Infraestructura DevOps

PID_00266617

Joan Caparrós Ramírez
Lorenzo Cubero Luque
Jordi Guijarro Olivares

Tiempo mínimo de dedicación recomendado: 7 horas

Universitat Oberta de Catalunya

Joan Caparrós Ramírez

Ingeniero superior en Informática por la UAB, máster en Seguridad de las TIC por la UOC y máster en Diseño y programación de aplicaciones móviles por La Salle. Desarrollador Fullstack web & mobile especializado en entornos de alto rendimiento y seguridad. Actualmente desarrolla funciones de técnico líder de proyectos en el Área de Cálculo y Aplicaciones en el Consorci de Serveis Universitaris de Catalunya (CSUC).

https://www.linkedin.com/in/joan-caparros

Lorenzo Cubero Luque

Ingeniero superior en Informática por la UPC y máster en Gestión de las tecnologías de la información por La Salle. Ha desarrollado proyectos de implantación de metodologías ágiles en equipos DevOps. Actualmente lidera el equipo responsable de los servicios TI de una multinacional suiza dedicada al marketing digital, Netcentric AG.

@lj_cubero

https://www.linkedin.com/in/lorenzocubero

Jordi Guijarro Olivares

Ingeniero en Informática por la UOC y máster en Gestión de tecnologías de la información por la URL. Coordinador de operaciones y ciberseguridad tecnológica en el CSUC, donde lidera las actividades técnicas en los ámbitos de servicios *cloud* y ciberseguridad del consorcio. También coordina el equipo de ciberseguridad CSUC-CSIRT de la red académica y de investigación catalana (Anella Científica), y colabora en proyectos a nivel europeo en grupos de trabajo como SIG-CISS y la TF-CSIRT de la red académica y de investigación europea GéANT.

@jordiguijarro

https://es.linkedin.com/in/jordiguijarro

La revisión de este recurso de aprendizaje UOC ha sido coordinada por el profesor: Josep Jorba Esteve (2019)

Segunda edición: septiembre 2019

Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

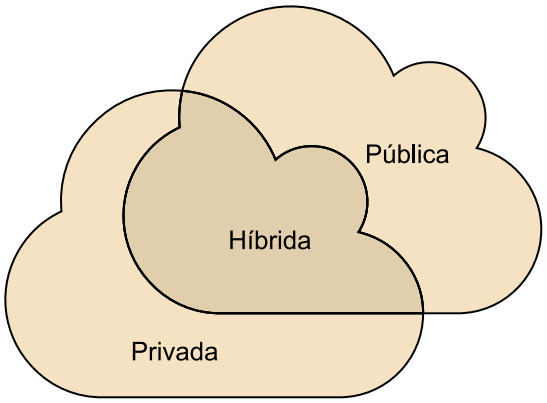
Índice

Objetivos

Formar al estudiante en los principales conceptos avanzados de DevOps y mostrar cómo esta práctica fomenta y desarrolla los principios del *cloud computing* para el uso de infraestructuras y el despliegue de aplicaciones en condiciones de integración continua y plataformas colaborativas.

1. Un repaso de modalidades del *cloud*

Figura 1. Conceptualización de las distintas formas del *cloud*



El *cloud* comprende el concepto básico por el que definiremos la entrega de servicios informáticos a clientes o usuarios por medio de una red. Este nuevo modelo de prestación de servicios permite añadir una capa de abstracción frente a los clientes que no saben dónde estos están ubicados (normalmente alojados en varios proveedores y repartidos por todo el mundo) ni la gestión de recursos que usan. Los servicios en la nube atienden las peticiones recibidas y aportan una flexibilidad y adaptabilidad de recursos frente a la demanda de forma totalmente transparente.

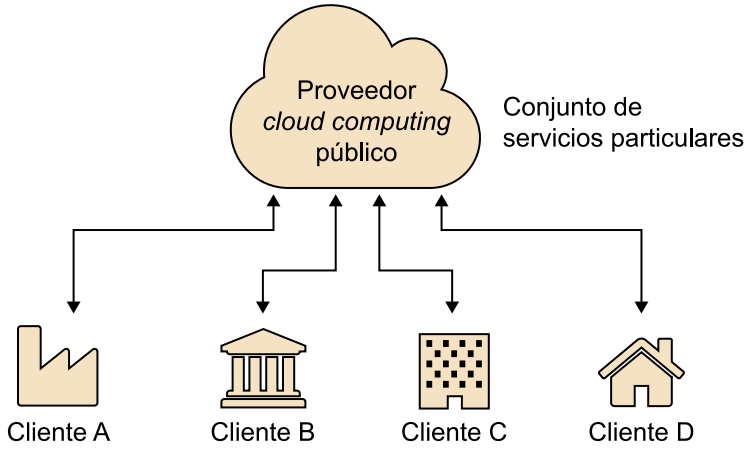
Dependiendo de la ubicación de los servidores clasificaremos el concepto de *cloud* en: *cloud* público, *cloud* privado y *cloud* híbrido.

1) *Cloud* público

El *cloud* público es la forma básica y común de implementar los servicios en la nube. Los servicios informáticos en el caso del *cloud* público son ofrecidos mediante servidores y almacenamiento ubicados en proveedores externos a la red interna de la empresa, lo cual da lugar a que la administración y el servicio se ofrezcan íntegramente mediante el uso de Internet.

Los recursos en un *cloud* público son compartidos con otras organizaciones que hacen uso de la red de recursos suministrados a la nube del proveedor, como en el caso de un *hosting* compartido, haciendo uso de la capacidad de cómputo y almacenamiento dentro de lo contratado y disponible para todos los servicios albergados.

Figura 2. Estructura de un *cloud* público



Las ventajas del *cloud* público son:

- **Abaratamiento de costes**: el pago solo del servicio usado y el hecho que no sea necesario adquirir recursos propios representan un ahorro considerable frente a otras opciones.

- **Simplicidad**: los proveedores de servicio se encargan de la actualización y el mantenimiento de los recursos hardware, con lo cual nos olvidamos de los procesos tradicionales enfocados en la complejidad de la provisión de recursos físicos y nos centramos en el servicio suministrado.

- **Escalabilidad**: los recursos en los proveedores externos aportan volúmenes de recursos casi ilimitados, y dan la posibilidad de ejecutar procesos de autoescalado en cuyos recursos destinados se expanden de forma automática para hacer frente a puntas de peticiones y/o necesidades.

- **Inmediatez**: los grandes proveedores del *cloud* aportan rapidez de implementación y despliegue dentro de sus infraestructuras, lo que permite que se adapten al ciclo de vida breve de las aplicaciones de hoy en día.

- **Disponibilidad**: se garantiza una alta disponibilidad del servicio mediante una amplia red de servidores situados en varios proveedores de nube y en diferentes localizaciones, disminuyendo así la ratio de caídas.

Enlace de interés

Si queréis saber más sobre las ventajas del *cloud* público, podéis consultar el siguiente enlace: <https://www.claranet.es/blog/ventajas-del-cloud-publico-en-3-minutos>.

Ejemplos de servicios en *cloud* público

Ejemplos de servicios usualmente en *cloud* público son: Office en línea, Microsoft Azure, servicios de almacenamiento online (Dropbox, Google Drive), servicios de correo (Gmail, Hotmail...), etc.

2) *Cloud* privado

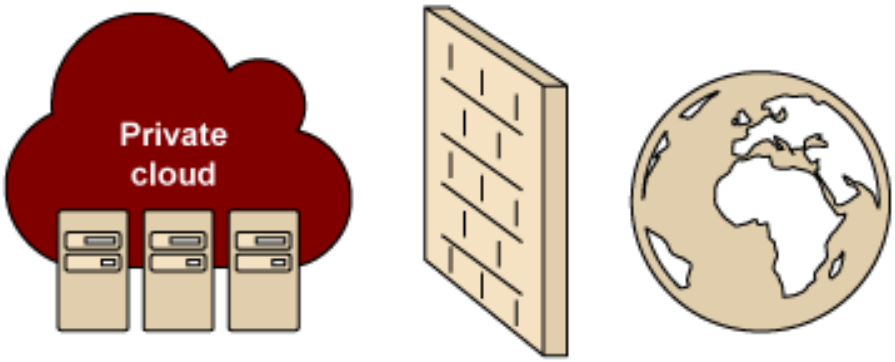
Un *cloud* privado representa la modalidad de la plataforma *cloud* basada en un entorno seguro de red privada utilizado por clientes específicos y no abiertos a todo el mundo como en el caso del *cloud* público.

Este tipo de nube interna aporta a las empresas las ventajas mencionadas en el *cloud* público, tales como escalabilidad e inmediatez, y dado que los procesos se ejecutan dentro de un entorno virtualizado suministrado por los recursos propios de la red privada se añade la posibilidad de personalización y control de los recursos dedicados a la infraestructura.

Cabe mencionar que la modalidad de *cloud* privado ofrece un mayor grado de privacidad y seguridad debido a que los recursos son solo compartidos por clientes específicos bajo medidas de seguridad añadidas, como *firewalls* o VPN, para garantizar el aislamiento de los recursos y la confidencialidad de las comunicaciones realizadas interna o externamente a la nube.

Los costes de las nubes privadas requieren del mismo gasto de mantenimiento, actualización y administración de los centros de datos tradicionales propios.

Figura 3. Estructura de un *cloud* privado



Ventajas del *cloud* privado:

- **Seguridad y privacidad**: la infraestructura utilizada es aislada para el uso exclusivo de clientes u organizaciones específicas, dotando a la nube de un mayor grado de privacidad y seguridad frente a los *clouds* públicos, donde todos los clientes acceden al mismo conjunto de recursos. Una de las técnicas básicas de aislamiento es el uso de *firewalls* para construir barreras físicas al acceso de los recursos y comunicaciones internas de la organización.

Ejemplos de servicios en *cloud* privado

Algunos ejemplos de servicios en *cloud* privado son: Intranet de entidades financieras, almacenamiento de datos *big data* y el Internet de las cosas (IoT), virtualización de entornos (VMware)...

