el editor *Swagger*. Si se selecciona una de las rutas de la *API REST*, se muestran los detalles de la misma, como se muestra en la imagen que se ve a continuación.

POST /mysql Crear db

Implementation Notes
Crea una base de datos local o en aws

Response Class (Status 200)
string

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
user	<input type="text" value="(required)"/>	Usuario que solicita el servicio	formData	string
env	<input type="text" value="(required)"/>	Entorno de creacion	formData	string
name	<input type="text" value="(required)"/>	Nombre de la aplicacion para la que se desea la db	formData	string
amazon	<input type="text"/>	si o no crear la db en amazon	formData	string

Response Messages

HTTP Status Code	Reason	Response Model	Headers
400	Faltan parametros obligatorios	string	
500	Error creando la db. Error cargando departamento	string	
default	Error inesperado	string	

Figura 9: Estructura detallada en SwaggerUI de un servicio web. Fuente: Propia

En esta imagen puede verse con claridad el tipo de parámetros que puede recibir el servicio, además de mostrar cuáles de ellos son obligatorios. También se muestra los tipos de salidas que se devolverán con sus correspondientes códigos *HTTP*.

El uso de este tipo de documentación de la *API REST* refuerza la idea de la facilitar las futuras modificaciones del sistema. Por ejemplo, a la hora de generar o modificar la “interfaz de usuario” del sistema como se ha mencionado previamente, este *SwaggerUI* permitiría a una persona ajena al desarrollo de la *API REST*, conocer a la perfección los métodos para comunicarse con esta. De esta forma no se dependería de la comunicación con el desarrollador original de la *API REST*, ni de estudiar su código para trabajar en nuevos sistemas que mantengan intercambios con esta.

CAPÍTULO 4

TECNOLOGÍAS UTILIZADAS

CAPÍTULO 4. TECNOLOGÍAS UTILIZADAS

4.1. Introducción

A lo largo de este capítulo se van a describir las diferentes tecnologías empleadas para llevar a cabo el desarrollo de cada uno de los elementos que componen el sistema. El desarrollo correspondiente a la programación tanto del bot de *Slack*, como de la *API REST*, se ha realizado a través del editor de texto *Notepad ++* [18].

4.2. API REST

El desarrollo de la *API REST* se ha llevado a cabo con el *framework Slim* de *PHP* en su versión 3 [19].

La elección de *PHP* [20] frente a otros lenguajes del lado del servidor se ha realizado debido a la experiencia previa con *PHP* con la que contaba gracias a la asignatura “Tecnologías para aplicaciones Web” de cuarto curso del Grado en Ingeniería de Tecnologías Específicas de Telecomunicación con Mención en Ingeniería Telemática.

El *framework Slim* de *PHP* ordena el código en distintas rutas las cuales responden a una petición *HTTP* específica. Cada ruta ejecuta una función que se encarga de devolver una respuesta *HTTP* a la petición recibida.

Slim permite configurar la respuesta que se devolverá de una forma muy sencilla y rápida de aprender. Nos permite, por ejemplo, configurar el código *HTTP* que devolverá la respuesta, modificar el cuerpo del mensaje y alterar una cabecera determinada de la respuesta, entre otras cosas.

```
$app->POST('/v1/jenkins/reply', function($request, $response, $args) {  
  
    $idParam = $request->getParsedBodyParam('id');  
  
    *  
    *  
    *  
  
    $reply = "Usuario informado."  
    $newResponse = $response->withStatus(200)  
    ..... ->write($reply);  
    return $newResponse;  
});
```

Figura 10: Ejemplo de servicio web Slim de PHP. Fuente: Propia

En esta imagen puede verse un ejemplo de una ruta correspondiente a una petición *HTTP POST* en *Slim*. La función recibe tres parámetros predefinidos. El primero corresponde a un objeto de la petición recibida. De este parámetro podemos extraer, por ejemplo, los parámetros que se encuentran en el cuerpo del mensaje, que son aquellos que hayan sido enviados por *POST*. El segundo parámetro consiste en un objeto que contiene la respuesta a enviar por defecto. Esta respuesta puede ser modificada antes de

ser enviada, pero por recomendación, si se desea alterar este objeto, es preferible copiar este objeto y trabajar con la copia, la cual se devolverá. Como se ve en el ejemplo, de la respuesta pueden modificarse campos como el código de estado *HTTP*, o el cuerpo, entre otras muchas posibilidades. Por último, el tercer parámetro corresponde con un array, el cual contiene los parámetros enviados en la ruta. En este caso la ruta es fija, pero podría contener varios parámetros los cuales serían recogidos a través de este *array*.

Entre todas sus características, la sencillez del *framework Slim* y la experiencia previa de la que disponía con *PHP*, han sido los puntos clave a la hora de escoger este lenguaje para la realización de la *API REST*.

Otra de las tecnologías empleadas en el desarrollo de nuestra *API REST* ha sido la *API* de *Docker* [21].

A través de llamadas a esta *API* se realizan la administración de los contenedores tanto en el servidor *Docker* correspondiente al entorno de desarrollo, como del clúster *Swarm* de los entornos de preproducción y producción.

La documentación correspondiente a esta *API* es muy clara y dispone de una estructura similar a la de *SwaggerUI* con la que también se ha trabajado, como ya se ha mencionado previamente, para llevar a cabo la documentación de nuestra propia *API REST*.

También, se ha hecho uso de la Interfaz de Línea de Comandos de *Amazon Web Services* (por sus siglas, *AWS CLI*) [22].

Esta tecnología se ha empleado para llevar a cabo la comunicación con la infraestructura de la que la empresa dispone en la nube de *Amazon Web Services*.

Como se ha mencionado con anterioridad, además de los entornos de preproducción y producción locales implementados a través de un clúster de *Swarm*, la empresa también dispone de entornos de preproducción y producción en la nube de *AWS*. A la hora de elegir entre los entornos locales o los de la nube, se tendrán en cuenta las características de la propia aplicación, siempre tratando de priorizar los entornos en la nube. Esto es debido a su mayor sencillez en la configuración de, por ejemplo, *firewall* o certificados. De todas formas, pueden existir aplicaciones que requieran, por ejemplo, acceder a elementos locales de forma constante, de forma que, si se implementasen en la nube, su rendimiento se vería ralentizado, además de poderse presentar ciertos inconvenientes en lo que a seguridad se refiere.

La documentación de *AWS CLI* es realmente amplia, permitiendo realizar una configuración minuciosa y detallada de todos los servicios ofrecidos en *AWS*. Como inconveniente, esta enorme cantidad de posibilidades hace el uso de esta tecnología algo más complicado que el de, por ejemplo, la administración de servidores *Docker* locales.

Por último, otras tecnologías que se han utilizado durante el desarrollo de la *API REST* han sido otros servicios *web* desarrollados por miembros de la empresa. Estos servicios *web* accesibles a través de sus correspondientes *API*, son el servicio que administra el

servidor *DNS* local de la empresa, y el servicio que administra los empleados contratados por la empresa.

4.3. Bot de *Slack*

El lenguaje utilizado para el desarrollo del bot de *Slack* ha sido el *framework Botkit* de *Nodejs* [23]. Este *framework* está diseñado específicamente para comunicarse con *Slack* a través de su *Real Time Messaging API* [24], pero de una forma mucho más simple e intuitiva. Su sencillez es el motivo por lo que se ha elegido este lenguaje para el desarrollo del bot, además de que, al ser un *framework* de *Nodejs* [25], tiene grandes similitudes con el lenguaje *Javascript* [26].

Esto también ha facilitado mucho el desarrollo ya que contaba con experiencia previa con *Javascript*, gracias, de nuevo, a la asignatura “Tecnologías para aplicaciones Web” de cuarto curso del Grado mencionado anteriormente.

La estructura del *framework Botkit* de *Nodejs* puede verse de forma análoga al *framework Slim* de *PHP*, ya que consta de diferentes funciones que se ejecutan de forma independiente en función de la petición recibida. En este caso, en lugar de contar con una ruta a la que hay que realizar una petición, se tiene una cadena de texto, con la cual se realizará una comparación entre lo que se ha introducido en el chat de *Slack* y la condición de cada una de las funciones.

En la siguiente imagen se puede ver un ejemplo de esto, de forma que si el texto recibido por el bot a través del chat de *Slack*, cumple la condición impuesta, se ejecutará el código que contenga la función.

```
controller.hears(['^ *mostrar vpcs(.) *$', '^ *show vpcs(.) *$'],  
  ['direct_message', 'direct_mention', 'mention'], function(bot, message) {  
  });
```

Figura 11: Ejemplo de implementación de un comando en el bot de Slack. Fuente: Propia

Como se puede ver, también el tipo de mensaje recibido por el bot. En este caso, se estaría recibiendo los mensajes enviados al bot a través de mensajes directos, menciones en una conversación privada y menciones a través de un chat grupal.

En la siguiente figura podemos ver la interfaz de la que dispone la aplicación de escritorio de *Slack*.

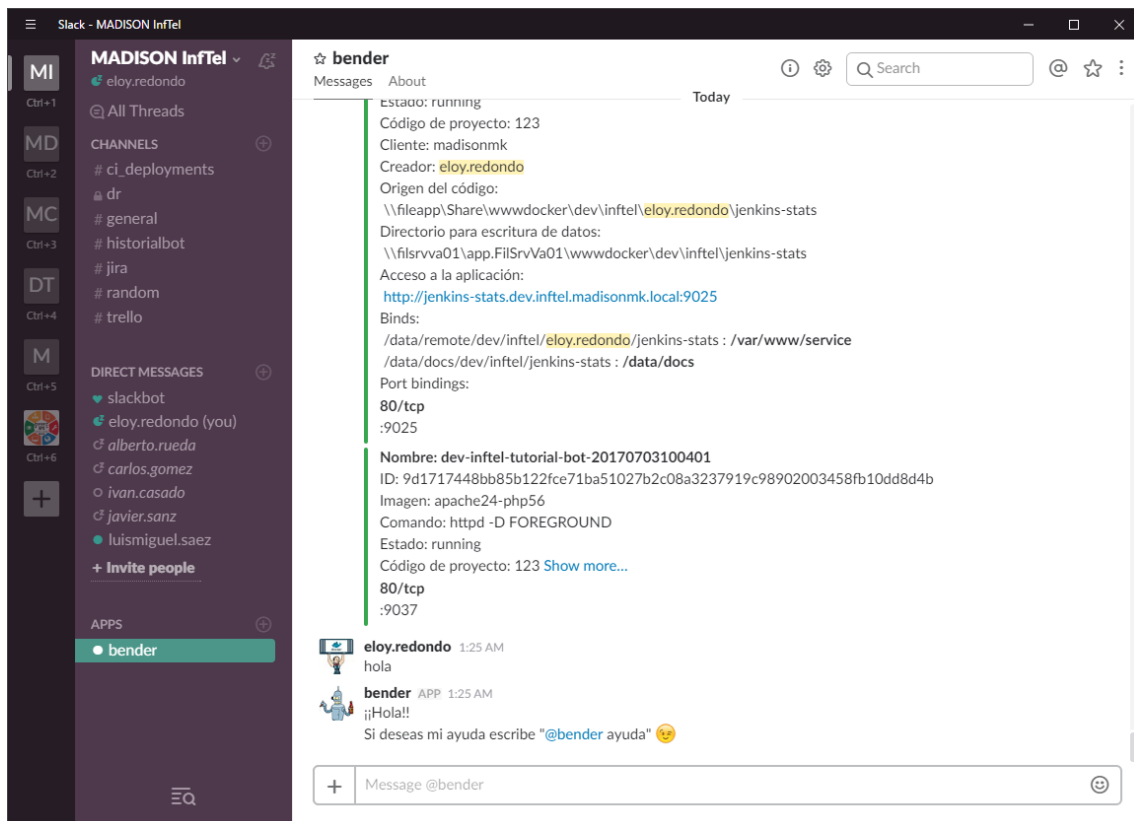


Figura 12: Interfaz de Slack. Fuente: Propia

En la imagen podemos ver la conversación con el bot, al que se ha nombrado “*bender*”. Los pasos a seguir para conectar *Slack* con nuestro bot desarrollado con el *framework Botkit*, son muy sencillos. Esto es gracias a que la conexión entre *Slack* y el bot desarrollado, se realiza desde el propio código del bot, a través de una *API key* que se nos facilita tras seguir unos pocos pasos en *Slack*.

Lo primero que debe hacerse es acceder a través de la web de *Slack*, a la sección de *Custom Integrations* de nuestro grupo. Luego, tras seleccionar la opción *Bots*, configuraremos uno nuevo para obtener la *API key*, como se muestra a continuación.

slack | App Directory | Search App Directory | Browse | Manage | Build | MADISON...

Browse Apps > Custom Integrations > Bots > Edit configuration

Bots | Added by eloy.redondo on October 7th, 2016 | Disable • Remove

Run code that listens and posts to your Slack team just as a user would.

Setup Instructions

Please refer to our [bot user API documentation](#), which tells you everything you need to know about setting up a bot integration.

Integration Settings

API Token

The library you are using will want an API token for your bot.

[Redacted API Token]

[Regenerate](#)

⚠ Be careful when sharing bot user tokens with applications. Do not publish bot user tokens in public code repositories. [Review token safety tips.](#)

Customize Name

Choose the username for this bot.

@bender

Figura 13: Pantalla de configuración de un bot de Slack. Fuente: Propia

Una vez configurado el bot del lado de *Slack*, es el turno de configurarlo en el lado del código, donde sólo tendremos que iniciar el proceso de *Nodejs* con la *API key* como parámetro.

```

if (!process.env.token) {
  console.log('Error: Specify token in environment');
  process.exit(1);
}

var Botkit = require('./lib/Botkit.js');
var os = require('os');

var controller = Botkit.slackbot({
  debug: true
});

var bot = controller.spawn({
  token: process.env.token
});

function start_rtm() {
  bot.startRTM(function(err, bot, payload) {
    if (err) {
      console.log('Failed to start RTM');
      return setTimeout(start_rtm, 5000);
    }
    console.log("RTM started!");
  });
}

controller.on('rtm_close', function(bot, err) {
  start_rtm();
});

start_rtm();

```

Figura 14: Inicialización del proceso Nodejs del bot de Slack. Fuente: Propia

Botkit permite también iniciar conversaciones de forma que, una vez recibido el texto que inicia la conversación, el resto de los mensajes enviados no lanzarán nuevas funciones, sino que se irán almacenando para poder ser utilizados posteriormente, tras finalizar la conversación.

Otra de las funcionalidades interesantes de este *framework*, es que se puede formatear los mensajes devueltos por el bot de forma que se aprovechen las funcionalidades que otorga *Slack*. Esto se hace devolviendo los mensajes en formato *JSON* (*JavaScript Object Notation*). De esta forma podremos conseguir mensajes más ordenados, más legibles y más estéticos.

4.4. Jenkins

El último de los tres elementos que compone el sistema de automatización, es el servidor *Jenkins* [12]. La configuración de este servidor se hace a través de su interfaz *web*, la cual resulta muy sencilla de utilizar. En la siguiente imagen se muestra la

pantalla principal correspondiente a la interfaz del servidor *Jenkins* implementado para nuestro sistema.

The screenshot shows the Jenkins web interface. At the top, there is a navigation bar with the Jenkins logo, a search bar, and user information (Administrador TI | Desconectar). Below the navigation bar, there is a sidebar on the left with various menu items like 'Nueva Tarea', 'Personas', 'Historial de trabajos', etc. The main area displays a table of tasks under the heading 'Todas las tareas.' The table has columns for 'S', 'W', 'Nombre', 'Último Éxito', 'Último Fallo', and 'Última Duración'. The tasks listed include 'agency-bmw-deploy', 'app-container-run', 'app-rebuild-cem_prosecur_cash_cronjobs', 'app-rebuild-dts-scheduledjobs', 'cem-python-ci', 'container_clone', 'container_create', 'container_delete', 'container_exec', 'container_start', 'container_stop', 'cron-digitelts-scheduledjobs', 'cron-estudios', 'db_create', 'db_exec', 'db_import', 'db_resize', 'deploy', and 'git-apache-deploy'.