- **Control de los recursos**: los recursos utilizados en esta modalidad de *cloud* son propias a la organización. Así, pues, esta tiene el control para escalar los recursos disponibles adaptándose a las necesidades que vayan surgiendo; por el contrario, la organización tiene que hacerse cargo del mantenimiento y la administración de la solución tomada.

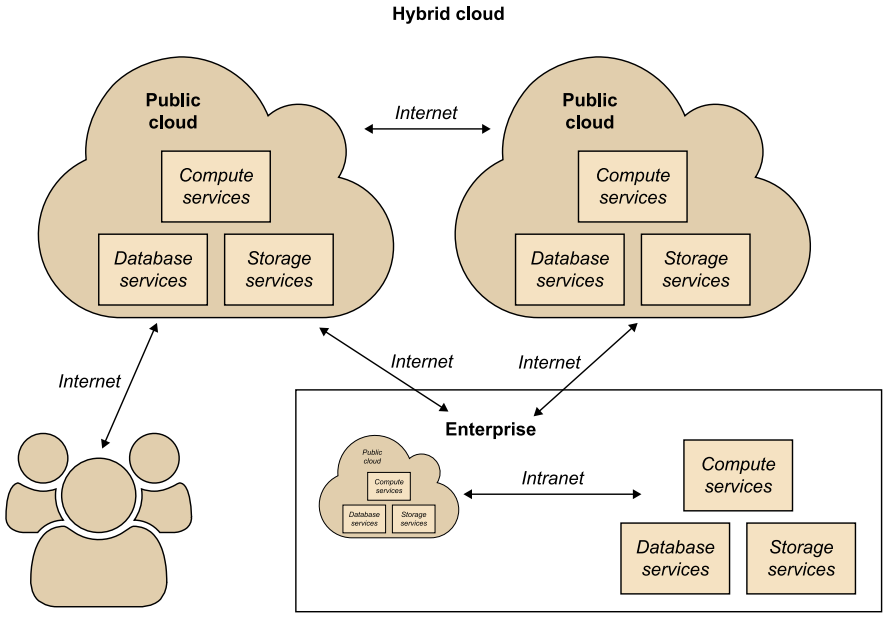
3) *Cloud* híbrido

Este es un servicio de *clouds* mixto que explota el uso de *clouds* tanto privados como públicos, una solución flexible que amplía nuestra infraestructura y nos permite administrar servidores dedicados y servidores *cloud* desde un mismo

panel de control. Los servicios en este tipo de *cloud* son ofrecidos de forma que puedan aprovecharse de las ventajas que cada uno de los *clouds* integrados, maximizando así la eficiencia, el ahorro y la escalabilidad.

Los *clouds* híbridos son la solución perfecta para proyectos que requieren infraestructuras adaptables a la demanda para una parte (por ejemplo, *frontend* de una web), manteniendo los recursos críticos en un *cloud* privado (por ejemplo, base de datos).

Figura 4. Estructura de un *cloud* híbrido



Enlaces de interés

Si queréis saber más sobre el *cloud* híbrido, podéis consultar los siguientes enlaces: <https://www.netapp.com/es/info/what-is-hybrid-cloud.aspx> y <https://www.interoute.es/cloud-article/what-hybrid-cloud>.

Las ventajas del *cloud* híbrido son un compendio de las ventajas del *cloud* público y privado, y las más remarcables son las siguientes:

- **Abaratamiento de costes**: los costes son adaptables a los costes basados en la infraestructura propia más los costes del uso de los recursos en un *cloud* público.

- **Escalabilidad**: los recursos son flexibles más allá de la propia infraestructura de la organización.

- **Seguridad y privacidad**: los recursos críticos son aislados de la infraestructura para una mayor seguridad y se sitúan en el *cloud* privado, limitando así el acceso a ellos.

- **Control de los recursos**: una infraestructura híbrida dota de un control total sobre todas las partes situadas en el *cloud* privado y una administración delegada para determinados puntos críticos frente a altas demandas.

- **Flexibilidad**: los proveedores puntualmente pueden suministrar una capacidad por encima de la del nivel contratado mediante el traslado de de-

Ejemplos de servicios en *cloud* híbrido

Como ejemplos de servicios en *cloud* híbrido podemos mencionar: servicios de entidades bancarias, médicas online, gestiones universitarias o conocidos servicios de *streaming* (Netflix).

terminadas funciones a un *cloud* público. Este servicio es conocido con el nombre de *bursting* y permite liberar recursos dentro de la infraestructura privada para procesos más críticos.

4) *Multicloud*

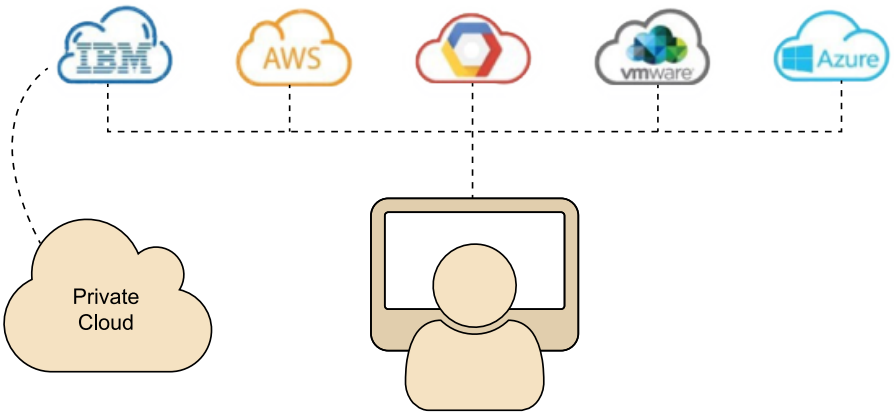
El *multicloud* o nube múltiple representa la implementación de la nube por distintos proveedores o de diversas soluciones dentro del *cloud* privado con una integración y administración bajo un solo panel de control.

Las ventajas comentadas en los distintos *clouds* son ampliadas aumentando la flexibilidad, la escalabilidad, la seguridad y la agilidad para hacer frente a las distintas necesidades que la organización requiera. Tomando en consideración que los distintos proveedores aportan diferentes precios sobre el uso del cómputo o almacenamiento, nos permiten un ahorro mediante el uso eficiente de recursos en proveedores determinados.

Respecto a la seguridad, esta se ve incrementada al no distribuir los datos y no poseerlos todos en un mismo lugar, mitigando el efecto de ataques de DoS u otros que solo afectarían a un único proveedor a la vez.

A nivel funcional no detectaremos muchas diferencias de un *cloud* híbrido más allá de la múltiple utilización tanto de proveedores públicos como privados.

Figura 4. Estructura de un *multicloud*



Enlaces de interés

Si queréis saber más sobre el *multicloud*, podéis consultar los siguientes enlaces: <https://es.slideshare.net/mariojosevillamizarcano/cloud-computing-oportunidades-para-empresarios-y-emprendedores> y <https://www.redhat.com/es/topics/cloud-computing/what-is-multicloud>.

Las ventajas del *multicloud* son las siguientes:

- **Optimización de costes**: encontrar una buena distribución de cargas en los distintos proveedores repercutirá a la baja sobre los costes del servicio.

- **Descentralización**: varios proveedores significan varios lugares de almacenamiento y servicio, añadiendo un plus de disponibilidad frente a posibles afectaciones de servicios de uno de ellos.

- **Seguridad:** los ataques a los servicios quedan mitigados, ya que estos no se encuentran bajo un solo punto; así, la única forma de denegar el servicio consistiría en un difícil ataque global a cada uno de los proveedores que conforman el *multicloud*.

1.1. Plataformas de gestión de la nube

Las plataformas de gestión de la nube (*Cloud Management Platforms*, CMP) son productos que proporcionan una plataforma para la gestión de entornos tanto en *clouds* públicos, privados, híbridos como *multicloud*. Los servicios ofrecidos por un CMP varían según el proveedor, pero todos ellos integran en un solo entorno un conjunto de herramientas software destinadas al autoprovisionamiento y la gestión del funcionamiento de nuestro *cloud*.

Las capacidades que debería incluir una plataforma de gestión en la nube según la CSCC (Cloud Standards Customer Council) son las siguientes:

- Gestión de accesos y autorizaciones: permite el control de acceso basado en políticas sobre los recursos de la nube y ofrece funciones de seguridad, como la encriptación o gestión de usuarios.
- Gestión de recursos en diferentes entornos: la monitorización de los servicios permitirá al administrador garantizar los requisitos de rendimiento y disponibilidad.
- Gestión financiera relacionada con los servicios en la nube suscritos.
- Integración con los entornos relevantes de la nube y los sistemas internos y externos de la empresa.
- Catálogos de servicios para admitir el aprovisionamiento de autoservicio o la aprobación de recursos.
- Gestor en la nube: guía basada en reglas para decisiones de colocación de activos

Las plataformas de gestión de la nube pueden ser específicas para cada proveedor en la nube o serlo por medio de otro proveedor. Remarcaremos que por lo general las implementaciones de las plataformas de gestión para *multicloud* estarán implementadas por un tercero.

Enlace de interés

Si queréis saber más sobre las plataformas de gestión de la nube, podéis consultar el siguiente enlace: <https://www.omg.org/cloud/deliverables/CSCC-Practical-Guide-to-Cloud-Management-Platforms.pdf>.

2. Fundamentos avanzados de DevOps

2.1. Introducción

Una vez vistos los conceptos básicos sobre DevOps, entramos en esta sección en la materia necesaria para tener una visión más avanzada sobre el enfoque del rol de DevOps.

- Uno de los puntos en los que centraremos nuestra atención es la **gestión de la configuración**, que nos permitirá acercar todos los entornos que utilizaremos para que sean lo más parecidos posible al entorno de producción final.

- Uno de los pilares de DevOps es la **integración continua**, que permite agregar los pequeños cambios realizados por los desarrolladores al conjunto de software desarrollado de manera que puedan ser testados y desplegados en entornos de desarrollo lo antes posible.

- Veremos también diferentes **estrategias de *testing***, todas ellas con el objetivo de minimizar los posibles errores que puedan producirse en los entornos productivos. Asimismo, se intentan detectar los errores en etapas tempranas del desarrollo cuando su corrección es más sencilla y menos costosa.

- Como continuación de la integración continua veremos el concepto de **entrega continua** en su aplicación más práctica, donde aprovecharemos para explicar un extenso ejemplo.

Como consecuencia de los conceptos anteriores, también veremos cómo afrontar los siguientes retos:

- La **gestión de infraestructura** y de los diferentes **entornos** resultado de las distintas etapas de la entrega continua. Tanto la infraestructura como los diferentes entornos deben ser extremadamente flexibles para adaptarse rápidamente a los cambios que se puedan producir en las diferentes etapas del desarrollo.

- La **gestión de los datos** también representa un reto en cuanto que deben ser persistentes y estar disponibles a pesar de que el software que da acceso a ellos va cambiando constantemente.

Cabe mencionar también que durante este capítulo está en el ánimo de los autores que se vayan examinando herramientas específicas y ejemplos donde el lector pueda ir viendo aplicaciones directas de los conceptos vistos.

2.2. Gestión de la configuración

En los entornos empresariales actuales podemos encontrar distintas formas de operar.

Algunas poseen máquinas físicas donde estas ejecutan el rol de servidor y ofrecen acceso a las aplicaciones desarrolladas. Esta forma de disposición de servidores es una visión clásica y simple, muy utilizada durante años, con obvias dificultades durante procesos habituales de migraciones de entorno y cambios o ampliaciones exigidos en el hardware utilizado.

Una visión más moderna, con una perspectiva más amplia sobre los sistemas, intentaría desvincular las máquinas físicas de los entornos de despliegue de servidores, haciendo que estos sistemas sean más flexibles mediante la utilización de herramientas de virtualización, como VMware, Xen, KVM, VirtualBox, entre otras.

Hemos descrito distintas formas de visión sobre los entornos de producción y cómo se pueden presentar, pero ¿y los equipos de los entornos de desarrollo? Imaginaos un equipo donde cada miembro trabaja en su propio ordenador, algunos utilizando portátiles, otros con equipos de sobremesa, con configuraciones diferentes entre ellos, incluso con sistemas operativos distintos, ¿no cabría esperar problemas derivados de esta diversidad?

Para la filosofía DevOps, los entornos de despliegue, tanto en desarrollo como en producción, toman una especial importancia. No solo es importante poseer un entorno de servidor funcional donde desplegar una aplicación; como hemos visto, muchos de los problemas durante la puesta en producción del software no derivan de fallos durante el proceso de desarrollo, sino que se presentan durante el despliegue en entornos donde estos difieren en especificaciones: distintos sistemas operativos, versiones de software y paquetería, configuraciones de seguridad, etc.

Para solucionar esta necesidad de homogeneización de entornos aparecen herramientas como Chef y Puppet, que nos permitirán que los sistemas utilizados por el equipo de desarrollo sean exactamente iguales a los que se utilizarán *a posteriori* durante el proceso de despliegue en entornos de producción. De esta manera se eliminará cualquier error de incompatibilidades y los entornos utilizados habrán sido testados durante las primeras fases del desarrollo del proyecto.

Estas soluciones implican un cambio de concepto de todos los equipos implicados en el desarrollo y despliegue de un proyecto, ya que la coordinación para la construcción de un sistema inicial será una tarea compartida entre los equipos de diseño de software y los equipos de operaciones encargados de la gestión de máquinas y sistemas.

Existe una multitud de herramientas especializadas en la gestión de la configuración donde los DevOps pueden apoyarse, entre las que destacan Puppet, Chef y Ansible. En esta sección explicaremos algunos ejemplos introductorios para la utilización de estas herramientas enfocadas a la disposición de entornos homogéneos desarrollo-producción.

2.2.1. Ansible

Ansible es una plataforma de software libre (GPLv3) desarrollada en python y ofrecida comercialmente por AnsibleWorks. Brinda una forma simple de automatización de procesos TI. Puede configurar sistemas, realizar despliegues de software y efectuar tareas avanzadas de despliegues continuos.

Enlace de interés

Para saber más sobre Ansible, podéis consultar el siguiente enlace: <https://www.ansible.com/how-ansible-works>.

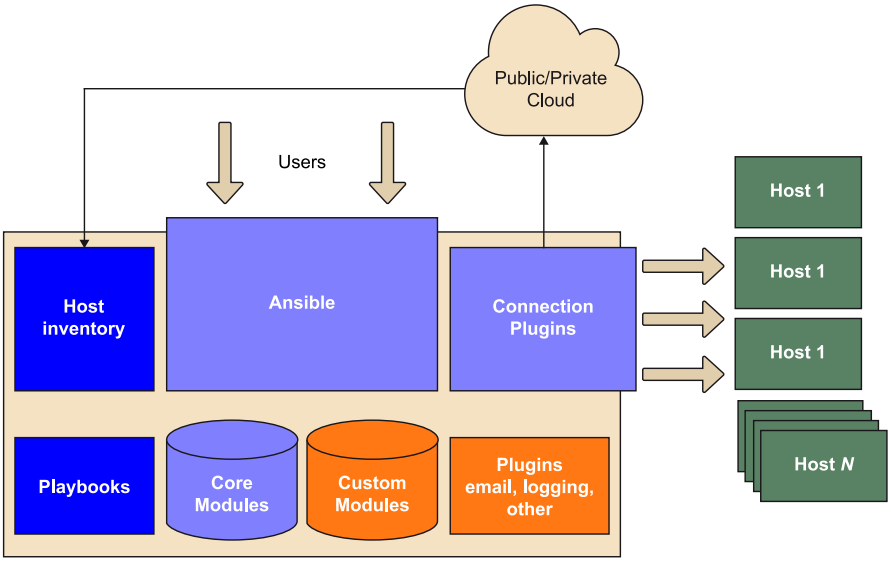
Estas son quizá sus características más importantes:

- Excelente rendimiento sin necesidad de instalación ni despliegues de agentes.
- Bajo coste operativo (*overhead* muy bajo).
- Método de autenticación por ssh (preferiblemente con claves) en paralelo.
- No necesita usuario root (permite la utilización de sudo).
- Permite utilizar comandos básicos.
- Acepta comandos en casi cualquier lenguaje de programación.

Ansible puede utilizarse en modo Ad-Hoc o mediante playbook. El modo Ad-Hoc nos permitirá ejecutar comandos en una sola línea sobre todos los *hosts*; este método es utilizado cuando se pretenden realizar tareas simples sobre todos nuestros *hosts*, mientras que para tareas más complejas los playbooks resultarán mucho más cómodos.

Los playbooks son ficheros escritos en formato YAML, que pueden presentarse en forma de un solo fichero o siguiendo un modelo estructurado, y contienen todos los parámetros necesarios para realizar una determinada tarea sobre un grupo de servidores.

Figura 2. Flujo de datos en Ansible



Fuente: <https://terry.im/wiki/terry/Ansible.html>.

2.3. Integración continua

El proceso de integración continua (*continuous integration*, CI) es un modelo propuesto por Martin Fowler, científico de ThoughtWorks y considerado uno de los gurús del desarrollo de software ágil y orientado a objetos, allá por el año 2000.

Martin Fowler expuso:

> «La integración continua es una práctica de desarrollo de software en la cual los miembros de un equipo integran su trabajo frecuentemente, como mínimo de forma diaria. Cada integración se verifica mediante una herramienta de construcción automática para detectar los errores de integración tan pronto como sea posible. Muchos equipos creen que este enfoque lleva a una reducción significativa de los problemas de integración y permite a un equipo desarrollar software cohesivo de forma más rápida».
>
> Martin Fowler (2006). «Continuous Integration».

La integración continua afecta especialmente al proceso de desarrollo del producto, añadiendo nuevas prácticas y automatizando las que ya existían.

Durante el desarrollo de un nuevo proyecto, los desarrolladores integran nuevos cambios dentro del código original. Sabemos que todas estas nuevas aportaciones tienen que ser puestas en los entornos de producción frecuentemente para asegurar un correcto desarrollo de las funcionalidades de nuestro proyecto, y es aquí donde la idea de la integración continua toma sentido. Todo proceso de integración continua comienza con la construcción de un proceso continuo de integraciones automáticas de un proyecto lo más a menudo posible; este proceso completo se conoce con el nombre de *pipeline*.

Las nuevas aportaciones serán dispuestas dentro del control de versiones (CVS, Git, Subversion, Mercurial o Microsoft Visual SourceSafe) donde se guardarán en el respectivo repositorio. El propósito de la integración continua será el de

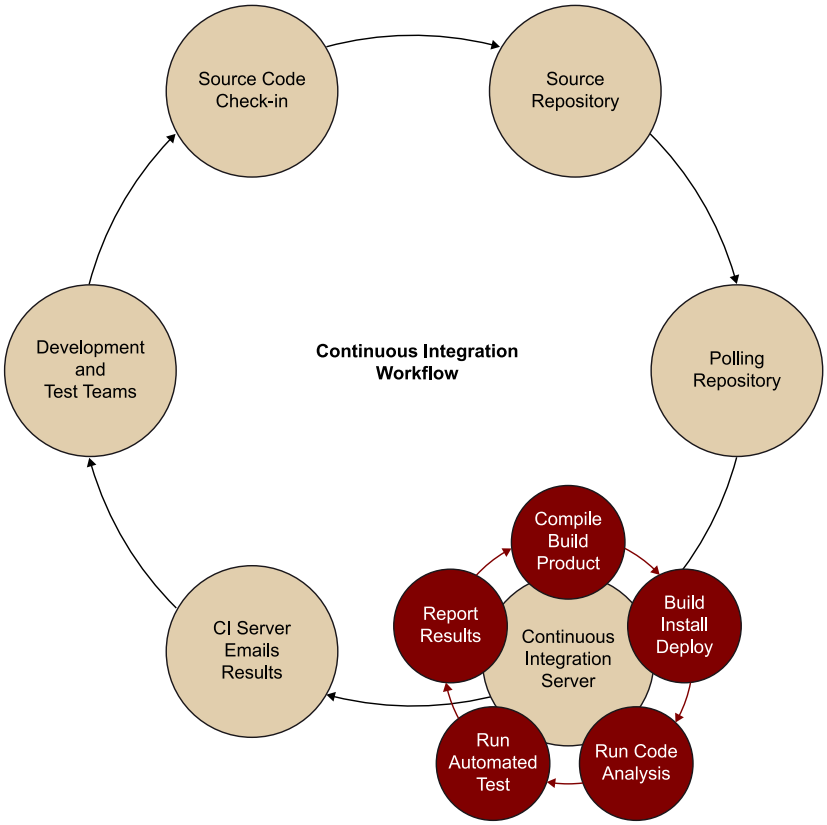
procesar esta nueva versión, compilando el código, desplegando y testando de manera automática, y devolviendo un informe a los desarrolladores con los posibles errores que se hayan producido dentro del proyecto durante la fase de test de este.