S	W	Nombre	Último Éxito	Último Fallo	Última Duración
		agency-bmw-deploy	7 días 20 Hor - #2	7 días 20 Hor - #1	9 Min 59 Seg
		app-container-run	1 Min 15 Seg - #12423	N/D	0.72 Seg
		app-rebuild-cem_prosecur_cash_cronjobs	11 días - #6	N/D	4 Min 20 Seg
		app-rebuild-dts-scheduledjobs	28 días - #8	28 días - #3	34 Seg
		cem-python-ci	3 días 22 Hor - #158	6 días 20 Hor - #153	27 Min
		container_clone	23 Hor - #3	N/D	3.7 Seg
		container_create	1 Hor 0 Min - #28	N/D	10 Min
		container_delete	20 Hor - #17	N/D	1.1 Seg
		container_exec	5 días 21 Hor - #5	N/D	1 Min 4 Seg
		container_start	44 Min - #37	N/D	1.3 Seg
		container_stop	19 Hor - #6	N/D	1.1 Seg
		cron-digitelts-scheduledjobs	16 Min - #2084	N/D	0.13 Seg
		cron-estudios	1 Min 22 Seg - #8260	N/D	51 Ms
		db_create	6 días 20 Hor - #7	7 días 20 Hor - #6	35 Seg
		db_exec	3 días 19 Hor - #20	4 días 0 Hor - #17	8 Min 33 Seg
		db_import	N/D	N/D	N/D
		db_resize	N/D	N/D	N/D
		deploy	7 días 0 Hor - #27	N/D	2 Min 2 Seg
		git-apache-deploy	N/D	6 días 21 Hor - #1	31 Min

Figura 15: Interfaz principal del servidor Jenkins implementado. Fuente: Propia

Como se puede observar, la interfaz se compone de varias secciones. La primera de ellas es la sección principal donde se muestran las tareas creadas, ordenadas en diferentes pestañas. A su izquierda vemos que en la parte superior se encuentran los distintos enlaces a los menús de configuración. Justo debajo, puede verse la sección correspondiente a las tareas actualmente en cola y ejecución.

Jenkins dispone de la funcionalidad de extenderse mediante *plugins* [27], que podemos descargar o desarrollar libremente. Estos permiten agregar nuevas funcionalidades como la realización de peticiones *HTTP*, o el envío de correos electrónicos tras la finalización de una tarea.

Este servidor será el encargado de recibir peticiones de ejecución de tareas y, a continuación, realizar la llamada a nuestra *API REST* que ejecute dicha tarea. De esta forma, las peticiones quedan registradas, junto con el usuario que la solicitó y la fecha de ejecución.

Para llevar a cabo esto, es necesaria la instalación del *plugin* que permite realizar peticiones *HTTP*. Este plugin puede instalarse desde su menú correspondiente, el cual se muestra a continuación.

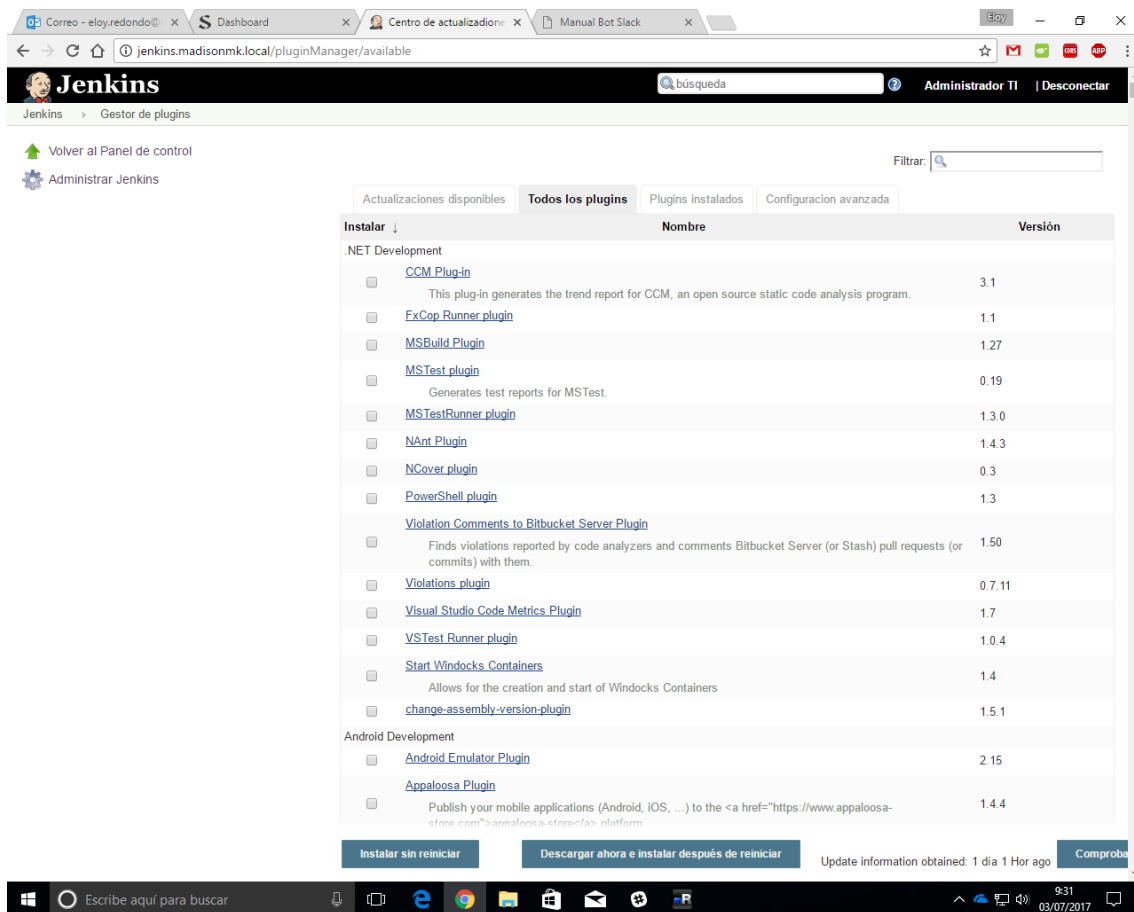


Figura 16: Menú de gestión de plugins del servidor Jenkins. Fuente: Propia

La elección de *Jenkins* para controlar la ejecución de tareas por parte de los usuarios, es debido a que además de la sencillez de su configuración, cuenta con un gran soporte y una gran integración con otras plataformas. Además, aunque en este caso su función principal sea la del control de tareas, es un *software* de integración continua. Como se verá más adelante, esto permite, un nuevo nivel en lo que automatización del desarrollo de aplicaciones se refiere.

CAPÍTULO 5

DESARROLLO DEL SISTEMA

CAPÍTULO 5. DESARROLLO DEL SISTEMA

5.1. Introducción

Para empezar a describir los pasos seguidos en el desarrollo del sistema, primero se describirán de forma detallada, todas las comunicaciones existentes entre cada uno de los tres componentes de este, que son el bot de *Slack*, la *API REST* y el servidor *Jenkins*.

En el siguiente esquema pueden verse cada una de las comunicaciones que tienen lugar en el momento en el que un usuario inicia la ejecución de una tarea desde el chat con el bot de *Slack*.

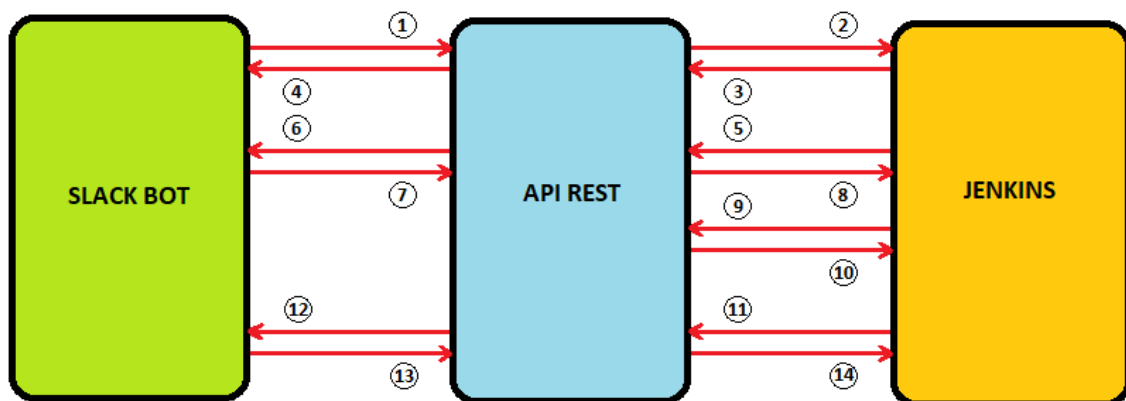


Figura 17: Esquema de las comunicaciones que se realizan a través del sistema de automatización. Fuente: Propia

Todas estas comunicaciones pueden verse como parte de tres pasos diferenciados, los cuales se van a describir:

- Creación de la tarea:
 1. Esta primera comunicación corresponde a una petición *HTTP POST* que se realiza desde el bot a la *API REST* en el momento en el que un usuario ha lanzado una tarea a través del chat de *Slack*.
 2. La *API REST* solicita la ejecución de una tarea en concreto al servidor de *Jenkins*.
 3. Una vez lanzada la tarea, el servidor de *Jenkins* confirma a la *API REST* la ejecución de la misma.
 4. La *API REST*, a su vez, confirma al bot el lanzamiento de la tarea, de forma que el usuario recibirá a través del chat de *Slack*, un mensaje de confirmación, haciéndole ver que la tarea se ha lanzado, o en el caso de no hacerlo, un mensaje de error.
- Ejecución de la tarea:
 5. Una vez iniciada la tarea, lo primero que hace el servidor de *Jenkins* es comunicar al usuario un tiempo estimado de ejecución de la tarea. Esto se realiza a través de una petición *HTTP POST* determinada a la *API REST*.

6. La *API REST* llama en este momento a un servicio que está escuchando en el bot de *Slack*, haciendo que este comunique al usuario el tiempo estimado de ejecución.
 7. Una vez informado el usuario, esto se confirma a la *API REST*.
 8. La *API REST*, por su parte, también confirma al servidor *Jenkins* que el usuario ha sido debidamente informado.
 9. A la vez que se iniciaba la comunicación para informar al usuario del tiempo estimado de ejecución, también se iniciaba una petición *HTTP POST* a la *API REST* para ejecutar la tarea en sí.
 10. Una vez realizada la tarea por parte de la *API REST*, se comunica al servidor *Jenkins* la salida de la ejecución.
- Devolución de la salida:
 11. Tras finalizar la ejecución de la tarea, *Jenkins* comienza la ejecución de una nueva tarea que será la encargada de comunicar la finalización de la anterior al usuario. Para ello, esta tarea realiza una petición *HTTP POST* a la *API REST*, de forma similar a cuando se deseaba informar al usuario del tiempo estimado de ejecución.
 12. La *API REST* se pone en comunicación con el bot de *Slack* de nuevo, el cual escribe al usuario para informarle de que la tarea ha finalizado su ejecución.
 13. El bot de *Slack* confirma a la *API REST* que el usuario ha recibido el mensaje.
 14. Finalmente, la *API REST* finaliza la comunicación con el servidor de *Jenkins*, confirmándole que el usuario ha sido informado.

5.2. Descripción técnica

5.2.1. Introducción

A lo largo de este documento se ha descrito la motivación de implementar un nuevo sistema de desarrollo de aplicaciones, la utilidad de su automatización, las tecnologías empleadas en su desarrollo y las comunicaciones entre los distintos elementos que lo componen.

Este punto se centrará en los pasos concretos que se llevan a cabo en cada una de las tareas que el bot nos permite ejecutar. Todos los comandos que se han implementado en el bot de *Slack* están descritos en la sección correspondiente al manual de usuario y, como se podrá comprobar, muchos de ellos son meramente informativos, por lo que en realidad no es necesaria la ejecución de tareas en el servidor de *Jenkins*, para que el usuario obtenga la respuesta. Esto es debido a que el objetivo de la implementación del servidor *Jenkins*, como se ha mencionado con anterioridad, se centra en dejar constancia de las tareas realizadas por los usuarios, de forma que la solicitud de información no se ha tenido en cuenta como ejecución de tarea. Por esta razón, todos los comandos del bot de *Slack* que solicitan información sólo realizan peticiones *HTTP GET* a la *API REST*, dejando sin intervenir al servidor *Jenkins*.

Los distintos comandos que sí se encargan de la ejecución de tareas, por lo que realizan cambios en alguno de los entornos de desarrollo y dejan constancia de su ejecución en el servidor *Jenkins*, puede agruparse en los siguientes grupos:

- Creación de aplicaciones.
- Operaciones en contenedores o servicios de contenedores.
- Subidas de nuevas versiones a preproducción y producción.
- Creación de bases de datos.
- Operaciones en bases de datos.

Todas las tareas que se ejecutan a través del servidor *Jenkins*, deben estar previamente configuradas en dicho servidor. Esta configuración se ha realizado tarea a tarea a través de la interfaz de usuario de Jenkins. Las tareas de Jenkins sólo realizan llamadas a la *API REST*, que es realmente donde se ejecutarán las acciones pertinentes.

A continuación, se muestra un ejemplo de la configuración de una tarea en *Jenkins*.

Jenkins [Administra](#)

Jenkins > Monitoring > container_start >

General Configurar el origen del código fuente Disparadores de ejecuciones Entorno de ejecución Ejecutar

Acciones para ejecutar después.

Proyecto nombre

Descripción

[Plain text] [Visualizar](#)

Run the build inside Docker containers

Desechar ejecuciones antiguas [?](#)

Strategy

Numero de días para mantener ejecuciones de proyectos

si no está vacío, sólo se mantendrán las ejecuciones con una edad inferior a este número de días

Número máximo de ejecuciones para guardar

si no está vacío, sólo se guardarán un número de ejecuciones inferior a este valor

[Avanzado...](#)

Docker Container

Esta ejecución debe parametrizarse [?](#)

Parámetro de texto [X](#)

Name [?](#)

Default Value

Description

[Plain text] [Visualizar](#)

Parámetro de texto [X](#)

Name [?](#)

Default Value

Description

[Plain text] [Visualizar](#)

Parámetro de texto [X](#)

Name [?](#)

[Guardar](#) [Apply](#)

Figura 18: Ejemplo de configuración de una tarea en Jenkins (1). Fuente: Propia

En la primera parte de la configuración, se asigna un nombre a la tarea y se escoge el número de días que las ejecuciones permanecerán guardadas en el servidor. De este modo se evita que el servidor almacene datos de tareas ya ejecutadas de forma indefinida.

El siguiente punto importante en la configuración se trata de establecer, cuando sea necesario, los parámetros que se deben recibir para ejecutar la tarea. En nuestro caso, todas las tareas reciben, como mínimo, dos parámetros: *channel* y *user*.

El parámetro *channel* será necesario para devolver la respuesta a la tarea al usuario que la ejecutó, como veremos más adelante. Por otro lado, el parámetro *user*, siempre se recibirá para dejar constancia del usuario que ha realizado cada tarea, aunque en la mayoría de ocasiones también se le da uso dentro de la funcionalidad de la propia tarea en la *API REST*.

A continuación, lo siguiente a configurar en las tareas de *Jenkins*, son los pasos de ejecución. Estos pasos como ya hemos mencionado, se tratan de peticiones *HTTP POST* a nuestra *API REST*. Más concretamente, y como se ha descrito con anterioridad, cada tarea primero realiza una llamada para enviar al usuario un tiempo de ejecución estimado. A continuación, realiza la llamada que ejecuta propiamente la tarea en la *API REST*. Finalmente, tras terminar la ejecución, la tarea invoca a otra que, a través de una última llamada a la *API REST*, comunicará al usuario la finalización de la ejecución.

Se la siguiente imagen puede verse la implementación de estos tres pasos en la tarea de *Jenkins*.