En este momento se produce una retroalimentación del sistema donde los desarrolladores ejecutan un bucle de cambio de versión-compilación-test-corrección, ya que los errores durante el proceso de integración son descubiertos con rapidez, generalmente en cuestión de minutos u horas, según la planificación de la frecuencia establecida del proceso de integración.

Una vez que los cambios superan todas las pruebas, estos como tales pueden ser integrados en el control de versiones de manera definitiva. La forma más sencilla de ver reflejado este proceso se encontraría en la ejecución de estos test donde el repositorio utilizado es el local y hasta no haber superado todas las fases de comprobaciones estos no serán subidos al repositorio remoto. Estos procesos pueden abstraerse a los repositorios remotos; ya existen métodos para ejecutar el código durante el llamado *pull-request* antes de asimilarlos dentro de la rama principal del proyecto.

En este punto, el código se ha visto testado e integrado al código final del proyecto, podemos decir que está listo para la puesta en producción.

Figura 3. Flujo de trabajo en integración continua



Fuente: <http://www.retrieverconsulting.com/Continuous%20Integration%20Workflow.jpg>.

2.3.1. Buenas prácticas de integración continua

1) Mantener un único repositorio de código fuente

El software de control de versiones es una herramienta indispensable para almacenar nuestro código fuente, y nos aportará información para hacer el seguimiento de todos los cambios que el código haya sufrido desde el inicio del proyecto, reflejando qué ficheros han sido modificados y por quién.

Dentro del repositorio habrá que incluir todos aquellos ficheros necesarios para poder construir nuestro proyecto. Hay que dar la posibilidad de que, una vez descargada la última versión, cualquier usuario pueda compilar, desplegar y ejecutar el proyecto de forma íntegra.

2) Automatizar la construcción del proyecto

Construir un proyecto puede resultar una tarea laboriosa, ya que requiere en muchas ocasiones la compilación del código fuente, y en el caso de sistemas avanzados, la creación de una base de datos, carga de esquemas, población de la base de datos, carga de configuraciones, etc.

La automatización de construcción mediante *scripts* o herramientas como Maven y Ant será indispensable en todos los proyectos, y habrá que centrar todos los esfuerzos en la fase de desarrollo.

3) Elaboración y ejecución de los test dentro de la construcción del proyecto

Se deberá incluir dentro del proceso de desarrollo la elaboración del conjunto de pruebas que certificarán que nuestro producto se comporta de una manera adecuada. Cualquier error será detectado en esta fase y se dará la oportunidad a los desarrolladores de modificar el código.

Existen numerosas herramientas para la construcción de test que simplifican la creación de estos; algunas de las más conocidas serían JUnit para Java, SimpleTest y PHPUnit para PHP, CPPUnit para C/C++, NUnit para la plataforma .NET, PyUnit para python, etc.

4) Integrar como mínimo una vez al día los cambios en la línea principal

Mantener la línea principal de código al día será fundamental para mantener a los desarrolladores al día de los cambios que este vaya sufriendo. Si esta práctica se ejecuta cada poco tiempo, los desarrolladores deberán primero actualizar su repositorio local, resolviendo cualquier conflicto existente antes de subir los cambios que tengan preparados.

Manteniendo esta filosofía de actualizaciones de repositorio local y remoto, se contribuirá a la resolución de conflictos cuando estos todavía son fáciles de arreglar. Así pues, la integración de los cambios diariamente será básica para la elaboración de un software donde los errores derivados de las actualizaciones se detectan casi de manera inmediata.

5) Construir la línea principal en la máquina de integración

A pesar de las continuas integraciones de los desarrolladores, todavía pueden surgir problemas causados por la diferencia de plataformas de desarrollo entre los miembros del equipo de desarrollo.

Será importante construir el proyecto dentro de la máquina de integración para asegurar que nuestro código es funcional y que no contiene errores de ejecución en la máquina de integración.

Las construcciones podrán realizarse de forma manual (en este caso el desarrollador construirá el proyecto y quedará pendiente del resultado de este) o mediante la utilización de un servidor de integración continua en el que se construirán todas las aportaciones integradas a la línea principal de proyecto y que reportará automáticamente a los desarrolladores cualquier error que durante el proceso de construcción haya detectado.

6) Mantener una ejecución rápida de la construcción del proyecto

Para asegurar un proceso de construcción y validación ágiles, se recomienda dentro de lo posible mantener procesos de construcción rápidos, que no deben superar los diez minutos.

En caso de que nuestro proceso de construcción y test supere el tiempo recomendado, deberemos agilizar el tiempo de ejecución de la construcción del proyecto. Para ello, limitaremos las pruebas dejando las más exhaustivas y complicadas por ejemplo para ejecuciones elaboradas durante la noche. Así, se agilizará el proceso de integración continua y cada día se poseerá un informe completo del estado de todas las pruebas sobre la línea principal.

7) Mantener las pruebas de integración en máquinas réplicas del entorno de producción

Será importante que el lugar en el que se realicen las pruebas de construcción y la ejecución del proyecto sea una réplica exacta del modelo de producción, ya que cada posible diferencia entre entornos puede suponer un riesgo para la correcta ejecución del código desarrollado.

Idealmente, las máquinas donde se realizará la integración continua deberán ser una réplica, o por lo menos, en el caso de que no sea posible, lo más parecidas al entorno de producción en cuanto a sistema operativo, configuraciones, versión de base de datos, etc., solo así podremos asegurar que el código, una vez puesto en producción, no sufra errores no detectados con anterioridad.

8) Almacenar los ejecutables de las versiones del proyecto

Durante cada fase de prueba y construcción exitosa dispondremos de binarios de los cuales podemos asegurar que han pasado los conjuntos de pruebas unitarias y que se han probado en un entorno similar al de producción sin ningún error; así pues, en cualquier etapa del *sprint* o iteración estaremos en disposición de mostrar la última versión plenamente funcional, sin ningún esfuerzo añadido.

Será importante mantener almacenados estos binarios para que todo el equipo involucrado en el desarrollo del software pueda ejecutar la última versión, facilitando las demostraciones y revisiones de los últimos cambios en la línea principal del proyecto.

9) Percepción rápida del estado y cambios del sistema

Todos los miembros del equipo deben saber el estado en el que se encuentra la línea principal del proyecto, Las herramientas web de los actuales servidores de integración continua también tienen como misión la de informar en qué estado se encuentra el proyecto, si se han aplicado cambios y si se han encontrado errores durante el proceso.

La comunicación del estado del proyecto es una de las partes más importantes, y para ello existen soluciones elaboradas y creativas pero igual de efectivas usando luces, lámparas de lava verdes y rojas totalmente integradas dentro del sistema, semáforos de estado del servidor y una multitud de ideas para hacer de este un proceso visible para todo el equipo.

Figura 4. Semáforo integrado con Jenkins basado en Raspberry Pi

Fuente: <https://en.wikipedia.org/wiki/Build_light_indicator>.

10) Automatizar el despliegue

Para implementar una integración continua se necesita disponer de distintos entornos: entornos de desarrollo, entornos de integración y entornos de producción; estos ejecutarán las tareas de construcción y test varias veces al día, por lo que se tiene que automatizar al máximo la fase de despliegue de aplicaciones.

2.3.2. Beneficios de la integración continua

La configuración de una integración continua y el entorno de despliegue continuo añaden una cierta sobrecarga para el proyecto, pero los beneficios son muy superiores.

Se incrementa la calidad del producto mediante la colaboración entre los desarrolladores, lo que hace más simple la resolución de problemas de integración de forma continua, y evita problemas durante las sucesivas fases de entrega del producto.

Se incrementa la calidad del proceso: la automatización y la monitorización de las pruebas continuas por parte del servidor de integración permiten crear un proceso limpio de retroalimentación con los desarrolladores, que definirán una filosofía de desarrollo óptima, que reducirá al mínimo cualquier error en el producto final.

Se incrementa la calidad de las personas: la implementación de la integración continua permite a los equipos trabajar en el desarrollo y la mejora continua de pruebas, así como ejercer buenas prácticas de programación que a la vez repercutirán en un código de mayor calidad. El proceso da seguridad al equipo, ya que en todo momento sabrá que el proyecto está funcionando.

Vistas las ventajas de la integración continua, cabe señalar que existen distintas herramientas que nos permitirán efectuar esta tarea. En este módulo comentaremos una de las más usadas: Jenkins, que permite la elaboración completa del *pipeline*, desde la incorporación de código hasta el despliegue en los distintos entornos.

A continuación se citan algunas de las herramientas más populares para realizar las tareas de integración continua:

- Jenkins.
- Travis Cl.
- Gitlab CI.
- Bamboo.
- Circle CI.
- ThoughtWorks.
- BuildBot.
- BuddyBuild Codeship.
- Cruise control.
- Go CD.
- Integrity.
- Strider CD.
- Urbancode deploy.
- AWS CodePipeline.
- Codefresh.

Como podemos ver, tendríamos una clase test donde definimos que queremos instalar siempre la última versión del paquete ejemplo. Esta clase la aplicaremos a tantos nodos como queramos. En la figura anterior se ha aplicado al nodo servidor-test.

2.3.3. Jenkins en detalle

Jenkins es un software de integración continua *open source* escrito en Java, está basado en el proyecto Hudson y proporciona integración continua para el desarrollo de software. Es un sistema corriendo en un servidor que es un con-

tenedor de servlets, como Apache Tomcat. Soporta herramientas de control de versiones, como CVS, Subversion, Git o Mercurial, y puede ejecutar proyectos basados en Apache Ant y Apache Maven, así como *scripts* de *shell*, etc.

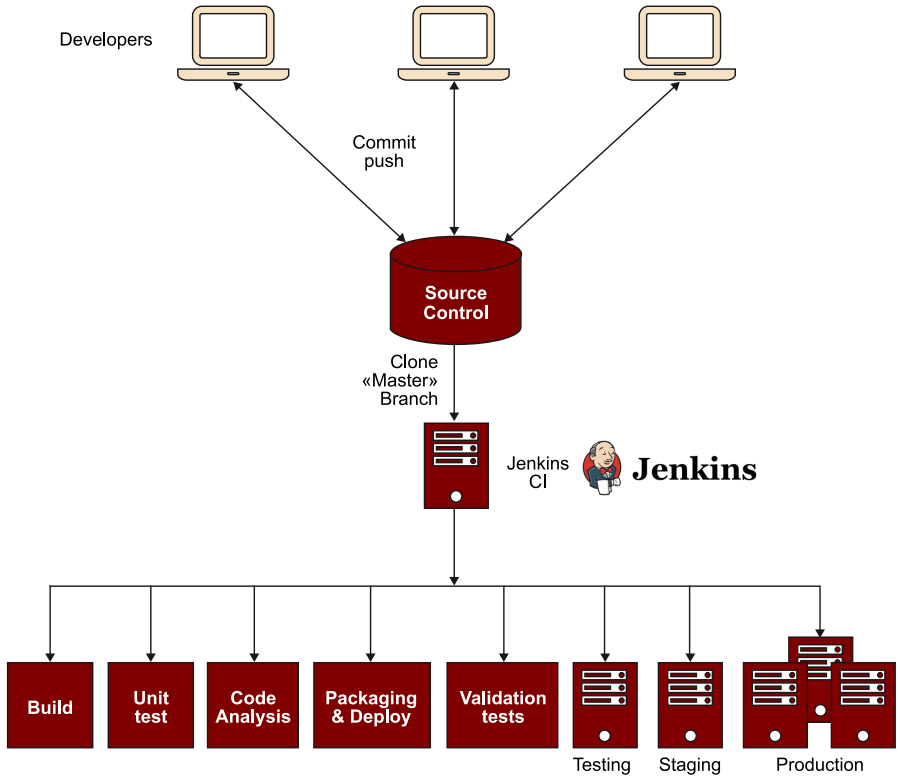
Una de las características de Jenkins es que es tremendamente flexible, con más de 350 *plugins* disponibles. Por ejemplo, es posible tener varios proyectos configurados en el mismo servidor que utilicen diferentes versiones de JAVA en el momento de la compilación.

Por otro lado, Jenkins es fácilmente clusterizable, lo que implica que se pueden tener varios esclavos configurados en el mismo servidor maestro. La ventaja más obvia es que se puede balancear la carga de los diferentes proyectos en varios servidores; otra ventaja es que los esclavos pueden basarse en diferentes arquitecturas, con lo cual se pueden compilar aplicaciones basadas en Linux y otras en Windows bajo el mismo servidor maestro.

Otra característica de Jenkins son sus integraciones con Git, JMeter, SOAP-UI, JUnit, Sonar, Nexus, etc. Esto permite ejecutar, por ejemplo, una batería de test antes de proceder a la compilación del código con el objetivo de asegurar un cierto grado de calidad, llegando incluso a abortar la compilación si los resultados de los test no son satisfactorios.

Por último, Jenkins está diseñado para estar en constante comunicación con los desarrolladores, y para ello consta de un potente sistema de notificaciones para seguir el estado de los diferentes proyectos. Jenkins es capaz de enviar correos, tuitear, activar señales luminosas o incluso sonoras, y mucho más gracias a su amplia API.

Figura 5. Flujo de datos en Jenkins



Fuente: <https://code2read.files.wordpress.com/2015/11/jenkins.png>.

2.4. Estrategias de *testing*

Hemos visto que la integración continua va unida a una filosofía de agilización de los métodos adoptados por todos los miembros involucrados en el proyecto, pero ¿qué hay de la calidad del software? Es obvio que la calidad del producto desarrollado tendrá que cumplir con los requerimientos funcionales acordados en todas las fases de entrega.

Enlace de interés

Para saber más sobre estrategias de *testing*, podéis consultar el siguiente enlace: <http://www.carlescliment.com/publications/calidad-e-integracion-continua-enero-2012>.

En muchas ocasiones habremos escuchado «la fase de pruebas del proyecto se ejecutará al final» o «no hay tiempo para poder aplicarlos de forma correcta», dando a entender que el proceso de QA demora el tiempo de desarrollo; este modo de pensar no existe dentro de la metodología de integración continua.

En un entorno tradicional, los equipos de Testing solían ser un área independiente dentro de la empresa y formaban parte del ciclo de vida del producto. Estos, mediante el uso de *checklist*, ejecutaban una y otra vez las mismas pruebas en cada nueva versión para asegurar que no se produjeran errores causados por el cambio de versión. Como hemos visto, la metodología DevOps tratará de asimilar esta forma de trabajar dentro de su fase de desarrollo.

Para asegurar en todo momento la calidad (valor) de nuestro proyecto, se definirán métodos ágiles de *testing* (*agile testing*) para acelerar las pruebas, testando y analizando el código desarrollado. Cualquier versión no apta para pasar a producción será desechada tan pronto como sea posible, con lo que nos ahorraremos pérdidas de tiempo que una mala integración hubiera supuesto.

La elaboración de los test cumplirá con la definición del acrónimo en inglés FIRST:

1) ***Fast*** (rápido). Cuanto más rápidos se ejecuten los test, mayor será la frecuencia con la que se ejecutarán. Los desarrolladores –mediante la ejecución de subconjuntos de pruebas– pueden agilizar la fase de pruebas. Cabe remarcar que antes de que el proyecto pase a producción se tendrán que haber pasado el total de las pruebas; de lo contrario, no se podrá verificar que el software desarrollado esté perfectamente acoplado, ya que se habrán testado sus partes pero no en su totalidad.

Algunas de las herramientas destinadas al *continual testing* (CT), como Infinitest, calculan qué test son más apropiados mediante el análisis de las dependencias de código.

2) ***I****solated* (aislado). Cada test debe tener una sola razón para fallar. El diseño de cada prueba tiene que ser independiente de los factores externos e independiente de los resultados de test anteriores. La ordenación de pruebas unitarias para mejorar el tiempo de ejecución es una señal de falta de aislamiento.

3) ***R****epeatable* (repetible). Los test deben devolver información sobre si han tenido éxito o no, y deben obtener los mismos resultados cada vez que se ejecutan. Ocasionalmente, las pruebas pueden fallar intermitentemente, debido a razones de sobre-especificación en la fase de test, lentitud en la carga de dependencias, uso de *threads* y procesos no deterministas, volatilidad de datos, etc. En cualquier caso, los fallos ocasionales o intermitentes serán difíciles de definir.

Para asegurar el correcto funcionamiento, se añadirán precondiciones que, en el caso de no cumplirse, detendrán las pruebas antes de que estas puedan empezar a ejecutarse.

4) ***S****elf-validating* (autoverificado, sin ambigüedad). Para que un buen test verifique que se ha superado de forma correcta, debe dar un resultado inequívoco, descartando cualquier ambigüedad. Cuando todos los test son superados con éxito, podemos tener una total confianza en el código desarrollado y pasar al siguiente nivel. Si uno de los test fallase, el código sería puesto en disposición del equipo de desarrollo para corregirlo.

Si alguno de los resultados de los test dejara aspectos ambiguos abiertos a la interpretación humana, la validación de estos se convertiría en un lastre para la productividad de los desarrolladores. Por tanto, se descartará cualquier tipo de pruebas que puedan resultar ambiguas, ya que dejarán de ser útiles.

5) ***Timely*** (oportuno). ¿Cuando se deben escribir las pruebas?, ¿antes de escribir el código o una vez está preparado para su construcción (*build*)?

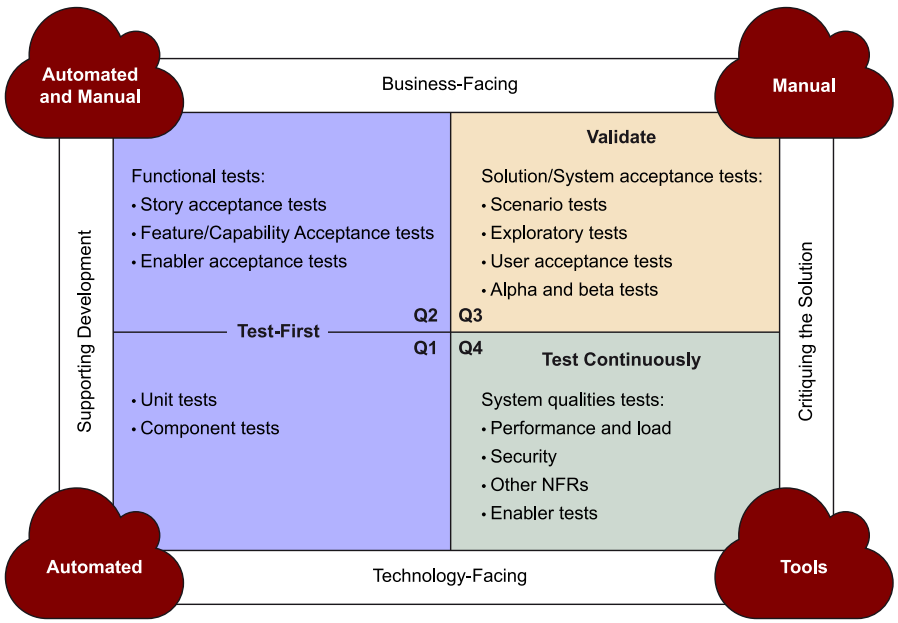
Las pruebas deben desarrollarse en el momento oportuno. El desarrollo de los test deberá realizarse antes del desarrollo del producto. Estos definirán el comportamiento del código que se pretende construir; así, ayudarán a entender en todo momento lo que se pretende de las partes del código desarrollado.