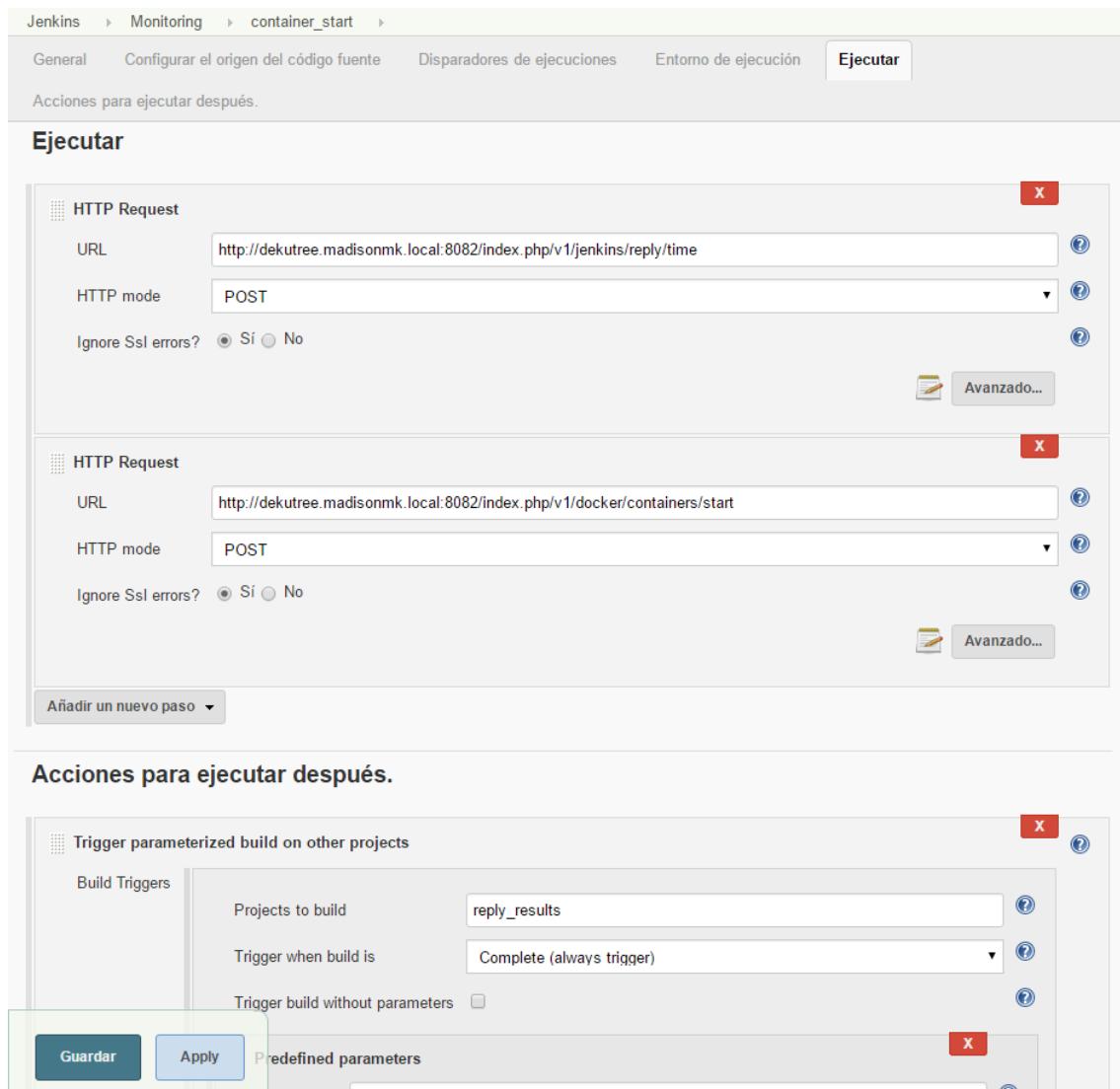


Figura 19: Ejemplo de configuración de una tarea en Jenkins (2). Fuente: Propia

Si nos centramos en las opciones avanzadas de la configuración de una de las peticiones *HTTP* que se ejecutan, se nos muestra los siguientes campos que podemos personalizar.

Jenkins > Monitoring > container_start >

General Configurar el origen del código fuente Disparadores de ejecuciones Entorno de ejecución **Ejecutar**

Acciones para ejecutar después.

URL: ?

HTTP mode: ?

Ignore Ssl errors? Sí No ?

Authorization

Authenticate: ?

Headers

Accept: ?

Content-type: ?

Custom headers

Header	<input type="text" value="Content-type"/>
Value	<input type="text" value="application/x-www-form-urlencoded"/>
Mask value	<input type="checkbox"/>

If checked, this will mask the value in the logs.

Body

Pass build params to URL? Sí No ?

Request body: ?

Figura 20: Ejemplo de configuración de una tarea en Jenkins (3). Fuente: Propia

Estos campos personalizables, permiten modificar las cabeceras de la petición, así como el cuerpo o, si fuese necesario, incluir autenticación.

Una vez que se tienen las tareas configuradas en el servidor de *Jenkins*, cuando un usuario ejecute un comando en el bot de *Slack* correspondiente a una de estas tareas, se lanzará de ejecución de dicha tarea a través de la *API REST*, como ya se ha descrito en la secuencia de comunicaciones que se realizan entre los tres elementos del sistema.

Este servicio *web* hace uso de la *API* de *Jenkins* para lanzar la ejecución de la tarea solicitada por el usuario, como se muestra a continuación.

```

$app->POST('/vl/jenkins/containers/clone', function($request, $response, $args) {

    $channel = $request->getParsedBodyParam('channel');
    $user = $request->getParsedBodyParam('user');
    $name = $request->getParsedBodyParam('name');
    $image = $request->getParsedBodyParam('image');
    if($channel && $user && $name && $image){
        $login = getenv("JENKINS_USER");
        $pass = getenv("JENKINS_PASS");
        $host = getenv("JENKINS_HOST");
        $token = getenv("JENKINS_TOKEN");
        $tarea = "container_clone";
        $ch = curl_init($host . "/job/" . $tarea . "/buildWithParameters?token=" . $token
            . "&name=" . $name
            . "&image=" . $image
            . "&user=" . $user
            . "&channel=" . $channel);
        curl_setopt($ch, CURLOPT_POST, true);
        curl_setopt($ch, CURLOPT_USERPWD, "$login:$pass");
        curl_setopt($ch, CURLOPT_HTTPAUTH, CURLAUTH_BASIC);
        curl_setopt($ch, CURLOPT_HEADER, true);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
        $headers = curl_exec($ch);
        curl_close($ch);

        $headers = explode("\r\n", $headers);
        for($i=0;$i<count($headers);$i++){
            if(strpos($headers[$i], 'Location') !== false){
                $url = explode(' ', $headers[$i]);
                break;
            }
        }
        $url = $url[1] . "api/json";
        $ch = curl_init($url);
        curl_setopt($ch, CURLOPT_USERPWD, "$login:$pass");
        curl_setopt($ch, CURLOPT_HTTPAUTH, CURLAUTH_BASIC);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
        $resp = curl_exec($ch);
        curl_close($ch);

        $resp_json = json_decode($resp);
        if($resp_json->id == ""){
            $reply = "Error en servidor Jenkins.";
            $status = 500;
        }else{
            $reply = "Tarea " . $tarea . " creada por " . $user . " con ID: " . $resp_json->id;
            addHistorial($reply);
            $reply = $resp_json->id;
            $status = 200;
        }
    }
    else{
        $reply = "Opciones incorrectas.";
        $status = 400;
    }
    $newResponse = $response->withStatus($status)
        ->write($reply);
    return $newResponse;
});

```

Figura 21: Ejemplo de llamada para lanzar una tarea en Jenkins. Fuente: Propia

Por su parte, los comandos destinados a la obtención de datos para informar al usuario, los cuales no lanzan la ejecución de tareas en *Jenkins*, pueden agruparse como:

- Información sobre la infraestructura en la nube de AWS.
- Información sobre contenedores o servicios de contenedores.
- Información sobre tareas.

A continuación, se explicarán algunos procedimientos que se llevan a cabo en la *API REST* para ejecutar cada una de las tareas más importantes, incluyendo la interacción con la *API* de *Docker* o con otros servicios *web* y la utilización de las herramientas previamente descritas.

5.2.2. Creación de aplicaciones

5.2.2.1. Entorno de desarrollo

A la hora de crear de cero una nueva aplicación, el primer paso es el de crear un contenedor en el entorno de desarrollo. Para ello, el usuario debe introducir el comando correspondiente, *crear contenedor*.

Tras introducir el comando, el bot iniciará una conversación en la que se irá preguntando al usuario distintos parámetros y características del contenedor que desea crear. Lo primero que se da a elegir es el entorno de creación para el contenedor. Tras seleccionar la opción *dev* (desarrollo), se realizarán una serie de preguntas entre las que se encuentran:

- El nombre de la aplicación, que servirá para crear, junto con el entorno seleccionado, el nombre del contenedor, la ruta de acceso al código fuente y la *URL* de acceso a la aplicación. Los detalles sobre este proceso se detallarán más adelante.
- La imagen con la que crear el contenedor, que, como se ha detallado previamente, será la que defina el tipo de contenedor que se creará, incluyendo el sistema operativo y los módulos que incluirá, entre otras cosas.
- Directorio de escritura para ficheros, si se desea uno.
- Ruta de lectura, por si la aplicación requiere acceso a algún servidor de la empresa.

Tras introducir los parámetros deseados, a través de las distintas llamadas que se han descrito con anterioridad, se ejecutará la tarea que se encargará de la creación del contenedor, a través de una petición a la *API REST* desarrollada.

Para llevar a cabo la creación del contenedor, la función correspondiente de la *API REST*, ordena los parámetros recibidos en un objeto en formato *JSON* y, a continuación, realiza una llamada a la *API* de *Docker* del servidor de desarrollo que se encarga de la creación del contenedor con el *JSON* recibido.

Como se introdujo en la sección correspondiente a los contenedores *Docker*, aunque su funcionalidad resulte similar a la de una máquina virtual, en realidad son procesos ejecutándose en el servidor. Esto hace que los contenedores no sean persistentes, de forma que, si un contenedor es destruido, todo su contenido es eliminado.

Para evitar esto, lo que se hace en la creación del contenedor es lo que se denomina como *bind*. A partir de los parámetros que se introdujeron por *Slack*, se creará un *bind* en el contenedor que enlace el *document root* (directorio donde se busca el código de la aplicación) del contenedor, con un directorio del sistema de ficheros del servidor.

En nuestro caso, el *document root* de cada contenedor se almacena en el directorio */var/www/service* del sistema de ficheros interno del contenedor. Este directorio se enlaza con el directorio */data/remote/{entorno}/{departamento}/{usuario}/{aplicación}*.

De esta forma se consigue que el código de la aplicación que corre dentro de cada contenedor, sea persistente y no se elimine junto con el contenedor, pero aún falta algo fundamental para el entorno de desarrollo, que es que los desarrolladores puedan modificar el código de una forma rápida y que estos se vean reflejados lo antes posible.

Para ello, lo que se ha hecho es montar en el directorio */data/remote* del servidor de desarrollo mencionado antes, otro directorio de un servidor al que los usuarios pueden acceder a través del explorador de archivos de *Windows*. De esta forma se consigue que los desarrolladores puedan trabajar en su máquina local, y los cambios que se realizan se vean reflejado al instante.

Este sistema de *binds* también se aplica para la creación del directorio de escritura de datos y el acceso al directorio de lectura, en el caso en el que el usuario lo hubiese seleccionado.

Otro de los parámetros que se configura mediante el *JSON* es el puerto en el que el contenedor escuchará. Para ello, durante la creación se asigna el primer puerto libre del que se dispone en un rango determinado. Posteriormente, tras la creación del contenedor, se procede a crear un alias en el servidor *DNS* local, que sea fácilmente recordable por los usuarios y que apunte al servidor *Docker* de desarrollo.

Tanto la obtención del departamento del usuario, como la creación del alias en el *DNS* para acceder a la aplicación, se consiguen a través de servicios web internos de la empresa ya existentes.

De esta forma ya tendríamos el contenedor creado en el entorno de desarrollo. Por supuesto, todo este procedimiento es transparente para el usuario que ejecuta la tarea. En la sección correspondiente del manual se detalla el procedimiento desde el punto de vista de usuario.

5.2.2.2. Entorno de preproducción

Para la creación del servicio de contenedores correspondiente al entorno de preproducción el procedimiento es distinto en función de si el usuario elige subir la aplicación a la nube de *AWS* o tenerla en la infraestructura local a través de un servicio en un clúster *Swarm*.

Al igual que en el caso anterior, el proceso se inicia cuando el usuario introduce el comando *crear contenedor*, salvo que en esta ocasión selecciona *pre* (preproducción) como entorno. A continuación, las preguntas que se le realizan al usuario son, fundamentalmente:

- Seleccionar un contenedor creado previamente en desarrollo para pasarlo a preproducción.
- Introducir el repositorio de *Bitbucket* y la rama en la que se encuentra el código correspondiente a su aplicación.

Bitbucket [28] es un servicio de alojamiento web, el cual tiene contratado la empresa para que cada grupo de desarrollo guarde las versiones de sus aplicaciones. De esta forma, cuando la *API REST*, comienza la ejecución de la tarea correspondiente a la creación de un servicio de contenedores en preproducción, lo que se realizará será añadir el código de la aplicación contenido en la rama del repositorio introducido por el usuario, a la imagen con la que se había creado el contenedor de desarrollo.

Como puede verse, de esta forma el código de la aplicación pasa a formar parte del contenedor, en lugar de almacenarse externamente como se hacía en el entorno de desarrollo. Esto es debido a que, en preproducción y producción, los cambios no se realizan de forma tan habitual, por lo que, para realizarlos, será necesario crear de nuevo la imagen para introducir la última versión del código de la aplicación.

Por el contrario, al igual que en el caso anterior, si se desea disponer de un escritorio de lectura o escritura, esto sí se hará a través de *binds*, para así tener siempre bien diferenciados los ficheros que componen la aplicación, de los ficheros que esta puede generar como consecuencia de su funcionamiento. De todas formas, esto sólo es posible hacerlo cuando el usuario haya seleccionado mantener la aplicación en la infraestructura local y no en la nube de *AWS*.

En el caso de crear la aplicación en la nube de *Amazon Web Services*, el proceso que se lleva a cabo es bastante más complejo y consta de múltiples pasos realizados a través de *AWS CLI*. Estos pasos van desde la creación como tal del servicio, hasta la creación de un *listener* en un balanceador que conduzca las peticiones que recibe la aplicación a los contenedores que se encuentren repartidos por las distintas instancias *EC2* de las que se disponga.

5.2.2.3. Entorno de producción

En este caso, cuando el usuario selecciona *pro* (producción), como entorno de creación, el único procedimiento que se lleva a cabo es análogo al del entorno de preproducción, ya sea para aplicaciones en la nube o locales.

5.2.3. Operaciones con contenedores

Una diferencia a destacar entre un contenedor de desarrollo y un servicio de contenedores de, por ejemplo, preproducción, es que el servicio se mantiene los contenedores corriendo, mientras que el contenedor de desarrollo puede estar parado o arrancado. Cuando se crea un contenedor en desarrollo, este se encuentra inicialmente parado.

Se podría haber configurado la función de creación de contenedores en desarrollo de la *API REST* para arrancar el contenedor una vez creado, pero hay un caso especial en el que resulta útil que sean los usuarios los que puedan arrancar a través del bot de *Slack* sus contenedores, cosa que pueden hacer a través del comando *arrancar contenedor*. Este caso es en el que un usuario haya decidido crear su contenedor a partir de una imagen que no contenga ningún tipo de servidor, por ejemplo, una imagen que contenga un intérprete *PHP*, pero no un servidor *Apache* [29]. En este caso, arrancar el

contenedor, no supone mantener corriendo un proceso *Apache*, sino ejecutar un fichero *PHP*, que, tras ser ejecutado, dejará el contenedor parado de nuevo.

Otra de las operaciones útiles con contenedores a tener en cuenta es la ejecución de comandos dentro de los contenedores. Esta acción puede ser llevada a cabo por parte de los usuarios a través del chat de *Slack* con el comando *ejecutar comando*. Por ejemplo, gracias a esto, los usuarios pueden hacer uso del manejador de dependencias *Composer* [30], el cual es bastante usado en aplicaciones *PHP*, sin necesidad de solicitar el acceso por *SSH* al interior del contenedor.

5.2.4. Subidas de nuevas versiones de una aplicación

Previamente, se ha mencionado que, en los entornos de preproducción y producción, el código de las aplicaciones se incluye dentro de la propia imagen con la que se crean los contenedores, por lo que no puede ser modificada manualmente.

Esto, sin embargo, no supone que haya que realizar todo el proceso de creación de un servicio de contenedores nuevo en estos entornos cada vez que se quiera actualizar la aplicación a una nueva versión. Para ello, los usuarios disponen del comando *redesplegar aplicacion* en el bot de *Slack*, que les permite tanto realizar una nueva subida de código, como volver a una versión anterior de la aplicación de una forma muy sencilla.

El proceso que se realiza en la *API REST* realmente es muy similar al de la creación del servicio de contenedores original, sin embargo, en lugar de pedir de nuevo todos los parámetros al usuario, estos se extraen de unos *labels* creados en la primera versión del servicio, haciendo más fácil e intuitiva la realización de estos procedimientos por parte de los usuarios.

5.2.5. Creación de bases de datos

La creación de bases de datos a través del bot de *Slack*, permite a los usuarios crear de forma autónoma bases de datos tanto en desarrollo como en preproducción. Para crear una base de datos de producción, los desarrolladores deben seguir contactando con el departamento de Infraestructura y Telecomunicaciones, ya que es un proceso que depende de cada tipo de aplicación y de los requisitos de la misma.

El servicio web de la *API REST* que lleva a cabo la creación de las bases de datos, se encarga de elegir el host donde se creará en función del entorno elegido. Además, se creará un usuario de acceso a la vez que la base de datos.

En la infraestructura local, que en este caso incluye el entorno de desarrollo y uno de los entornos de preproducción, esta creación se hace a través de *Ansible* [31]. *Ansible* permite aplicar configuraciones a través de lo que se denomina como *Playbooks*. Estos *Playbooks* son una especie de scripts en formato *YAML* que describen acciones a realizar en distintos *hosts*. En este caso, el *Playbook* que se encarga de la creación de bases de datos, además crea un usuario para acceder a la misma y le da los permisos necesarios y acceso desde unas subredes determinadas.

Por el contrario, la creación de bases de datos en la infraestructura en la nube de *AWS*, se realiza de forma más tradicional, a través de la extensión *mysqli* de *PHP* [32].

5.2.6. Operaciones en bases de datos

Al igual que se han mencionado algunas operaciones a realizar con contenedores de bastante importancia para los usuarios, con las bases de datos ocurre algo similar.

Algunas de las funciones que los desarrolladores suelen necesitar realizar con respecto a bases de datos, se han automatizado a través de servicios *web* en la *API REST*.

Una de estas funcionalidades es la posibilidad de importar los datos de una base de datos a otra. Los usuarios podrán llevar a cabo esta operación siempre y cuando posean de usuario y contraseña en cada una de las bases de datos, con los permisos necesarios en cada una de ellas, los cuales se les solicitará a través de la conversación correspondiente con el bot de *Slack*.

Otra funcionalidad a tener en cuenta, es la posibilidad de ejecutar modificaciones sobre sus bases de datos. Esto permite a los usuarios, por ejemplo, crear nuevas tablas, modificar datos, etc. Debido a que normalmente, estas modificaciones constan de la ejecución de una cantidad bastante importante de sentencias *Mysql*, no tendría sentido pedir que las introdujeses a través de *Slack*. En su lugar, el bot solicita a los usuarios un repositorio de *Bitbucket*, en el que se buscará en una rama y ruta predefinidas como se describirá más adelante en el manual de usuario.

5.2.7. Comandos de solicitud de información

Existen multitud de comandos implementados para que los usuarios puedan acceder a la distinta información ya sea de contenedores, bases de datos, infraestructura, etc.

Los métodos que se llevan a cabo para recuperar la información dependen del tipo del tipo de información. Por ejemplo, los datos de los distintos elementos que forman parte de la infraestructura creada en la nube de *AWS*, se obtienen empleando la herramienta *AWS CLI*, descrita con anterioridad. Por el contrario, la información correspondiente a contenedores o servicios de contenedores, se obtiene a través de la *API* de *Docker*. Otro tipo de información a la que los usuarios podrían querer acceder, es la correspondiente a las tareas ejecutadas por ellos mismo, lo que se obtiene a través de la *API* de *Jenkins*.

Como se ha mencionado previamente, estos comandos de solicitud de información, no ejecutan tareas propiamente dichas, sino que se limitan a realizar peticiones *HTTP GET* a la *API REST*. Esto es debido a que, al no realizar cambios en el sistema, no hay necesidad de dejar constancia de ellos.

Dado que la información proporcionada puede ser bastante amplia, en este tipo de tareas se aprovecha la posibilidad que ofrece *Slack* de aplicar un formato determinado a los mensajes. Esto se lleva a cabo devolviendo los mensajes al chat de *Slack* en formato *JSON* con unos campos determinados que *Slack* es capaz de interpretar.

A continuación, se muestra un ejemplo de la creación de esta respuesta en formato *JSON* en el bot.

```
controller.hears(['^ *mostrar vpcs(.) *$', '^ *show vpcs(.) *$'],
  ['direct_message', 'direct_mention', 'mention'], function(bot, message) {
  var i;
  var j;
  bot.api.users.info({user: message.user}, function(error, response){
    var user = response.user.name;
    var url = "http://localhost:8082/index.php/vl/aws/ec2/vpcs?user=" + user;
    if(comprobarRespuesta(user, admins) == 0){
      admin = 1;
      if(message.match[1] != ""){
        url = url + "&dep=" + message.match[1].slice(1);
      }
    }
    var XMLHttpRequest = require("xmlhttprequest").XMLHttpRequest;
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.open("GET", url);
    xmlhttp.timeout = 5000;
    xmlhttp.ontimeout = function() {
      bot.reply(message, 'Tiempo de espera de respuesta alcanzado.');
```

```
    };
    xmlhttp.onreadystatechange = function() {
      if(xmlhttp.readyState == 4 && xmlhttp.status == 200) {
        vpcs = JSON.parse(xmlhttp.responseText);
        var resp = '{"text": "Numero de VPCs: ' + vpcs.length + '", "attachments": [';
        for(i=0; i<vpcs.length; i++){
          if(i!=0){
            resp = resp + ',';
          }
          resp = resp + '{"title": "Nombre: ' + vpcs[i].name + '",
            "text": "ID: ' + vpcs[i].id
            + '\\nEstado: ' + vpcs[i].state
            + '\\nBloque Cidr: ' + vpcs[i].cidr
            + '", "fields": [';
          for(j=0; j<vpcs[i].subnets.length; j++){
            if(j!=0){
              resp = resp + ',';
            }
            var cont = parseInt(j) + parseInt(1);
            resp = resp + '{"title": "Subnet ' + cont + ': ' + vpcs[i].subnets[j].name + '",
              "value": "ID: ' + vpcs[i].subnets[j].id
              + '\\nEstado: ' + vpcs[i].subnets[j].state
              + '\\nBloque Cidr: ' + vpcs[i].subnets[j].cidr + '", "short": true}';
          }
          switch(vpcs[i].state){
            case "available":
              resp = resp + '], "color": "good"}';
              break;
            default:
              resp = resp + '], "color": "danger"}';
          }
        }
        resp = resp + ']]';
        bot.reply(message, JSON.parse(resp));
      }
      else if(xmlhttp.readyState == 4 && xmlhttp.status != 200){
        bot.reply(message, "Error cargando subnets.");
      }
    };
    xmlhttp.send();
  });
});
```

Figura 22: Ejemplo de función del bot que devuelve un mensaje en un formato JSON reconocible por Slack. Fuente: Propia

5.2.8. Comunicaciones generales

Una vez explicado el funcionamiento interno de las tareas, sólo queda por explicar el funcionamiento de las comunicaciones que se encargan de transmitir al usuario la información sobre las tareas.

Como se ha descrito con anterioridad, cada tarea de Jenkins realiza una primera petición a la *API REST* antes de realizar la petición que se encargará de la ejecución de la tarea. Esta primera petición es la encargada de enviar al usuario el tiempo estimado de ejecución. El funcionamiento desde el punto de vista del usuario se verá en la sección correspondiente a las pruebas y ejemplos.

El procedimiento que se lleva a cabo para informar al usuario enviar como parámetro a la *API REST*, un parámetro que contiene un identificador de la tarea que se está ejecutando. Una vez recibido el parámetro en la *API REST*, este se utiliza para consultar a través de la *API* de *Jenkins* dicha tarea. El objetivo es obtener dos parámetros. El primero de ellos es el *channel*, el cual se recibió desde el bot de *Slack* y contiene un identificador único del canal de *Slack* desde el que se lanzó la tarea. El siguiente parámetro es *estimatedDuration*. Este parámetro es generado en el servidor *Jenkins* una vez la tarea comienza a ejecutarse. Este tiempo se calcula a partir de las cinco últimas ejecuciones correctas de la tarea en concreto.

A continuación, se muestra el servicio *web* que se encarga de transmitir el tiempo estimado al bot de *Slack*.

```

$app->POST('/v1/jenkins/reply/time', function($request, $response, $args) {

    $idParam = $request->getParsedBodyParam('id');
    $login = getenv("JENKINS_USER");
    $pass = getenv("JENKINS_PASS");
    $host = getenv("JENKINS_HOST");
    $ch = curl_init($host . "/job/" . $idParam
        . "/api/json?tree=estimatedDuration,queueId,building,actions[parameters[name,value]]");

    curl_setopt($ch, CURLOPT_USERPWD, "$login:$pass");
    curl_setopt($ch, CURLOPT_HTTPAUTH, CURLAUTH_BASIC);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    $build = curl_exec($ch);
    curl_close($ch);
    $build_json = json_decode($build);
    $id = $build_json->queueId;
    if($build_json->building == "true"){
        foreach($build_json->actions[0]->parameters as $parameter){
            if($parameter->name == "channel"){
                $channel = $parameter->value;
                break;
            }
        }
        $time = formatTime($build_json->estimatedDuration);
        $resp = "Ejecutando tarea " . $id . ". Tiempo estimado: " . $time . ".\n";
        addHistorial($resp);

        $campos = "channel=" . $channel . "&resp=" . $resp;
        $rangoBots = array((int)getenv("NODEJS_PORT_RANGE_FIRST"),(int)getenv("NODEJS_PORT_RANGE_LAST"));

        $botHost = getenv("NODEJS");
        for($i=$rangoBots[0];$i<=$rangoBots[1];$i++){
            $ch = curl_init("http://" . $botHost . ":" . $i . "/reply");
            curl_setopt($ch,CURLOPT_POST,true);
            curl_setopt($ch, CURLOPT_POSTFIELDS, $campos);
            curl_exec($ch);
        }

        $reply = "Estimación de tiempo enviada.";
    }
    else{
        $reply = "Estimación de tiempo no enviada: Tarea ya terminada.";
    }
    $newResponse = $response->withStatus(200)
        ->write($reply);
    return $newResponse;
});

```

Figura 23: Servicio web que se encarga de transmitir al bot de Slack el tiempo estimado de una tarea. Fuente: Propia

De una forma muy similar, tras la finalización de cada tarea en *Jenkins*, se lanza otra encargada de informar al usuario de dicho evento, la cual, como puede verse a continuación, realiza una llamada muy similar a la API REST.

```

$app->POST('/v1/jenkins/reply', function($request, $response, $args) {

    $idParam = $request->getParsedBodyParam('id');
    $login = getenv("JENKINS_USER");
    $pass = getenv("JENKINS_PASS");
    $host = getenv("JENKINS_HOST");
    $ch = curl_init($host . "/job/" . $idParam . "/api/json?tree=queueId,result,actions[parameters[name,value]]");
    curl_setopt($ch, CURLOPT_USERPWD, "$login:$pass");
    curl_setopt($ch, CURLOPT_HTTPAUTH, CURLAUTH_BASIC);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    $build = curl_exec($ch);
    curl_close($ch);
    $build_json = json_decode($build);
    foreach($build_json->actions[0]->parameters as $parameter){
        if($parameter->name == "channel"){
            $channel = $parameter->value;
            break;
        }
    }
    $id = $build_json->queueId;
    if($build_json->result == "SUCCESS"){
        $resp = "Tarea " . $id . " terminada correctamente.\n";
    }
    else{
        $resp = "Error en la ejecución de la tarea " . $id . ".\n";
    }

    addHistorial($resp);

    $campos = "channel=" . $channel . "&resp=" . $resp;
    $rangoBots = array((int)getenv("NODEJS_PORT_RANGE_FIRST"),(int)getenv("NODEJS_PORT_RANGE_LAST"));

    $botHost = getenv("NODEJS");
    for($i=$rangoBots[0];$i<=$rangoBots[1];$i++){
        $ch = curl_init("http://" . $botHost . ":" . $i . "/reply");
        curl_setopt($ch,CURLOPT_POST,true);
        curl_setopt($ch, CURLOPT_POSTFIELDS, $campos);
        curl_exec($ch);
    }

    $reply = "Usuario informado.";
    $newResponse = $response->withStatus(200)
        ->write($reply);
    return $newResponse;
});

```

Figura 24: Servicio web que se encarga de transmitir al bot de Slack el aviso de la finalización de una tarea. Fuente: Propia

Como vemos en ambos servicios web, para informar al usuario, se llama a un *web service* del bot de Slack, el cual se muestra a continuación.

```

controller.setupWebserver(process.env.port, function(err, webserver) {
    controller.createWebhookEndpoints(webserver);
});

controller.webserver.post('/reply', function(req, res){
    var channel = req.body.channel;
    var resp = req.body.resp;
    bot.say({
        text:resp,
        channel:channel
    });
    res.send();
});

```

Figura 25: Servicio web del bot de Slack para comunicar mensajes a un chat determinado. Fuente: Propia

Como puede verse, el servicio *web* del bot, recibe el identificador del canal de *Slack* donde imprimirá el mensaje, que también recibe. Estos mensajes se imprimirán tanto en el chat de *Slack* del usuario que lanzó la tarea, como un chat grupal del departamento de Infraestructura y Telecomunicaciones de la empresa. De esta forma, a la vez que los usuarios van siendo informados sobre la actividad de sus tareas, esta actividad puede ser monitorizada por los miembros del departamento mencionado.

A continuación, se muestra una captura del chat grupal de monitorización, donde se va teniendo constancia de cada una de las ejecuciones de los usuarios.



Figura 26: Chat grupal del historial de tareas. Fuente: Propia

5.3. Manual de usuario

5.3.1. Creación de contenedores

5.3.1.1. Creación de contenedor en desarrollo

Para crear un contenedor, el comando a emplear en el bot de *Slack* es *crear contenedor*. Este comando inicia una conversación con el bot en la cual se nos pedirán varios datos relevantes.

La primera pregunta que se nos hace, corresponde con el entorno donde deseamos crear el contenedor (a través de los comandos *crear contenedor dev*, *crear contenedor pre* y *crear contenedor pro*, nos saltamos esta pregunta).

La primera elección que debemos hacer es la de la imagen a partir de la cual se creará nuestro contenedor. Debemos elegir la imagen que se adapte a los requerimientos que tendrá nuestra aplicación.

Por ejemplo, si se trata de una aplicación que necesita un servidor *Apache* pero no tiene ningún otro requisito, elegiremos la imagen *apache24-php56*.

Las siguientes preguntas corresponden con datos referentes a nuestra aplicación, como el cliente del proyecto y el código de egreso de este. Además, se nos preguntará por el nombre de nuestra aplicación, a partir del cual se generará el nombre del contenedor que estamos creando.

A continuación, deberemos elegir si queremos que el contenedor tenga acceso de lectura a un directorio remoto concreto. En caso de necesitarlo, deberemos introducir la ruta completa de dicho directorio.

También se nos preguntará si deseamos o no tener acceso de lectura/escritura a un directorio determinado desde el contenedor. Este directorio se creará (en caso de que no exista) en una ruta predeterminada, y sólo podremos elegir la ruta desde este. Por ejemplo, si introducimos *datos/miApp*, el directorio remoto se creará en la ruta `\\ruta_predeterminada\datos\miApp`.

Por último, se nos dará la opción de cambiar las variables de entorno por defecto que se generarán en el contenedor. En caso de querer añadir nuevas variables deberemos contestar *si* y, a continuación, introducir nuestras variables de entorno con el formato `VARI=VALUE1,VAR2=VALUE=2,...`

Tras terminar de contestar todas las preguntas, se nos notificará de la creación de una tarea con un *ID* determinado. Este *ID* podemos emplearlo para consultar en cualquier momento el estado en el que se encuentra la tarea a través del comando *mostrar tarea idTarea*, la cual puede encontrarse en cola, en ejecución, o finalizada.

En el momento en el que la tarea se empieza a ejecutar, se nos comunica un tiempo estimado de esta y, una vez finalizada, se nos volverá a enviar un mensaje informándonos de si ha habido algún error o de si la tarea se ejecutó correctamente.

Si la tarea terminó correctamente, ya tendremos creado nuestro nuevo contenedor y a través del comando *mostrar contenedores* podremos ver el nombre su nombre, que se habrá creado con el formato *entorno-departamento-nombreAplicación-timestamp*, junto con toda la información de utilidad de este.