Las pruebas escritas *a posteriori* requieren un esfuerzo adicional para los desarrolladores, que deberán refactorizar el código hasta tener una batería de test que cumplan con los principios mencionados (FIRST). Este modo de trabajar da una sensación de invertir demasiado tiempo en el «pulido» de código y puede desembocar en un abandono del desarrollo de pruebas.

2.4.1. Los cuadrantes del Agile Testing

Brian Marick en el 2003 expuso por primera vez en su blog la idea de clasificar los diferentes tipos de pruebas ejecutadas durante el desarrollo del producto en cuatro cuadrantes. Posteriormente, esta idea fue mejorada hasta resultar en la imagen actual de los cuadrantes del Agile Testing. La misión fundamental de la clasificación de las pruebas es ayudar a contextualizar y guiar a los equipos durante la integración de las pruebas para garantizar la calidad del producto desarrollado.

Figura 6. Cuadrantes del Agile Testing



Así pues, se presenta una matriz 2 x 2 donde los ejes definirán las pruebas desde los diferentes puntos de vista posibles:

- El eje vertical definirá la orientación de las pruebas, pudiendo ser orientadas al negocio (*business-facing*), comprensibles por el usuario, o tomadas desde el punto de vista tecnológico (*technology-facing*), descritas en lenguaje de los desarrolladores para evaluar si el código se comporta de una manera correcta.

- El eje horizontal definirá el tipo de pruebas, basadas en el soporte de los desarrolladores (*supporting development*) mediante la evaluación del código interno o por los que critican el producto (*critiquing the solution*) a través de la evaluación del sistema en contra de las necesidades del usuario.

La enumeración de los cuadrantes Q1, Q2, Q3 y Q4 no determina la importancia ni el orden del uso de las pruebas. Estos nombres no pretenden establecer ningún orden determinado de ejecución y están dispuestos de manera arbitraria.

- **Cuadrante Q1**: Contiene las pruebas unitarias y componentes. Estas pruebas, escritas por los desarrolladores, se ejecutan antes y después de cada cambio en el código y sirven para confirmar que el sistema funciona correctamente. La automatización reducirá el tiempo dedicado en la fase de test por parte de los desarrolladores y permitirá asegurar la calidad del producto.

- **Cuadrante Q2**: Contiene las pruebas funcionales (pruebas de aceptación de usuario). Estas pruebas no unitarias basadas en la abstracción de historias de usuarios permiten validar el sistema desde un punto de vista más técnico de lo que podría realizar un usuario final y son fácilmente automatizables.

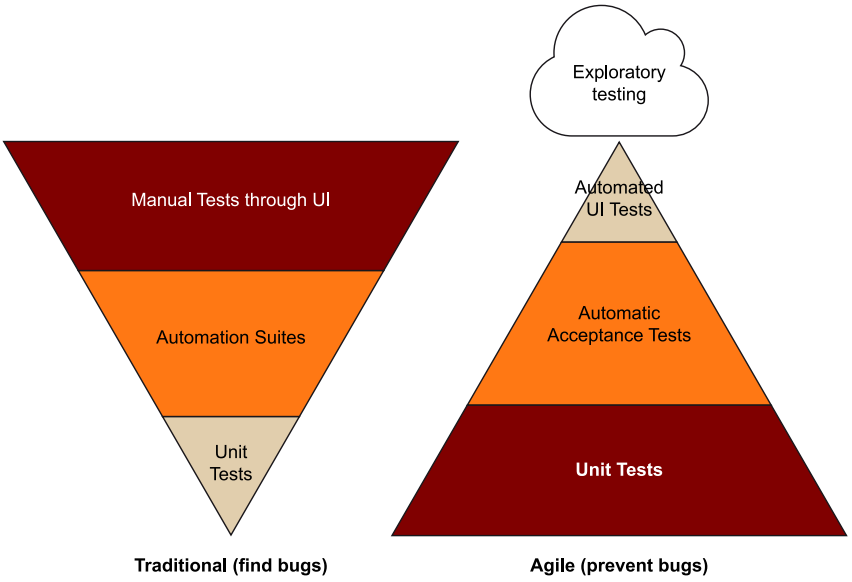
- **Cuadrante Q3**: Contiene las pruebas de aceptación a nivel de sistema. Estas serán ejecutadas de forma manual debido a la implicación de las pruebas de aceptación del usuario, pruebas exploratorias y pruebas basadas en los diferentes escenarios definidos validando si el comportamiento del sistema cumple con los requisitos funcionales.

- **Cuadrante Q4**: Contiene las pruebas de calidad del sistema y verifica si este cumple con los requisitos no funcionales (NFR). Estas pruebas son fácilmente automatizables y existen un amplio abanico de herramientas que permiten realizar pruebas de rendimiento, pruebas de usabilidad, pruebas de seguridad, pruebas de estabilidad, etc.

2.4.2. La pirámide del *testing*

La pirámide del Agile Testing es una manera práctica de describir la diferencia entre las pruebas de software tradicional y las pruebas durante el desarrollo iterativo. Este concepto fue descrito por primera por Mike Cohn.

Figura 7. Pruebas tradicionales frente a Agile Testing



Fuente: <http://www.pmoinformatica.com/2015/03/que-es-el-agile-testing.html>.

Las pruebas tradicionales pueden ser descritas por la pirámide de la izquierda. La mayoría de los test se llevan a cabo utilizando planes de prueba de forma manual que implican el uso de la interfaz gráfica. En la parte central, algunos de los test serán lanzados automáticamente para poner a prueba los servicios del producto, y finalmente en última posición encontraríamos algunos test unitarios.

Esta forma de pensar fue diseñada para encontrar errores, dentro de una mentalidad tradicional por etapas en las que se codifica y se corrige. Los desarrolladores parten de la idea de que su código es perfecto, y a continuación el equipo de test prueba a conciencia el producto para reportar cualquier problema en caso de ser detectado, lo que alarga considerablemente la fase de codificación y prueba.

Desde el punto de vista del Agile Testing, la pirámide aparece de forma invertida (lado derecho de la figura 7). Esta nueva manera de enfocar las pruebas presenta una base sólida, donde la automatización adquirirá un peso importante y con ello una mayor contribución por parte de los desarrolladores para evitar errores. La parte superior de la pirámide es muy pequeña, la interfaz del usuario es un elemento cambiante, cualquier modificación significa un cam-

bio en los test, con el consecuente gasto de tiempo en el desarrollo. El mantenimiento de la capa de test puede superar fácilmente al valor de las pruebas en la capa de la interfaz y deberá mantenerse como una capa con test básicos.

La automatización de los procesos de test implica una visión clara del sistema y cómo este debe ejecutarse, asegurando que cada vez que se pasan los test, el código desarrollado cumpla las especificaciones funcionales de forma correcta. La esencia del nivel medio de la pirámide recaerá en la capa lógica de nuestro software (reglas de negocio, servicios...), que encontraremos inmediatamente por debajo de la interfaz gráfica; así, en esta capa las pruebas automatizadas pueden ser ejecutadas asegurando que estas funcionan correctamente sin tener que utilizar la interfaz de usuario con herramientas tipo Selenium.

La pirámide Agile requiere un cambio de mentalidad en las organizaciones: los desarrolladores validarán su propio trabajo mediante los juegos de test unitarios e incluirán una etapa de decisión sobre las herramientas que utilizar para efectuarlos. Dependiendo del lenguaje usado en la codificación del software, podremos elegir por ejemplo implementaciones típicas dentro de la familia xUnit.

En esta fase aparecen diferentes enfoques sobre cómo realizar las pruebas, basándose en distintas prácticas relacionadas con el Agile Testing.

2.4.3. Test Driven Development (TDD)

Mediante el desarrollo dirigido por pruebas orientamos el enfoque primero a la elaboración de los test para después desarrollar el código. Tras el primer cambio de mentalidad este método es muy productivo, ya que obliga a enfrentarse al problema desde el punto de vista del resultado, y una vez finalizado el desarrollo, ya se dispone del conjunto de pruebas que ejecutar.

Los *bugs* detectados servirán para ampliar el conjunto de pruebas y hacerlas a su vez más robustas.

La granularidad de la codificación permitió escribir las conocidas tres leyes del TDD:

- No se deberá escribir ningún código antes de haber escrito un test unitario que falle.

- No se deberá escribir más de una prueba que falle en ejecución o en compilación.

- No se deberá escribir más código que el suficiente para hacer pasar el test actual.

Ejecutando estas tres leyes se entrará en un ciclo rápido, en el que será necesario iterar varias docenas de veces antes de finalizar una unidad de test completa.

Red - Green - Refactor

Este método se interpreta como un microciclo dentro del desarrollo dirigido por pruebas, y es ejecutado en cada comprobación total de los test o cada *n* ciclos de estos. El proceso empezará con la definición de una prueba que falle, fase que es conocida como Red por el color en el que se suelen mostrar los resultados fallidos en las herramientas de *testing*.

Cita

«Make it work. Make it right. Make it fast». Ken Beckt

«Getting software to work is only half of the job». Robert C. Martin

Dado un test fallido, el segundo paso será implementar el código mínimo para hacer que este lo pase; en este momento nos encontraremos en fase Green.

El último paso dentro del ciclo Red - Green - Refactor se centra en la limpieza de código, eliminando duplicidades y refactorizando el código.