Entre esta información, es de especial interés la ruta al origen del código, que nos dirigirá al directorio en el cual introduciremos nuestro código y desarrollaremos. También se nos muestran las rutas a los directorios en los que tenemos permisos de lectura o lectura/escritura desde el contenedor, en caso de haber seleccionado las opciones correspondientes durante la creación de este. Hay que tener en cuenta que estos directorios, en caso de no existir previamente, no se crearán hasta arrancar el contenedor a través del comando *arrancar contenedor*.

En la sección de *Binds*, encontramos una línea por cada directorio mencionado anteriormente, cada una con dos rutas separadas por ":". La de la izquierda corresponde con la ruta en la que están montados en el *host* de *Docker* dichos directorios, mientras que la de la derecha es la que más nos interesa, ya que son las rutas de acceso a esos directorios, visibles desde nuestro contenedor, y que, por lo tanto, emplearemos en nuestra aplicación.

Por último, también se nos muestra la *URL* de acceso a nuestra aplicación, en la que podremos ver los cambios que realicemos en el código.

5.3.1.2. Creación de servicio de contenedores en preproducción

Al seleccionar el entorno de preproducción (*pre* al crear un contenedor), lo primero que se nos pide es seleccionar un contenedor ya existente en desarrollo (*dev*), del cual determinará el tipo de contenedor a crear (imagen del contenedor), junto a otros datos como el código de egreso o el nombre de la aplicación.

Para agilizar la elección del contenedor de desarrollo, es preferible emplear directamente el comando *crear contenedor pre* para poder aplicar filtros que simplifiquen el listado de contenedores a elegir.

Lo siguiente que se nos pregunta es si deseamos modificar las variables de construcción predeterminadas *GIT_REPO* (repositorio *.git* de la aplicación) y *GIT_BRANCH* (rama del repositorio). Si necesitásemos alguna otra variable a mayores debido a necesidades especiales de nuestra aplicación, seleccionando *si* podríamos introducir todas las necesarias con el formato *VAR1=VALUE1,VAR2=VALUE2,...*, pero en la mayoría de los casos no necesitaremos modificarlas.

A continuación, en el caso de NO haber modificado las variables de construcción predeterminadas, se nos preguntará por estas (repositorio y rama). Es importante saber que el contenedor que se creará, contendrá el código que se encuentre en ese momento en la rama del repositorio introducido, por lo tanto, deberemos haber subido todos los últimos cambios realizados en nuestra aplicación antes de proceder.

En la última pregunta deberemos elegir entre subir la aplicación a *Amazon Web Services* o no.

En el caso de seleccionar *no*, el contenedor se creará en local. Esta opción es la recomendada para aplicaciones que necesiten tener acceso a directorios remotos locales. Si el servicio se ha creado en *AWS*, tras finalizar la tarea, podremos ejecutar el comando *mostrar info app* para consultar en cualquier momento datos de interés de nuestro servicio.

Para consultar las versiones disponibles de nuestra aplicación de preproducción, podemos utilizar el comando *mostrar versiones*. Este comando también nos mostrará la versión activa en ese momento tanto en el entorno de *pre* como de *pro*.

5.3.1.2. Creación de servicio de contenedores en producción

A través del comando *crear contenedor pro*, podremos crear un servicio de contenedores en producción. En este caso, sólo se nos preguntará por el nombre del servicio de preproducción que queremos pasar a producción, tras haber introducido previamente si dicho servicio era de *AWS* o local.

5.3.2. Arranque de contenedores

Como se describía en la sección referente a la creación de contenedores, el directorio de trabajo (origen del código), así como el de lectura/escritura en el caso de haberlo seleccionado, no se crean, en caso de no existir previamente, hasta arrancar el contenedor.

El comando para llevar a cabo el arranque del contenedor es *arrancar contenedor*. La acción de arrancar el contenedor tiene distintos efectos en función del tipo de contenedor con el que estemos trabajando, el cual viene determinado por la imagen que seleccionamos durante el proceso de creación de este.

Por ejemplo, si disponemos de un contenedor creado a partir de la imagen *apache24-php56*, arrancarlo supondrá lanzar un servidor *Apache*, haciendo accesible la *URL* del contenedor. Por el contrario, si el contenedor fue creado a partir de la imagen *php7-generic*, arrancarlo ejecutará el código *PHP* almacenado en el contenedor, y, tras la ejecución, el contenedor volverá a estar parado.

5.3.3. Eliminación de contenedores

El comando para eliminar un contenedor es *eliminar contenedor*.

Hay que tener en cuenta que eliminar un contenedor NO borra los directorios creados por este, como serían el directorio que contiene el origen del código y el directorio para escritura de datos (este último sólo en caso de haberlo seleccionado durante la creación).

Esto hace que sea posible reutilizar estos directorios con otros contenedores, ya que sus rutas dependen de los datos introducidos durante la creación, de forma que, si al crear un nuevo contenedor elegimos el mismo nombre de aplicación que se introdujo en el contenedor eliminado, el directorio con el origen del código será el mismo.

De forma similar, si introducimos, en caso de necesitarlo, el mismo nombre para el directorio de lectura/escritura, podremos acceder a los ficheros que ya teníamos almacenados.

5.3.4. Ejecutar comandos en los contenedores

A través del comando *ejecutar comando*, es posible ejecutar comandos en los contenedores creados en el entorno de desarrollo.

Una vez que seleccionemos el contenedor en el que se desea ejecutar el comando, introduciremos dicho comando y, tras finalizar la tarea correspondiente, podremos ver la respuesta a través del comando *mostrar tarea IdTarea*.

5.3.5. Clonación de contenedores

La clonación de un contenedor permite levantar un contenedor en un puerto ya ocupado por otro. Esto puede resultar útil en el caso de que se haya modificado una imagen a partir de la cual estaba creado un contenedor, por ejemplo, para añadir un nuevo módulo

de *PHP*. En este caso, en lugar de crear un nuevo contenedor, podría usarse el comando *clonar contenedor*, para que el nuevo contenedor mantuviese el antiguo puerto y ahorrando, así, mucho tiempo.

Cabe mencionar que sólo puede estar arrancado un contenedor en un mismo puerto, por lo que se aconseja eliminar (o al menos parar con el comando *parar contenedor*) el antiguo contenedor tras haberlo clonado.

Una vez finalizada la clonación, al igual que tras una creación, será necesario arrancar el contenedor con el comando *arrancar contenedor*.

5.3.6. Subidas a preproducción y producción

5.3.6.1. Aplicaciones en preproducción

Tras haber creado previamente un contenedor en preproducción a través del comando *crear contenedor pre*, no es necesario volverlo a ejecutar este proceso cada vez que se quiera subir una nueva versión para actualizar la aplicación. En su lugar, podremos hacerlo con el comando *redesplegar aplicacion*.

Este comando nos pide el nombre del servicio que deseamos actualizar, y el entorno en el que queremos realizar la subida, además de si la aplicación se encuentra en local o en la nube de *AWS*.

Al seleccionar el entorno de preproducción, se nos pregunta si deseamos desplegar una nueva versión o una ya existente. Si elegimos desplegar una nueva versión, la aplicación se actualizará con el código existente en ese momento en la rama del repositorio que se introdujo en la primera subida de la aplicación (creación del contenedor en *pre*), así que debemos asegurarnos de disponer en dicha rama de ese repositorio, el código que deseamos desplegar.

Si por el contrario, lo que deseamos es recuperar una versión anterior de nuestra aplicación, optaremos por redesplegar una versión existente y, acto seguido, seleccionaremos la versión deseada.

5.3.6.1. Aplicaciones en producción

Al seleccionar el entorno de producción, se nos mostrarán las versiones disponibles en *AWS* o en el *registry* de imágenes local, para que seleccionemos la que deseamos desplegar. Al listarse las versiones también se nos muestra una indicación de las versiones que se encuentran actualmente activas en cada uno de los entornos.

Una vez realizado el despliegue de una aplicación, si la aplicación estaba en la nube de *AWS*, podemos utilizar el comando *mostrar info app*, para comprobar la versión de la aplicación en los contenedores que contienen la aplicación y, de esta forma, asegurarnos del estado en el que se encuentran los cambios realizados.

También podemos comprobar las versiones desplegadas en cada uno de los entornos en cualquier momento con el comando *mostrar versiones*.

5.3.7. Creación de bases de datos

Podemos crear una nueva base de datos empleando el comando *crear db*. Este comando nos permite crear una base de datos *MySQL* tanto en desarrollo (*dev*), como en preproducción (*pre*).

Lo que se nos pregunta es el nombre de la aplicación para la cual necesitamos la base de datos. A partir del nombre que facilitemos, se asignará un nombre a la base de datos con el formato *appDepartamentoNombreAppDb*. Por ejemplo, si introducimos como nombre de aplicación *miAplic*, la nueva base de datos se creará con el nombre *appDepartamentoMiAplicDb*.

En el caso de querer crear la base de datos en preproducción, se nos preguntará si se desea en *AWS*. En el caso de seleccionar *no*, la base de datos se creará localmente.

También se nos dará la opción de importar los datos de una base de datos que ya se encuentre en desarrollo, de forma que, si seleccionamos *si*, se buscará una base de datos de desarrollo que coincida con el nombre de la nueva para realizar el importado de datos o de sólo el esquema.

Una vez terminada la tarea de creación, deberemos utilizar el comando *mostrar tarea idTarea* para acceder a la salida de la tarea, donde se encontrarán los datos de acceso a la nueva base de datos (*host*, nombre de la base de datos, usuario y contraseña).

5.3.8. Importación de bases de datos

El comando *importar db* nos permite tanto importar todo el contenido de una base de datos a otra, como importar sólo su esquema.

Al ejecutar el comando se nos preguntarán todos los datos (*host*, nombre de la base de datos, usuario y contraseña) de ambas bases de datos y, finalmente, si deseamos importar sólo el esquema o no.

A través de este comando, por motivos de seguridad, no es posible extraer los datos de una base de datos de producción, por lo que la base de datos origen deberá ser de desarrollo o preproducción.

5.3.9. Ejecución de ficheros *sql*

El comando *ejecutar sql* permite ejecutar un fichero *.sql* en una base de datos de *AWS* o local.

Para ello será necesario disponer de un fichero llamado *deploy.sql* en la rama *master* de un repositorio, dentro de la ruta *deploy/scripts/*.

También puede almacenarse en la misma ruta, un fichero llamado *rollback.sql*, el cual podremos ejecutar en una base de datos a través del comando *rollback sql*, de estructura similar al anterior.

5.3.10. Listado completo de comandos

- *crear contenedor*
 - Descripción: Crea un contenedor.
 - Utilización: *crear contenedor*
 - Ejemplo: *crear contenedor*
- *crear contenedor dev*
 - Descripción: Crea un contenedor en desarrollo.
 - Utilización: *crear contenedor dev*
 - Ejemplo: *crear contenedor dev*
- *crear contenedor pre*
 - Descripción: Crea un contenedor en preproducción.
 - Utilización: *crear contenedor pre* [-a aplicación][-c creador][-d departamento]
 - Ejemplo: *crear contenedor pre -a miApp -c nombre.apellido*
- *crear contenedor prp*
 - Descripción: Crea un contenedor en producción.
 - Utilización: *crear contenedor pro*
 - Ejemplo: *crear contenedor pro*
- *eliminar contenedor*
 - Descripción: Elimina un contenedor.
 - Utilización: *eliminar contenedor* [-a aplicación][-c creador][-d departamento] [-e entorno]
 - Ejemplo: *eliminar contenedor -a miApp -c nombre.apellido -e pre*
- *arrancar contenedor*
 - Descripción: Arranca un contenedor.
 - Utilización: *arrancar contenedor* [-a aplicación][-c creador][-d departamento]
 - Ejemplo: *arrancar contenedor -a miApp -c nombre.apellido*
- *parar contenedor*
 - Descripción: Para un contenedor.
 - Utilización: *parar contenedor* [-a aplicación][-c creador][-d departamento]
 - Ejemplo: *parar contenedor -a miApp -c nombre.apellido*
- *clonar contenedor*
 - Descripción: Clona un contenedor.
 - Utilización: *clonar contenedor*
 - Ejemplo: *clonar contenedor*
- *ejecutar comando*
 - Descripción: Ejecuta un comando en un contenedor.
 - Utilización: *ejecutar comando* [-a aplicación][-c creador][-d departamento]
 - Ejemplo: *ejecutar comando -a miApp -c nombre.apellido*
- *mostrar contenedores*
 - Descripción: Muestra los contenedores y servicios de todos los entornos.
 - Utilización: *mostrar contenedores* [-a aplicación][-c creador][-d departamento] [-e entorno] [-t tipo(local|aws)]
 - Ejemplo: *mostrar contenedores -a miApp -c nombre.apellido -e pre -t local*
- *mostrar imagenes*
 - Descripción: Muestra las imágenes disponibles para la creación de contenedores.
 - Utilización: *mostrar imagenes*

- Ejemplo: *mostrar imagenes*
- *crear db*
 - Descripción: Crea una base de datos *mysql* local o en *AWS*.
 - Utilización: *crear db*
 - Ejemplo: *crear db*
- *importar db*
 - Descripción: Importa los datos o sólo el esquema de una base de datos a otra.
 - Utilización: *importar db*
 - Ejemplo: *importar db*
- *ejecutar sql*
 - Descripción: Ejecuta un fichero *.sql* almacenado en un repositorio, en una base de datos.
 - Utilización: *ejecutar sql*
 - Ejemplo: *ejecutar sql*
- *rollback sql*
 - Descripción: Ejecuta un fichero de *rollback* almacenado en un repositorio, en una base de datos.
 - Utilización: *rollback sql*
 - Ejemplo: *rollback sql*
- *redesplegar aplicacion*
 - Descripción: Despliega una nueva versión o una ya existente de una aplicación en preproducción o producción.
 - Utilización: *redesplegar aplicacion*
 - Ejemplo: *redesplegar aplicación*
- *actualizar aplicacion*
 - Descripción: Actualiza las variables de entorno de una aplicación en preproducción o producción.
 - Utilización: *actualizar aplicacion*
 - Ejemplo: *actualizar aplicacion*
- *mostrar info app*
 - Descripción: Muestra información de una aplicación en *AWS*, como la *URL* de acceso y los contenedores existentes.
 - Utilización: *mostrar info app*
 - Ejemplo: *mostrar info app*
- *mostrar versiones*
 - Descripción: Muestra las versiones existentes de un servicio de contenedores.
 - Utilización: *mostrar versiones*
 - Ejemplo: *mostrar versiones*
- *mostrar logs*
 - Descripción: Muestra los logs de una aplicación en *AWS*.
 - Utilización: *mostrar logs*
 - Ejemplo: *mostrar logs*
- *crear instancia*
 - Descripción: Crea una instancia *EC2* en *AWS*.
 - Utilización: *crear instancia*
 - Ejemplo: *crear instancia*

- *eliminar instancia*
 - Descripción: Elimina una instancia *EC2* en *AWS*.
 - Utilización: *eliminar instancia idInstancia*
 - Ejemplo: *eliminar instancia i-xxxxxxx*
- *parar instancia*
 - Descripción: Detiene una instancia *EC2* en *AWS*.
 - Utilización: *parar instancia idInstancia*
 - Ejemplo: *parar instancia i-xxxxxxx*
- *arrancar instancia*
 - Descripción: Arranca una instancia *EC2* en *AWS*.
 - Utilización: *arrancar instancia idInstancia*
 - Ejemplo: *arrancar instancia i-xxxxxxx*
- *reiniciar instancia*
 - Descripción: Reinicia una instancia *EC2* en *AWS*.
 - Utilización: *reiniciar instancia idInstancia*
 - Ejemplo: *reiniciar instancia i-xxxxxxx*
- *redimensionar instancia*
 - Descripción: Cambia el tipo de una instancia *EC2* en *AWS*.
 - Utilización: *redimensionar instancia idInstancia*
 - Ejemplo: *redimensionar instancia i-xxxxxxx*
- *mostrar cpu*
 - Descripción: Muestra el uso de *CPU* en las últimas 24 horas de una instancia *EC2* en *AWS*.
 - Utilización: *mostrar cpu idInstancia*
 - Ejemplo: *mostrar cpu i- xxxxxxx*
- *mostrar instancias*
 - Descripción: Muestra una lista de instancias *EC2* en *AWS*.
 - Utilización: *mostrar instancias [departamento]*
 - Ejemplo: *mostrar instancias inftel*
- *mostrar claves*
 - Descripción: Muestra las claves disponibles en *AWS*.
 - Utilización: *mostrar claves [departamento]*
 - Ejemplo: *mostrar claves inftel*
- *mostrar subnets*
 - Descripción: Muestra las *VPCs* junto con las *subnets* existentes en *AWS*.
 - Utilización: *mostrar subnets [departamento]*
 - Ejemplo: *mostrar subnets inftel*
- *mostrar certificados*
 - Descripción: Muestra los certificados existentes en *AWS*.
 - Utilización: *mostrar certificados [departamento]*
 - Ejemplo: *mostrar certificados inftel*
- *mostrar rds*
 - Descripción: Muestra las instancias *RDS* existentes en *AWS*.
 - Utilización: *mostrar rds [departamento]*
 - Ejemplo: *mostrar rds inftel*
- *redimensionar rds*

- Descripción: Cambia el tipo de una instancia *RDS* en *AWS*.
- Utilización: *redimensionar rds nombreHost*
- Ejemplo: *redimensionar rds miRds*
- *mostrar tareas*
 - Descripción: Muestra una lista de las 10 últimas tareas ejecutadas por el usuario en las últimas 24 horas.
 - Utilización: *mostrar tareas*
 - Ejemplo: *mostrar tareas*
- *mostrar tarea*
 - Descripción: Muestra los detalles de una tarea.
 - Utilización: *mostrar tarea idTarea*
 - Ejemplo: *mostrar tarea 1250*
- *repetir tarea*
 - Descripción: Vuelve a ejecutar la tarea indicada.
 - Utilización: *repetir tarea idTarea*
 - Ejemplo: *repetir tarea 1250*
- *mostrar comandos*
 - Descripción: Muestra los comandos disponibles del bot.
 - Utilización: *mostrar comandos*
 - Ejemplo: *mostrar comandos*
- *ayuda*
 - Descripción: Muestra la ayuda del bot.
 - Utilización: *ayuda*
 - Ejemplo: *ayuda*
- *uptime*
 - Descripción: Muestra el tiempo que lleva levantado el bot de *Slack*.
 - Utilización: *uptime*
 - Ejemplo: *uptime*
- *reiniciar*
 - Descripción: Reinicia el bot de *Slack*.
 - Utilización: *reiniciar*
 - Ejemplo: *reiniciar*

5.4. Pruebas y ejemplos

5.4.1. Introducción

A continuación, se mostrarán a través de capturas y breves explicaciones, algunos ejemplos de los comandos más importantes que los usuarios pueden llevar a cabo a través del bot de *Slack*.