Esta filosofía está basada en la idea de que nuestra mente está limitada y no es capaz de llevar a cabo dos objetivos simultáneamente: hacer que el software funcione correctamente y que el software posea una estructura correcta.

2.4.4. Acceptance Test Driven Development (ATDD)

Se describe el ATDD como la práctica en la que todo el equipo, clientes, desarrolladores y *testers* analizan y discuten conjuntamente los criterios de aceptación. Esta tarea dará como resultado un conjunto de pruebas aceptables de las que los desarrolladores dispondrán antes de empezar con su tarea. Poniendo las ideas en común se asegura que todos los miembros comparten una misma visión del proyecto y de lo que se espera obtener de él.

2.4.5. Behaviour Driven Development (BDD)

También conocido como Story Driven Development, es una evolución en la forma de pensar que encontramos detrás del TDD y el ATDD. Este enfoque se inicia con el análisis de una prueba funcional o una historia de usuario que es desarrollada mediante las fases del TDD hasta que se superan todas las pruebas. BDD establece que:

- Solo se desarrollarán aquellos comportamientos que contribuyan directamente a satisfacer las pruebas que superar con el fin de minimizar código residuo o no utilizado.

- Se describirán los comportamientos en una notación común para los analistas, los desarrolladores y los *testers* con el fin de mejorar la comunicación.

2.4.6. *Testing* exploratorio

El *testing* exploratorio se define como un enfoque simultáneo del aprendizaje de la funcionalidad, el diseño de pruebas y la ejecución de pruebas. Este es un método que valora a los *testers* como una parte integral del proceso, y ayuda a los desarrolladores a mantenerse al día con el desarrollo de los proyectos de software ágiles, cambiando la mentalidad de la visión estructurada de la fase de la construcción del proyecto al hacerla más global.

2.4.7. Automatización de pruebas de regresión

La ejecución de las fases del TDD define una continua modificación del código, donde en cualquier momento del desarrollo pueden surgir carencias de funcionalidades o comportamientos no esperados del software. Las pruebas de regresión intentan descubrir los errores producidos por cambios dentro del proyecto. Estos errores normalmente son producidos por errores en la contextualización, que exponen fragilidades del rediseño de nuestra aplicación.

Tanto la integración continua como la refactorización deberán incluir pruebas de regresión automatizadas para asegurar un correcto funcionamiento de la aplicación en todo momento.

2.4.8. Automatización de pruebas unitarias

Mediante la utilización de herramientas de la familia XUnit, los desarrolladores dispondrán de un *framework* completo donde poder elaborar pruebas unitarias y en el que asegurar que cada componente de la aplicación produce una salida determinada para una entrada dada.

Las pruebas serán ejecutadas en cada modificación del código automáticamente y notificadas al equipo para asegurar que todas las funcionalidades devuelven un valor esperado.

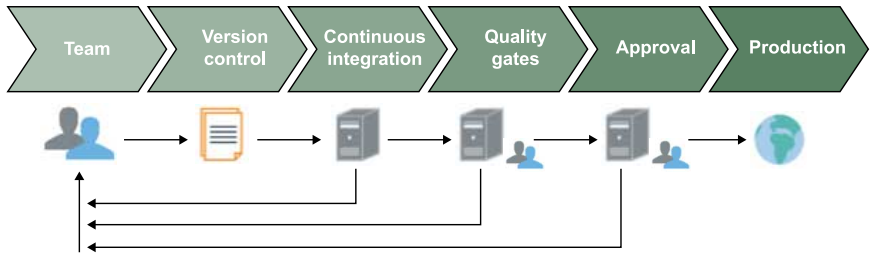
2.5. Entrega continua

Hoy en día, la entrega continua se entiende como la evolución lógica de la integración continua, de tal modo que para que los cambios realizados en Desarrollo puedan ser susceptibles de ser entregados en Producción en el menor tiempo posible, minimizando los riesgos de implantación.

Gracias a este proceso, podemos evitar largas puestas en producción, las sorpresas de última hora y los cambios muy grandes y costosos, evitando que haya demasiada incertidumbre a la hora de la entrega y teniendo un mayor control sobre todo el proceso.

Esta metodología es usada por grandes compañías, como **Flickr**, con varios despliegues diarios; **Amazon**, liberando en producción una versión cada 11,6 segundos de media; y **Facebook**, desplegando la compilación de código en producción al menos una vez al día con cambios menores o una vez a la semana con cambios mayores.

Figura 8. Flujo de datos en la entrega continua



Fuente: <https://s3.amazonaws.com/media-p.slid.es/uploads/stevenmaguire/images/723464/chart-continuous-delivery.png>.

Una vez vistos los conceptos de control de versiones, la integración continua y las diferentes estrategias de test, ahora pondremos el foco en las diferentes estrategias de despliegue en producción.

En cuanto a herramientas, se pueden utilizar tanto las propias herramientas de integración continua como el propio servidor Jenkins; y también se pueden utilizar las propias herramientas de gestión de configuración como Puppet o Chef.

2.5.1. Desplegando vía Jenkins

En este caso, Jenkins nos ofrece la flexibilidad tanto de acceder a los servidores de producción para hacer la transferencia de la nueva versión del software, por ejemplo en el caso de desplegar código php en una aplicación basada en web, como de utilizar uno de los muchos *plugins* que ofrece Jenkins.

El siguiente ejemplo corresponde a la configuración XML de un proyecto Maven que compila el código JAVA de un repositorio Git y lo despliega en el entorno de producción:

```
<?xml version='1.0' encoding='UTF-8'?>
<maven2-moduleset plugin="maven-plugin@2.13">
  <actions/>
  <description>Deployment of deploy-all-packages from project/code.git to PRODUCTION servers.
  </description>
  <keepDependencies>false</keepDependencies>
  <properties>
    <jenkins.model.BuildDiscarderProperty>
      <strategy class="hudson.tasks.LogRotator">
        <daysToKeep>-1</daysToKeep>
        <numToKeep>20</numToKeep>
```

```
        <artifactDaysToKeep>-1</artifactDaysToKeep>
        <artifactNumToKeep>1</artifactNumToKeep>
      </strategy>
    </jenkins.model.BuildDiscarderProperty>
    <org.jenkinsci.plugins.mavenrepocleaner.MavenRepoCleanerProperty
    plugin="maven-repo-cleaner@1.2">
      <notOnThisProject>false</notOnThisProject>
    </org.jenkinsci.plugins.mavenrepocleaner.MavenRepoCleanerProperty>
    <com.sonyericsson.rebuild.RebuildSettings plugin="rebuild@1.25">
      <autoRebuild>false</autoRebuild>
      <rebuildDisabled>false</rebuildDisabled>
    </com.sonyericsson.rebuild.RebuildSettings>
    <hudson.model.ParametersDefinitionProperty>
      <parameterDefinitions>
        <hudson.model.ChoiceParameterDefinition>
          <name>server</name>
          <description>To which server to deploy</description>
          <choices class="java.util.Arrays$ArrayList">
            <a class="string-array">
              <string>http://servidor-produccion-1.dominio.com:8080</string>
              <string>http://servidor-produccion-2.dominio.com:8080</string>
              <string>http://servidor-produccion-3.dominio.com:8080</string>
            </a>
          </choices>
        </hudson.model.ChoiceParameterDefinition>
        <hudson.model.StringParameterDefinition>
          <name>password</name>
          <description>Admin password for the selected server</description>
          <defaultValue>admin</defaultValue>
        </hudson.model.StringParameterDefinition>
        <hudson.model.StringParameterDefinition>
          <name>deploy-version</name>
          <description>Which version of the complete-packages to deploy (leave empty
          to use latest available version)</description>
          <defaultValue></defaultValue>
        </hudson.model.StringParameterDefinition>
      </parameterDefinitions>
    </hudson.model.ParametersDefinitionProperty>
  </properties>
  <scm class="hudson.plugins.git.GitSCM" plugin="git@2.5.2">
    <configVersion>2</configVersion>
    <userRemoteConfigs>
      <hudson.plugins.git.UserRemoteConfig>
        <url>ssh://git@servidor-git.com:7999/proyecto/codigo.git</url>
      </hudson.plugins.git.UserRemoteConfig>
    </userRemoteConfigs>
    <branches>
```