5.4.2. Creación de un contenedor de desarrollo

En la siguiente imagen, podemos ver la conversación que se inicia tras introducir el comando crear contenedor en el chat con el bot de *Slack*.

☆ **bender**
 Messages About Today

eloy.redondo 10:32 AM
 crear contenedor

bender APP 10:32 AM
 Creando contenedor: (introduce "cancelar" para anular la creación)
 Introduce el entorno.
 dev pre pro

eloy.redondo 10:32 AM
 dev

bender APP 10:32 AM
 Cargando imagenes...
 ¿Qué imagen deseas para el contenedor? Introduce su número.
 1. apache24 2. apache24-php56 3. apache24-php56-ti 4. apache24-php56-wkhtmltopdf 5. dts-
 apache24-php56-wkhtmltopdf-generic 6. dts-apache24-php56-wkhtmltopdf-lite-generic 7. dts-apache24-
 php56-generic 8. apache24-php7 9. apache24-php7-wkhtmltopdf 10. dts-apache24-php7-wkhtmltopdf-
 generic 11. dts-apache24-php7-generic

eloy.redondo 10:33 AM
 2

bender APP 10:33 AM
 Imagen elegida: apache24-php56
 ¿Cuál es el cliente de proyecto? (Ej. jazztel)

eloy.redondo 10:33 AM
 etsit

bender APP 10:33 AM
 ¿Cuál es el código del proyecto? (Código agresio)

eloy.redondo 10:33 AM
 123asd

bender APP 10:33 AM
 Introduce el nombre de tu aplicación.

+ prueba-creacion

bold *italics* ~strike~ `code`

```
preformatted
```

 >quote

Figura 27: Ejemplo del uso del comando “crear contenedor” (1). Fuente: Propia

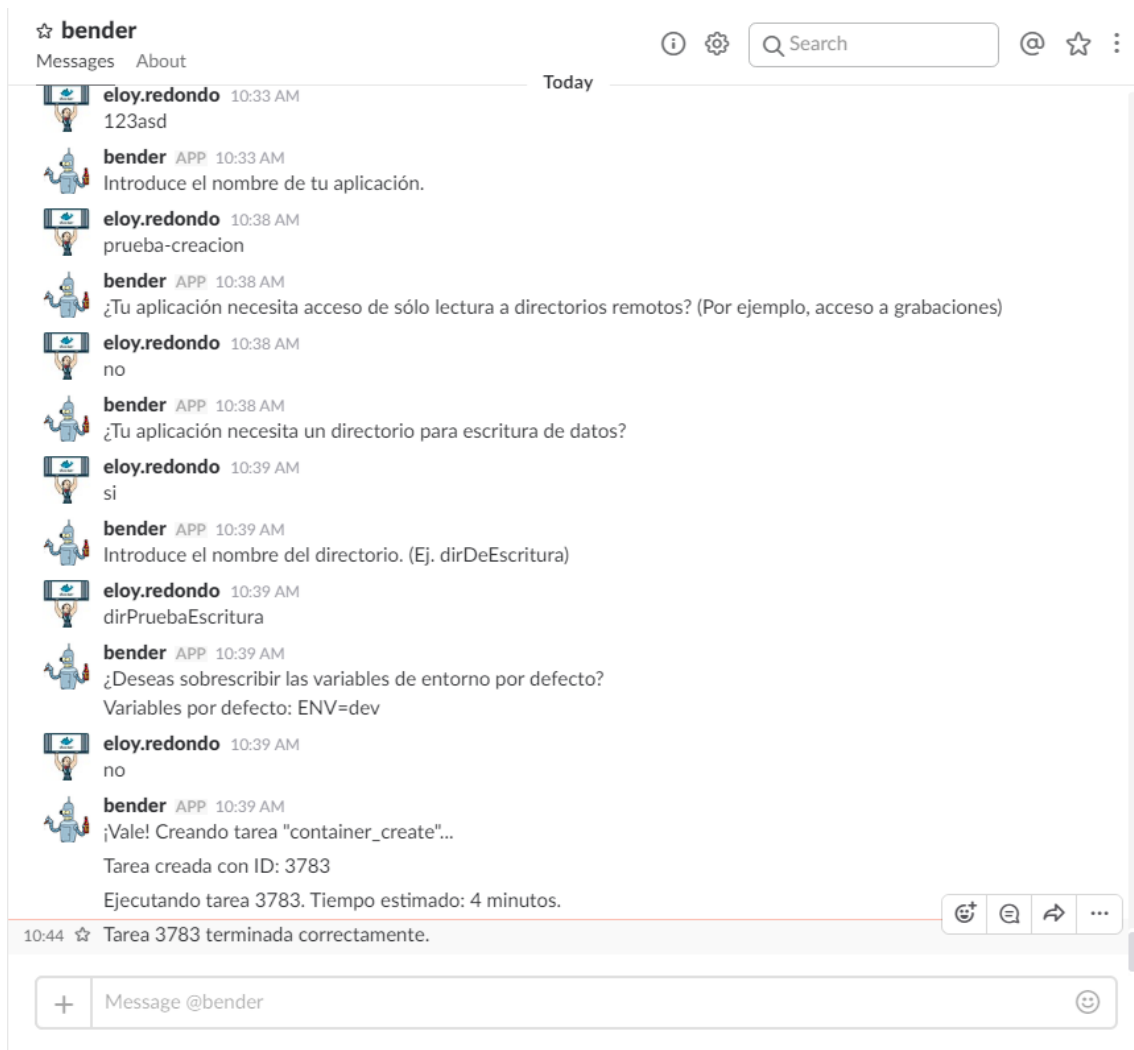


Figura 28: Ejemplo del uso del comando “crear contenedor” (2). Fuente: Propia

En las imágenes anteriores podemos ver al completo toda la conversación que tiene lugar con el bot para llevar a cabo la creación de un contenedor de desarrollo, lo que sería el primer paso a la hora de empezar el desarrollo de una nueva aplicación.


Como puede verse, el proceso tarda alrededor de 4 minutos. La mayor parte de este tiempo corresponde con la búsqueda de un puerto libre en el servidor en el que escuchará el contenedor y la creación del alias en el *DNS* local.


Una vez creado el contenedor, podemos ver sus características a través del comando *mostrar contenedores*, como se muestra a continuación.

☆ bender
Messages About

Today

Tarea creada con ID: 3783
Ejecutando tarea 3783. Tiempo estimado: 4 minutos.
Tarea 3783 terminada correctamente.

 **eloy.redondo** 10:51 AM
mostrar contenedores -d infstel -e dev -a prueba-creacion

 **bender** APP 10:51 AM
Cargando contenedores...
Numero de contenedores: 1

```

Nombre: dev-infstel-prueba-creacion-20170714104407
ID: 0c411de01042c576675ea617b1effb32359ad3fe648d8b05be236b0a1fa74438
Imagen: apache24-php56
Comando: httpd -D FOREGROUND
Estado: created
Código de proyecto: 123asd
Cliente: etsit
Creador: eloy.redondo
Origen del código:
  \\fileapp\Share\wwwdocker\dev\infstel\eloy.redondo\prueba-creacion
Directorio para escritura de datos:
  \\filsrvva01\app.FilSrvVa01\wwwdocker\dev\infstel\dirPruebaEscritura
Acceso a la aplicación:
http://prueba-creacion.dev.infstel.madisonmk.local:9033
Binds:
  /data/remote/dev/infstel/eloy.redondo/prueba-creacion : /var/www/service
  /data/docs/dev/infstel/dirPruebaEscritura : /data/docs
Port bindings:
80/tcp
:9033

```

+ Message @bender

Figura 29: Ejemplo del uso del comando “mostrar contenedores”. Fuente: Propia

Como se puede observar, el bot nos muestra toda la información correspondiente a los contenedores que se listan en función del filtro introducido que, en este caso, ha sido mostrar los contenedores del entorno de desarrollo del departamento de Infraestructura y Telecomunicaciones con nombre de aplicación “*prueba-creacion*”. Como el único contenedor que cumple estos requisitos es el que acabamos de crear, es el único que se muestra.

Cabe mencionar que sólo los miembros del departamento de Infraestructura y Telecomunicaciones de la empresa tienen operativo el uso del filtro “-d” para indicar el departamento, de forma que la gran mayoría de usuarios sólo pueden consultar los datos que correspondan a contenedores creados por los miembros de su propio departamento.

5.4.3. Arranque de un contenedor

Como se ha mencionado con anterioridad, los contenedores de desarrollo es necesario arrancarlos manualmente desde Slack, a diferencia de los servicios de preproducción y producción, que siempre se encuentran arrancados.

Un ejemplo de este proceso se muestra a continuación.

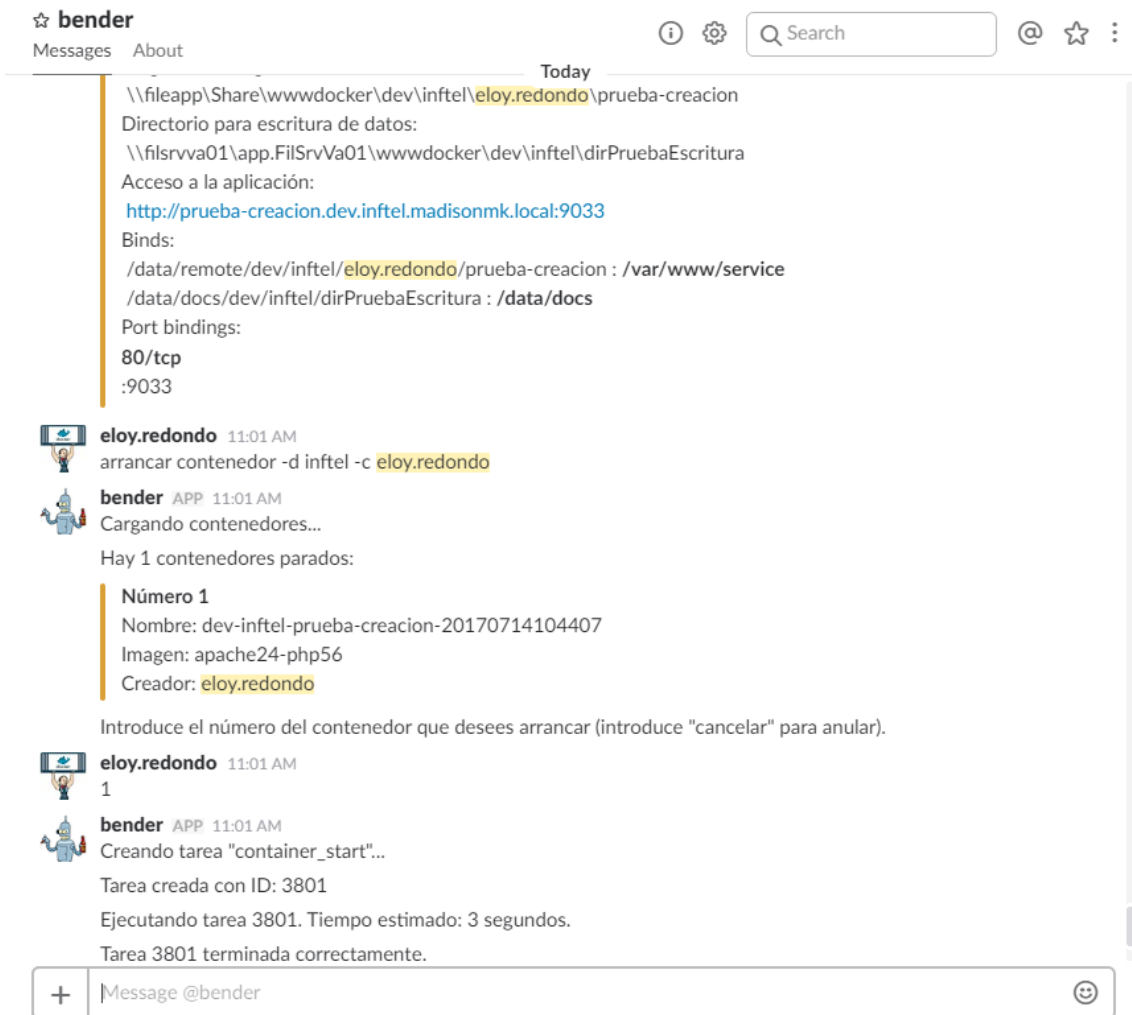


Figura 30: Ejemplo del uso del comando “arrancar contenedor”. Fuente: Propia

En este caso, el comando utilizado es *arrancar contenedor*. De nuevo se emplean los filtros descritos en la sección correspondiente al manual, esta vez, para listar los contenedores que estén parados creados por el usuario *eloy.redondo*. Una vez seleccionado el contenedor que se desea arrancar, que en este caso sólo hay uno para elegir, se ejecuta la tarea.

Si de nuevo mostramos las características del contenedor a través del comando *mostrar contenedores*, podremos observar que ahora el color con el que se muestra la información es el verde en lugar del amarillo, indicando que el contenedor está arrancado.

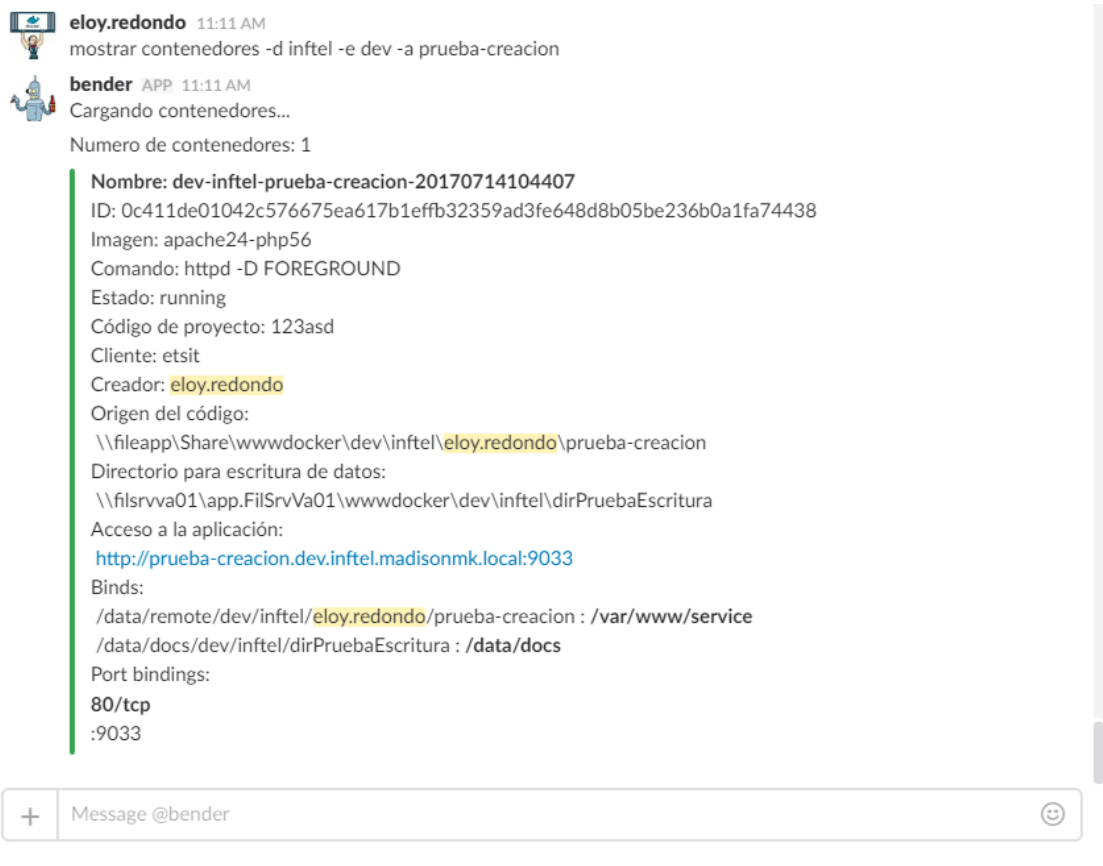


Figura 31: Ejemplo del uso del comando “mostrar contenedores” para listar los contenedores de desarrollo. Fuente: Propia

Una vez arrancado correctamente el contenedor, los usuarios pueden acceder al directorio origen del código que se muestra a través del explorador de archivos de *Windows*, y empezar a desarrollar ahí. Los cambios que vayan guardando en dicho directorio se verán reflejados de forma instantánea en la *URL* que se les facilita.

Como se puede ver en la imagen, también se facilita al usuario la ruta de acceso al directorio de escritura en el caso de que lo hayan solicitado. Fijándonos en la sección *Binds* de la información que se muestra, vemos que para crear ficheros en el directorio de escritura que asignado, hay que hacer que la aplicación los cree de forma interna en el directorio */data/docs* del contenedor.

5.4.4. Creación de un servicio en preproducción

A continuación, se muestra el proceso de creación de un servicio de contenedores en el entorno de preproducción.



Figura 32: Ejemplo del uso del comando “crear contenedor pre”. Fuente: Propia

En este caso, el servicio de contenedores se ha elegido crear en la infraestructura local y no en la nube de AWS. Si mostramos de nuevo los contenedores que contienen la aplicación llamada “*prueba-creacion*”, sin especificar ningún filtro de entorno, esta vez se mostraran tanto el contenedor de desarrollo, como el servicio *Swarm* de preproducción.

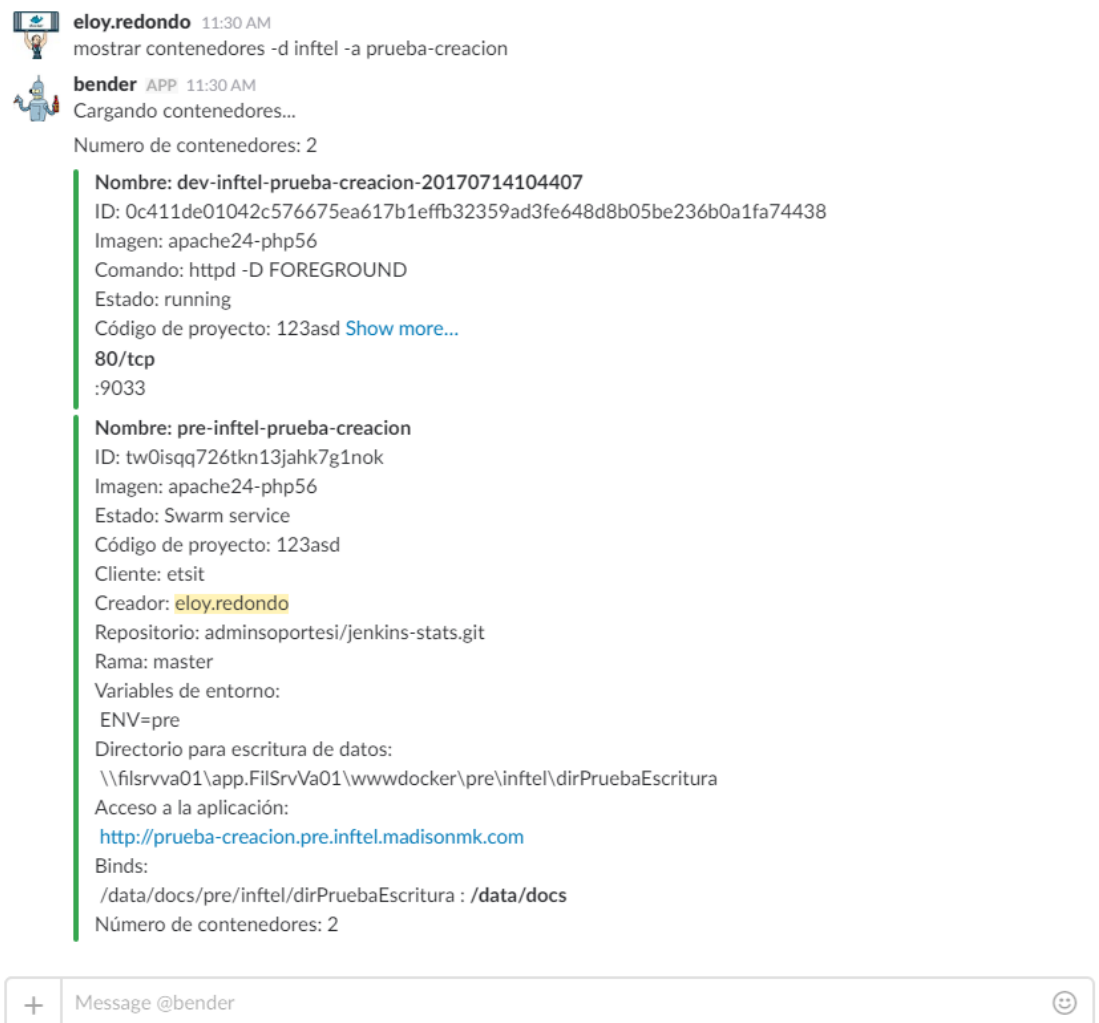


Figura 33: Ejemplo del uso del comando “mostrar contenedores” para listar los contenedores de varios entornos. Fuente: Propia

Como se muestra en la imagen, el número de contenedores que contiene el servicio *Swarm* es 2, que es el número con el que se crean por defecto. De esta forma, las peticiones que se envíen a la aplicación pueden llegar a cualquiera de los contenedores disponibles.

De forma similar a lo que ocurría en el entorno de desarrollo, al usuario se le muestra la *URL* de acceso a la aplicación, así como el directorio de escritura si el usuario lo había solicitado en la creación del contenedor de desarrollo. Sin embargo, no se muestra el *bind* correspondiente al *document root* de la aplicación y, por lo tanto, tampoco el directorio de origen del código. Esto es debido a que, en los entornos de preproducción y producción, el código de la aplicación forma parte de la imagen con la que se crean los contenedores que componen el servicio, por lo que no puede ser modificado.

5.4.5. Redespliegue de aplicaciones

Al contrario que en el entorno de desarrollo, al no ser posible la modificación del código de las aplicaciones en preproducción y producción, se utiliza el comando *redesplegar aplicacion*, para que los usuarios puedan subir nuevas versiones de su aplicación a estos entornos.



Figura 34: Ejemplo del uso del comando “redesplegar aplicacion” para subir una nueva versión. Fuente: Propia

Como puede verse, este comando es muy fácil de usar y permitiendo subir nuevas versiones de las aplicaciones a los usuarios, sin necesidad de solicitarlo al departamento de Infraestructura y Telecomunicaciones, como debían hacer antes de la implementación de este sistema de automatización.

En la conversación con el bot puede verse que una de las preguntas corresponde con el departamento del usuario. Esta pregunta sólo se realiza a los miembros del departamento de Infraestructura y Telecomunicaciones, de forma que estos puedan actualizar aplicaciones de cualquiera de los departamentos.

Es este otro ejemplo, se muestra el caso en el que, en lugar de querer subir una nueva versión de la aplicación, se desea volver a una versión anterior.



Figura 35: Ejemplo del uso del comando "redesplegar aplicacion" para volver a una versión anterior. Fuente: Propia


Puede verse que, en este caso, lo que ocurre es que se muestra una lista con las versiones disponibles, con una indicación de la versión desplegada actualmente en el entorno de preproducción y, de en caso de existir, también en el entorno de producción.


En este ejemplo se ha realizado con una aplicación desplegada en *AWS*, pero en el *Swarm* local se realizaría de forma análoga.


5.4.6. Información sobre los servicios


En la sección correspondiente al manual, se han descrito distintos comandos con los que los usuarios pueden obtener información de sus servicios de contenedores.


En el siguiente ejemplo vemos la utilización del comando *mostrar info app*, el cual muestra la *URL* de acceso a la aplicación, información sobre los contenedores que contiene el servicio y un listado de los últimos eventos del servicio de una aplicación desplegada en *AWS*.


 **eloy.redondo** 12:01 PM
mostrar info app


 **bender** APP 12:01 PM
Introduce el nombre de la aplicación ("cancelar" para anular).


 **eloy.redondo** 12:01 PM
docker-api-interface

 **bender** APP 12:01 PM
Introduce el entorno.
pre pro

 **eloy.redondo** 12:01 PM
pre

 **bender** APP 12:01 PM
¿A qué departamento pertenece la aplicación?
agency bpo digitelts estudios infstel si

 **eloy.redondo** 12:01 PM
infstel

 **bender** APP 12:01 PM
Cargando información...

Acceso a la aplicación:
[ALB-infstel-pre-1-██████████eu-west-1.elb.amazonaws.com:443](#)

Número de contenedores: 2
2017-06-13:15:49:15,30878e94-9efd-42bb-a72b-34684d5d1239,"RUNNING"->"RUNNING";arn:aws:ecs:eu-west-1:176423637738:task-definition/TSK-docker-api-interface-pre:24"
2017-06-13:15:48:12,a17cc4dd-8ed0-4d82-9df5-59b9754f6b96,"RUNNING"->"RUNNING";arn:aws:ecs:eu-west-1:176423637738:task-definition/TSK-docker-api-interface-pre:24"

Eventos del servicio:
Date: 1499610858.118
(service SVC-docker-api-interface-pre) has reached a steady state.

Date: 1499632504.016
(service SVC-docker-api-interface-pre) has reached a steady state.

Date: 1499654116.291
(service SVC-docker-api-interface-pre) has reached a steady state.

Date: 1499675751.353
(service SVC-docker-api-interface-pre) has reached a steady state.

Date: 1499697373.891
(service SVC-docker-api-interface-pre) has reached a steady state.

Date: 1499719012.659
(service SVC-docker-api-interface-pre) has reached a steady state.

Date: 1499740659.084
(service SVC-docker-api-interface-pre) has reached a steady state.

Date: 1499762296.951
(service SVC-docker-api-interface-pre) has reached a steady state.

Date: 1499783921.078
(service SVC-docker-api-interface-pre) has reached a steady state.

Date: 1500000176.627
(service SVC-docker-api-interface-pre) has reached a steady state.

Date: 1500021801.29
(service SVC-docker-api-interface-pre) has reached a steady state.


+ Message @bender 


Figura 36: Ejemplo del uso del comando “mostrar info app”. Fuente: Propia

5.4.7. Ejecución de comandos en un contenedor

Como ya se ha mencionado previamente en este documento, un comando de bastante utilidad para los desarrolladores es *ejecutar comando*.

Este comando del bot permite ejecutar un comando dentro de un contenedor de desarrollo sin necesidad de realizar una conexión *SSH* al contenedor. Esto permite a los desarrolladores ejecutar, por ejemplo, los comandos de *Composer*.


 **eloy.redondo** 2:18 PM
ejecutar comando -d infstel -a prueba-creacion


 **bender** APP 2:18 PM
Cargando contenedores...


Hay 1 contenedores:


Número 1
Nombre: dev-infstel-prueba-creacion-20170714104407
Imagen: apache24-php56
Creador: [eloy.redondo](#)

Introduce el número del contenedor en el que deseas ejecutar el comando (introduce "cancelar" para anular).


 **eloy.redondo** 2:18 PM
1


 **bender** APP 2:18 PM
Introduce el comando a ejecutar.

 **eloy.redondo** 2:18 PM
ls -l /

 **bender** APP 2:18 PM
Creando tarea "container_exec"...

Tarea creada con ID: 3938
Ejecutando tarea 3938. Tiempo estimado: 31 segundos.
Tarea 3938 terminada correctamente.

 **eloy.redondo** 2:19 PM
mostrar tarea 3938

 **bender** APP 2:19 PM
Número de tareas: 1

Tarea 3938
container_exec #7
Fecha: 2017-07-14
Hora: 14:19:04
Usuario: [eloy.redondo](#)
Salida:

```
total 56
drwxr-xr-x  2 root  root   4096 Jul 13 09:19 bin
drwxr-xr-x  3 root  root   4096 Jul 14 11:01 data
drwxr-xr-x  5 root  root   360 Jul 14 11:01 dev
drwxr-xr-x 33 root  root   4096 Jul 14 10:44 etc
drwxr-xr-x  2 root  root   4096 Jun 25 19:52 home
drwxr-xr-x  8 root  root   4096 Jul 13 09:21 lib
drwxr-xr-x  5 root  root   4096 Jun 25 19:52 media
drwxr-xr-x  2 root  root   4096 Jun 25 19:52 mnt
dr-xr-xr-x 974 root  root     0 Jul 14 11:01 proc
drwx----- 4 root  root   4096 Jul 13 09:21 root
drwxr-xr-x  4 root  root   4096 Jul 14 11:01 run
drwxr-xr-x  2 root  root   4096 Jun 25 19:52 sbin
drwxr-xr-x  2 root  root   4096 Jun 25 19:52 srv
dr-xr-xr-x 13 root  root     0 May 17 12:16 sys
drwxrwxrwt  2 root  root   4096 Jul 14 11:01 tmp
drwxr-xr-x 21 root  root   4096 Jul 13 09:21 usr
drwxr-xr-x 24 root  root   4096 Jul 13 09:21 var
```

Estado: Terminada


+ Message @bender 

Figura 37: Ejemplo de uso del comando "ejecutar comando". Fuente: Propia

En esta imagen podemos ver la ejecución del comando `ls -l /`, lo que lista todos los ficheros y directorios que pueden verse desde la raíz del contenedor seleccionado.

5.4.8. Creación de bases de datos

A continuación, se muestra un ejemplo de otro de los comandos de mayor utilidad para los desarrolladores, *crear db*.

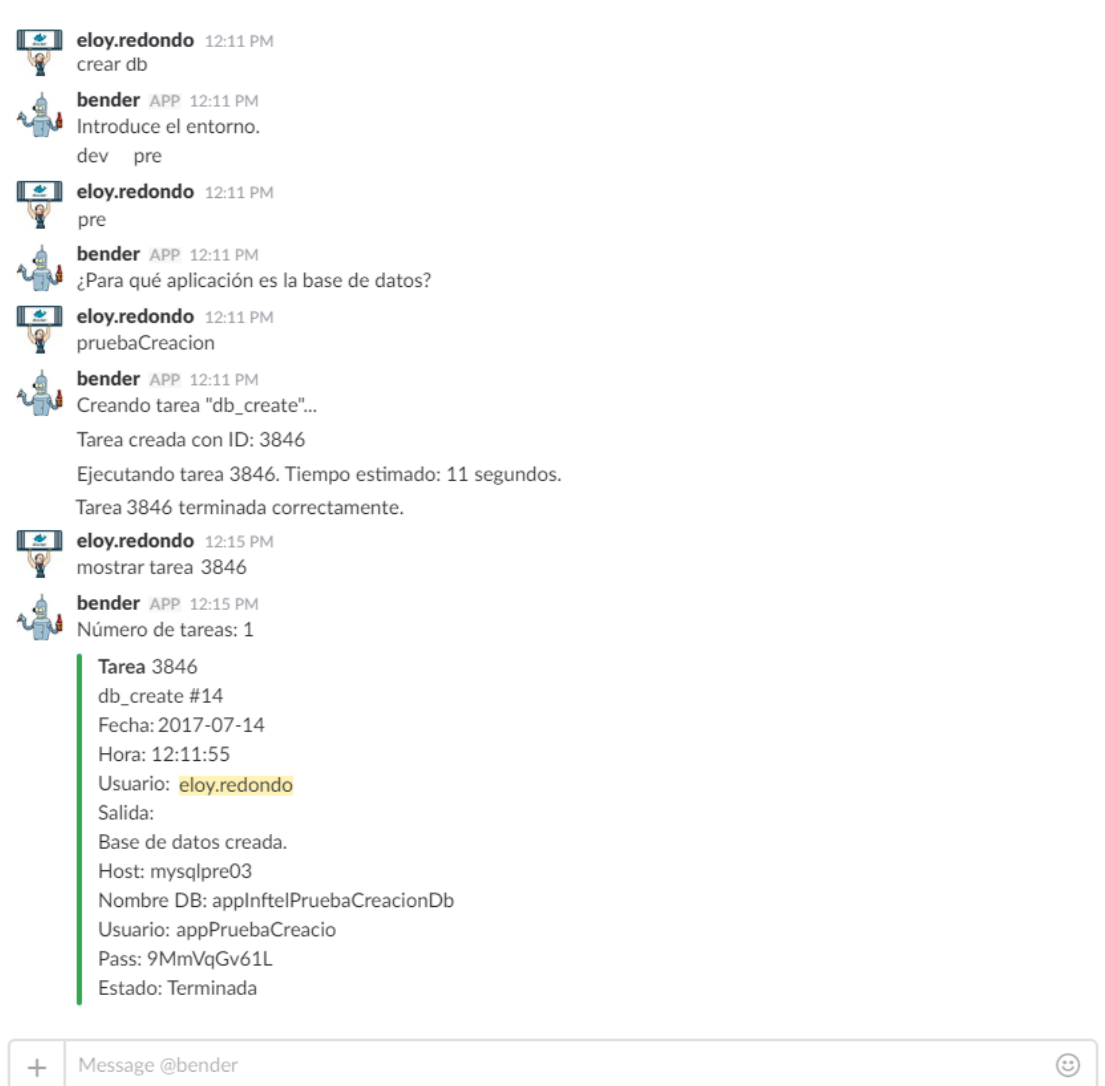


Figura 38: Ejemplo del uso del comando “crear db”. Fuente: Propia

Como se muestra en la imagen, tras crear la base de datos, se emplea el comando *mostrar tarea* para consultar la salida de la tarea y, de esta forma, consultar los datos de acceso a la base de datos.

5.4.9. Importar bases de datos

En la siguiente imagen se muestra un ejemplo del comando *importar db*, el cual permite mover los datos entre bases de datos.



Figura 39: Ejemplo del uso del comando "importar db". Fuente: Propia

Como vemos, tras solicitar al usuario los datos de acceso a cada una de las bases de datos, se ejecuta la tarea. Tareas de este tipo, aunque se informe al usuario con el tiempo

estimado basado en las últimas cinco ejecuciones correctas de la tarea, suelen variar mucho su tiempo de ejecución en función del tamaño de los datos a importar.

Por este motivo, resulta de bastante utilidad el comando *mostrar tarea*, que ya se ha visto en otras ocasiones. Cuando este comando se ejecuta una vez terminada correcta o incorrectamente una tarea, sirve para consultar la salida de dicha tarea.

Por el contrario, cuando *mostrar tarea* se utiliza tras la creación de la tarea, pero antes de su finalización, el comando sirve para ver el estado en el que se encuentra la tarea. De esta forma, un usuario puede comprobar, en el caso de apreciar mucha demora en la finalización de la tarea, si su tarea se encuentra en cola o en ejecución.

A continuación, se muestra un ejemplo en el que se ve la ejecución de una tarea.

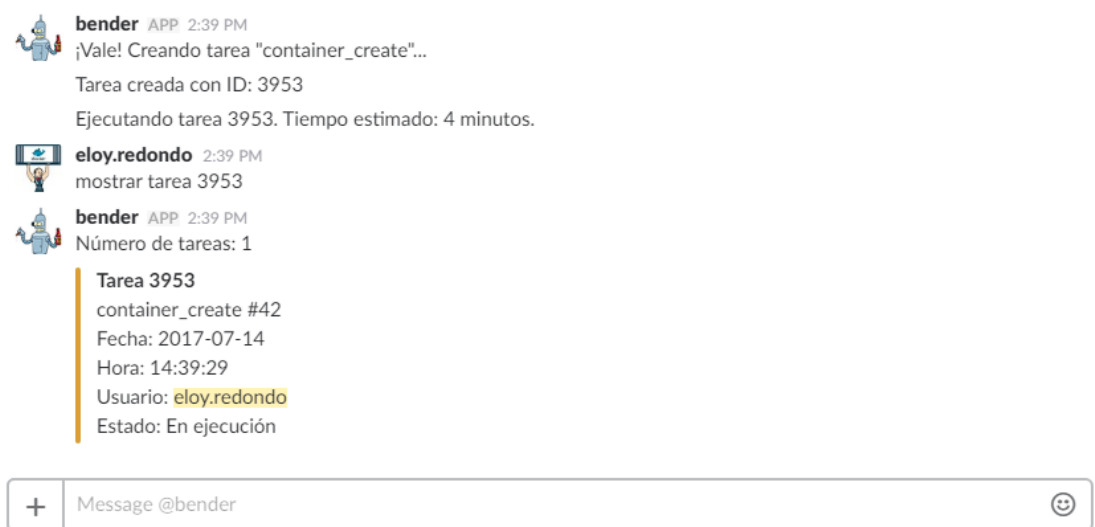


Figura 40: Ejemplo del uso del comando “mostrar tarea” para comprobar el estado de ejecución de una tarea. Fuente: Propia

CAPÍTULO 6
CONCLUSIONES Y LÍNEAS
FUTURAS

CAPÍTULO 6. CONCLUSIONES Y LÍNEAS FUTURAS

6.1. Conclusiones

A lo largo de este documento se ha ido explicando las decisiones que se han tomado y los pasos que se han realizado, con el objetivo principal de unificar los procedimientos llevados a cabo por los distintos equipos de desarrolladores de la empresa *MADISON Experience Marketing*.

Esto, aunque aún en proceso, se puede concluir que se ha conseguido, ya que, tras poco más de medio año desde el inicio de la implementación del nuevo sistema unificado, la gran mayoría de grupos de desarrollo, ya utilizan dicho sistema.

Tras una primera implementación bastante rápida, la necesidad de irse adaptando, en la medida de lo permisible, a cada uno de los proyectos en desarrollo de los equipos, ha hecho que el sistema esté en constante desarrollo, ampliándose y adaptándose cada vez que surge alguna nueva situación que no se había tenido en cuenta inicialmente.

Finalmente, puede concluirse que el éxito de la implantación del nuevo sistema, así como de los nuevos procedimientos a la hora de desarrollar, es el fruto de un gran esfuerzo colectivo. Por un lado, por parte de los miembros del departamento de Infraestructura y Telecomunicaciones que han colaborado en la configuración y migración de sistemas necesarios para hacer viable la implantación del nuevo sistema. Por otro lado, por parte de los grupos de desarrollo que se han adaptado a gestión de sus aplicaciones a través del bot de *Slack*.

A nivel personal, el hecho de haber tenido que trabajar con multitud de herramientas nuevas para mi hasta el momento, me ha hecho progresar muchísimo en lo que a conocimientos se refiere. Por otra parte, el hecho de haber estado trabajando codo con codo junto con mis compañeros de departamento, y a la vez, el hecho de haber necesitado colaborar con los distintos grupos de desarrollo, me ha hecho progresar también como persona, y, en especial, como miembro de un mecanismo mayor.

6.2. Líneas futuras

La implementación del nuevo sistema de automatización es un gran paso hacia la independencia de los grupos de desarrollo que, aunque bajo monitorización, ya tienen la posibilidad de realizar casi de forma totalmente autónoma, todo el proceso de desarrollo de una aplicación, desde su creación, hasta su final puesta en producción.

Sin embargo, para llevar a cabo este proceso, sigue siendo necesaria la ejecución de forma activa de diferentes acciones por parte de alguien. Gracias al nuevo sistema ya no es necesario que esta persona sea un miembro del departamento de Infraestructura y Telecomunicaciones, sino que cualquier desarrollador puede ser esta parte activa a través de la ejecución de comandos a través del bot de *Slack*. De todas formas, el próximo gran avance en el sistema de automatización sería que ciertas acciones se ejecuten de forma autónoma a través de Integración Continua (*CI*).

Esta *CI* se basaría en el despliegue automático de las nuevas versiones de las aplicaciones. Por ejemplo, cada vez que los desarrolladores realicen un cambio en una

determinada rama de su repositorio, se desplegaría la nueva versión en el entorno de preproducción, mientras que, si el cambio se produce en otra rama, el despliegue se realizaría en el entorno de producción.

La Integración Continua, también se basa en realizar muchas otras acciones aparte de las subidas de nuevas versiones de las aplicaciones. Por ejemplo, tras detectar un cambio en el repositorio, antes de subir los cambios al entorno predefinido, la aplicación puede ser desplegada en un contenedor de prueba. De esta forma, se pueden realizar todo tipo de *tests* sobre ese contenedor, como comprobar que no existen errores en el código, o que la aplicación no presenta ningún fallo de seguridad.

Tras la realización de las pruebas pertinentes, el contenedor de prueba se eliminaría y se realizaría la subida de la nueva versión de la aplicación. En punto clave de todo el proceso de *CI* es que todas estas acciones son totalmente transparentes a los desarrolladores, cuya única interacción es la de actualizar las ramas de su repositorio.

Mientras tanto, a pesar de esta menor interacción de los desarrolladores con el sistema, cada nueva versión de las aplicaciones es sometida a un mayor control de calidad, pero de forma automática.

Dado que *Jenkins* permite la implementación de la Integración Continua, ya se están empezando a realizar las primeras pruebas con uno de los equipos de desarrollo. El principal y más grave problema que conlleva la implantación de este nuevo sistema, es que cambia totalmente los procedimientos de los equipos de desarrollo, por lo que resulta muy complicado que este sistema pueda extenderse más allá de a unos pocos grupos concretos.

REFERENCIAS

- [1] MADISON Experience Marketing, «Madison | experience marketing,» 2016. [En línea]. Available: <http://www.madisonmk.com/>. [Último acceso: 14 Julio 2017].
- [2] Slack Technologies, «Slack: Where work happens,» [En línea]. Available: <https://slack.com/>. [Último acceso: 14 Julio 2017].
- [3] Docker Inc., «Docker - Build, Ship, and Run Any App, Anywhere,» 2017. [En línea]. Available: <https://www.docker.com/>. [Último acceso: 14 Julio 2017].
- [4] Docker Inc., «Comparing Containers and Virtual Machines,» 2017. [En línea]. Available: https://www.docker.com/what-container#/package_software. [Último acceso: 14 Julio 2017].
- [5] Docker Inc., «Package software into standardized units for development, shipment and deployment,» 2017. [En línea]. Available: https://www.docker.com/what-container#/package_software. [Último acceso: 14 Julio 2017].
- [6] Docker Inc., «Docker Hub,» 2016. [En línea]. Available: <https://hub.docker.com/>. [Último acceso: 14 Julio 2017].
- [7] Alpine Linux Development Team, «Alpine Linux: index,» 2017. [En línea]. Available: <https://alpinelinux.org/>. [Último acceso: 14 Julio 2017].
- [8] Amazon Web Services, Inc. o sus empresas afiliadas, «AWS | Cloud Computing - Servicios de informática en la nube,» 2017. [En línea]. Available: <https://aws.amazon.com/es/>. [Último acceso: 14 Julio 2017].
- [9] Docker Inc., «Swarm mode key concepts,» 2017. [En línea]. Available: <https://docs.docker.com/engine/swarm/key-concepts/>. [Último acceso: 14 Julio 2017].
- [10] Amazon Web Services, Inc. o sus empresas afiliadas, «¿Qué es Docker? – Amazon Web Services (AWS),» 2017. [En línea]. Available: <https://aws.amazon.com/es/docker/>. [Último acceso: 14 Julio 2017].
- [11] Amazon Web Services, Inc. o sus empresas afiliadas, «Amazon Web Services EC2 - Simple Cloud Hosting,» 2017. [En línea]. Available: <https://aws.amazon.com/es/ec2/>. [Último acceso: 14 Julio 2017].
- [12] Jenkins CI, «Jenkins,» 2017. [En línea]. Available: https://jenkins.io. [Último acceso: 14 Julio 2017].
- [13] SmartBear Software, «Swagger – The World's Most Popular Framework for APIs.,» 2017. [En línea]. Available: <https://swagger.io/>. [Último acceso: 14 Julio 2017].

- [14] <https://www.openapis.org/>, «Open API Initiative,» 2017. [En línea]. Available: <https://www.openapis.org/>. [Último acceso: 14 Julio 2017].
- [15] SmartBear Software, «Swagger Editor,» 2017. [En línea]. Available: <https://swagger.io/swagger-editor/>. [Último acceso: 14 Julio 2017].
- [16] SmartBear Software, «Swagger Specification,» 2017. [En línea]. Available: <https://swagger.io/specification/#info-object-example--20>. [Último acceso: 14 Julio 2017].
- [17] SmartBear Software, «Swagger UI,» 2017. [En línea]. Available: <https://swagger.io/swagger-ui/>. [Último acceso: 14 Julio 2017].
- [18] D. Ho, «Notepad++ Home,» 2016. [En línea]. Available: <https://notepad-plus-plus.org/>. [Último acceso: 14 Julio 2017].
- [19] Slim Framework Team, «Slim Framework - Slim Framework,» [En línea]. Available: <https://www.slimframework.com/>. [Último acceso: 14 Julio 2017].
- [20] PHP Group, «PHP.net,» 2017. [En línea]. Available: <http://php.net/>. [Último acceso: 14 Julio 2017].
- [21] Docker Inc., «Docker Engine API v1.30 Reference - Docker Documentation,» 2017. [En línea]. Available: <https://docs.docker.com/engine/api/v1.30/>. [Último acceso: 14 Julio 2017].
- [22] Amazon Web Services, Inc. o sus empresas afiliadas, «AWS Command Line UI,» 2017. [En línea]. Available: <https://aws.amazon.com/es/cli/>. [Último acceso: 14 Julio 2017].
- [23] XOXCO, Inc., «GitHub - howdyai/botkit: Botkit is a toolkit for making bot applications,» 2017. [En línea]. Available: <https://github.com/howdyai/botkit/>. [Último acceso: 14 Julio 2017].
- [24] Slack Technologies, «Real Time Messaging API | Slack,» [En línea]. Available: <https://api.slack.com/rtm>. [Último acceso: 14 Julio 2017].
- [25] Node.js Foundation, «Node.js,» 2017. [En línea]. Available: <https://nodejs.org/es/>. [Último acceso: 14 Julio 2017].
- [26] «JavaScript,» 2016. [En línea]. Available: <https://www.javascript.com/>. [Último acceso: 14 Julio 2017].
- [27] Jenkins CI, «Jenkins Plugins,» 2017. [En línea]. Available: <https://plugins.jenkins.io/>. [Último acceso: 14 Julio 2017].
- [28] Atlassian, «Bitbucket | The Git solution for professional teams,» 2017. [En línea]. Available: <https://bitbucket.org/>. [Último acceso: 14 Julio 2017].
- [29] The Apache Software Foundation, «Welcome! - The Apache HTTP Server

Project,» 2017. [En línea]. Available: <https://httpd.apache.org/>. [Último acceso: 14 Julio 2017].

[30] N. Adermann y J. Boggiano, «Composer,» 2017. [En línea]. Available: <https://getcomposer.org/>. [Último acceso: 14 Julio 2017].

[31] Red Hat, Inc., «Ansible is Simple IT Automation,» 2017. [En línea]. Available: <https://www.ansible.com/>. [Último acceso: 14 Julio 2017].

[32] The PHP Group, «PHP: MySQLi - Manual - PHP.net,» 2017. [En línea]. Available: <http://php.net/manual/es/book.mysqli.php>. [Último acceso: 14 Julio 2017].