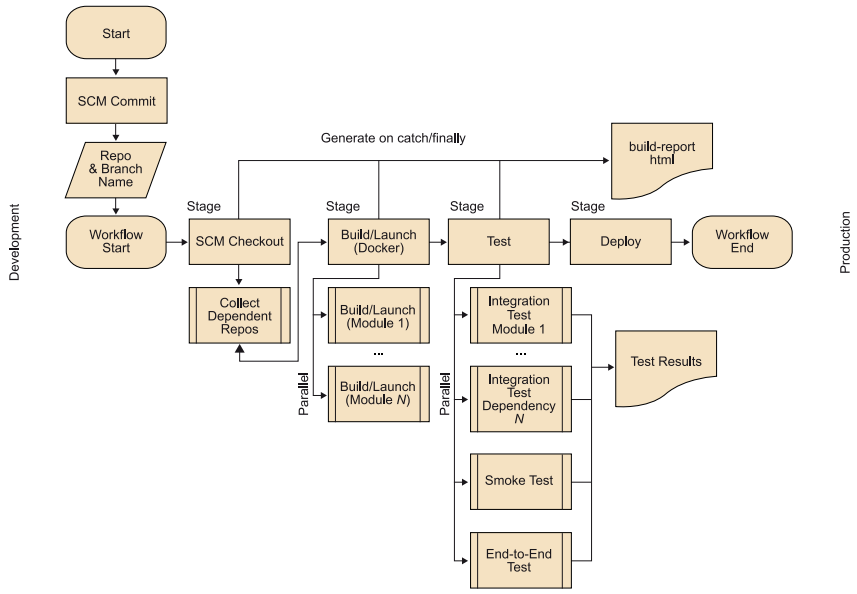
```
      <hudson.plugins.git.BranchSpec>
        <name>*/develop</name>
      </hudson.plugins.git.BranchSpec>
    </branches>
    <doGenerateSubmoduleConfigurations>false</doGenerateSubmoduleConfigurations>
    <gitTool>Default</gitTool>
    <submoduleCfg class="list"/>
    <extensions/>
  </scm>
  <canRoam>true</canRoam>
  <disabled>true</disabled>
  <blockBuildWhenDownstreamBuilding>false</blockBuildWhenDownstreamBuilding>
  <blockBuildWhenUpstreamBuilding>false</blockBuildWhenUpstreamBuilding>
  <jdk>JDK8</jdk>
  <triggers/>
  <concurrentBuild>false</concurrentBuild>
  <rootModule>
    <groupId>com.project.deploy</groupId>
    <artifactId>project-deploy-all-packages</artifactId>
  </rootModule>
  <rootPOM>deploy-all-packages/pom.xml</rootPOM>
  <goals>-e clean install -U -Dcq.server=$server -Dcq.password=$password
  -Dinstall-project-all</goals>
  <mavenName>Maven 3.2.x</mavenName>
  <mavenOpts>-Xmx1024m</mavenOpts>
  <aggregatorStyleBuild>true</aggregatorStyleBuild>
  <incrementalBuild>false</incrementalBuild>
  <ignoreUpstremChanges>true</ignoreUpstremChanges>
  <ignoreUnsuccessfulUpstreams>false</ignoreUnsuccessfulUpstreams>
  <archivingDisabled>false</archivingDisabled>
  <siteArchivingDisabled>false</siteArchivingDisabled>
  <fingerprintingDisabled>false</fingerprintingDisabled>
  <resolveDependencies>false</resolveDependencies>
  <processPlugins>false</processPlugins>
  <mavenValidationLevel>-1</mavenValidationLevel>
  <runHeadless>false</runHeadless>
  <disableTriggerDownstreamProjects>false</disableTriggerDownstreamProjects>
  <blockTriggerWhenBuilding>true</blockTriggerWhenBuilding>
  <settings class="jenkins.mvn.DefaultSettingsProvider"/>
  <globalSettings class="jenkins.mvn.DefaultGlobalSettingsProvider"/>
  <reporters>
    <hudson.maven.reporters.MavenMailer>
      <recipients>mail@domain.com</recipients>
      <dontNotifyEveryUnstableBuild>false</dontNotifyEveryUnstableBuild>
```

```
      <sendToIndividuals>true</sendToIndividuals>
      <perModuleEmail>false</perModuleEmail>
    </hudson.maven.reporters.MavenMailer>
  </reporters>
  <publishers/>
  <buildWrappers>
    <hudson.plugins.timestamper.TimestamperBuildWrapper plugin="timestamper@1.8.4"/>
  </buildWrappers>
  <prebuilders/>
  <postbuilders/>
  <runPostStepsIfResult>
    <name>SUCCESS</name>
    <ordinal>0</ordinal>
    <color>BLUE</color>
    <completeBuild>true</completeBuild>
  </runPostStepsIfResult>
</maven2-moduleset>
```

Hasta el momento, para conseguir una *pipeline* de integración continua usando Jenkins, teníamos que definir dependencias entre proyectos. De esta manera, se ejecutaban en orden y se lograba la *pipeline* deseada. La versión 2.0 de Jenkins soluciona las limitaciones anteriores porque permite definir las fases del flujo: **Checkout**, **Build**, **Test** y **Deploy** gracias al *plugin* Pipeline; de esta manera, se pueden ejecutar en paralelo acciones correspondientes a la misma fase.

Cabe destacar que dicha *pipeline* puede ser generada mediante código, lo que permite hacer cambios muy fácilmente y usar control de versiones sobre las *pipelines*.

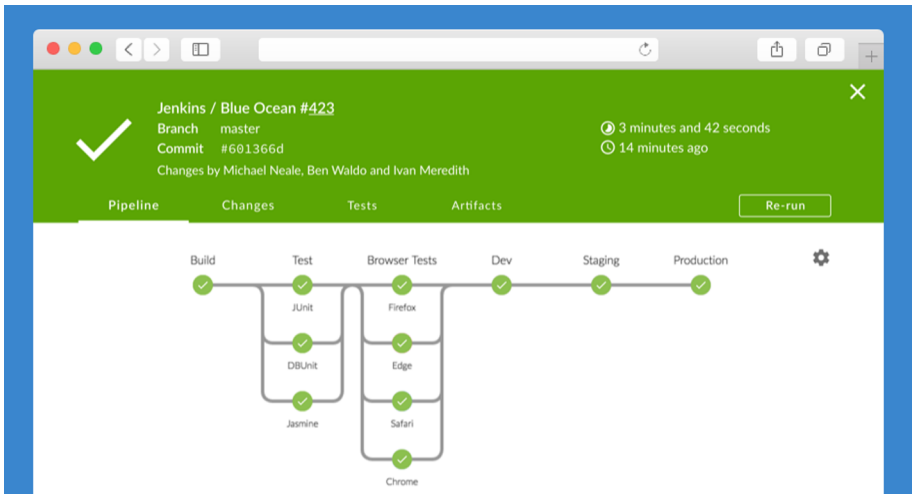
Figura 9. Flujo definido usando el *plugin* Pipeline



Fuente: <https://jenkins.io/doc/book/pipeline/overview/>.

Para finalizar el ejemplo de despliegue basado en Jenkins veamos la nueva interfaz Blue Ocean, que nos muestra de una manera visual las diferentes fases y sus partes.

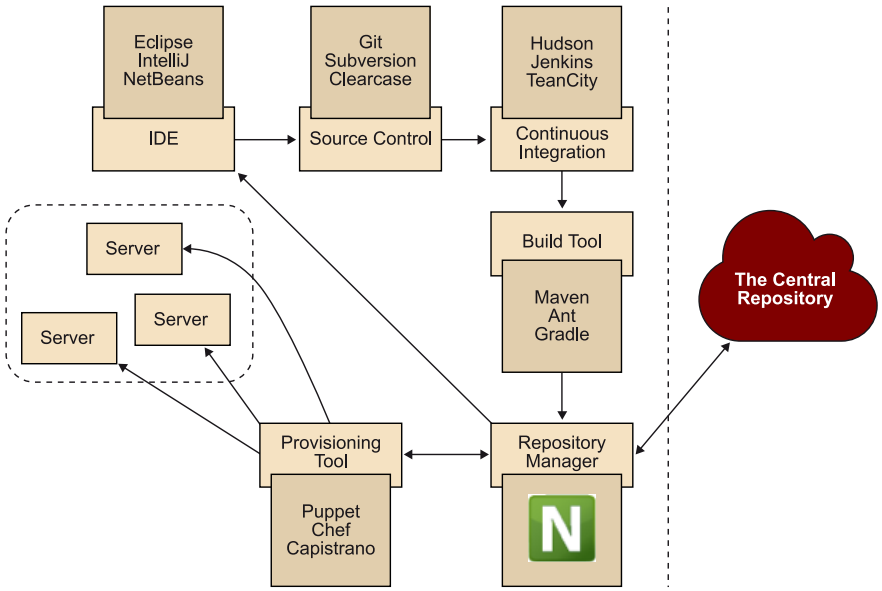
Figura 10. Visualización del flujo mediante la interfaz Blue Ocean



2.5.2. Desplegando vía gestor de configuración

Veamos ahora el uso de un gestor de configuración para realizar el despliegue en un entorno de *testing*. En este caso, igual que en el anterior, debemos realizar previamente el proceso de integración continua donde se genera una versión nueva; en este ejemplo generamos un fichero instalable tipo RPM Package Manager, en adelante rpm. Este fichero es transferido a un servidor de distribución tipo Nexus Repository OSS accesible por el gestor de configuración, que en este caso actúa de herramienta de aprovisionamiento.

Figura 11. Despliegue mediante Nexus y un gestor de configuración



En el caso de utilizar Puppet, el Puppet master será el encargado de distribuir el paquete. Los nodos del entorno de *testing* que así lo tengan configurado verán su versión del software automáticamente actualizada. El detalle de la configuración sería el siguiente:

Figura 12. Detalle de la configuración de un nodo de test

```
class test {
  package { 'ejemplo':
    ensure => 'latest',
  }
}

node servidor-test {
  class { 'test': }
}
```

Como podemos ver, tendríamos una clase test donde definimos que queremos instalar siempre la última versión del paquete ejemplo. Esta clase la aplicaremos a tantos nodos como queramos (en la anterior figura se ha aplicado al nodo servidor-test).

2.5.3. Estrategias para el despliegue

El despliegue en producción es el paso más crítico. Se puede hacer desplegando código nuevo directamente o implementando un cambio de configuración, y esto se puede realizar de varias maneras:

- Implementando una instalación paralela de una nueva versión del código y pasar a la nueva instalación con un cambio de configuración.

- Desplegando una nueva versión del código con el comportamiento antiguo y un indicador de característica, y cambiar al nuevo comportamiento con un cambio de configuración.

- Mediante el despliegue de servidores independientes (uno que ejecute el código antiguo, y uno nuevo) y redirigir el tráfico del antiguo al nuevo con un cambio de configuración en el nivel de enrutamiento de tráfico. Estos a su vez pueden hacerse de una vez o gradualmente.

La implementación de una nueva versión generalmente requiere un reinicio, a menos que el intercambio en caliente sea posible y, por lo tanto, requiere una interrupción en el servicio o implementar redundancia: reiniciando las instancias detrás de un balanceador de carga o iniciando nuevos servidores antes de tiempo y luego simplemente redirigir el tráfico a los nuevos servidores.

Al implementar una nueva versión en producción, en lugar de implementarla inmediatamente en todas las instancias o usuarios, puede desplegarse en una sola instancia o fracción de usuarios primero y, a continuación, desplegarse a todos o gradualmente desplegarse en fases para captar cualquier tipo de pro-

blema. Esto agrega complejidad debido a que múltiples versiones se ejecutan simultáneamente, pero reduce el impacto en caso de que aparezcan problemas en producción.

2.6. Gestión de infraestructura y entornos

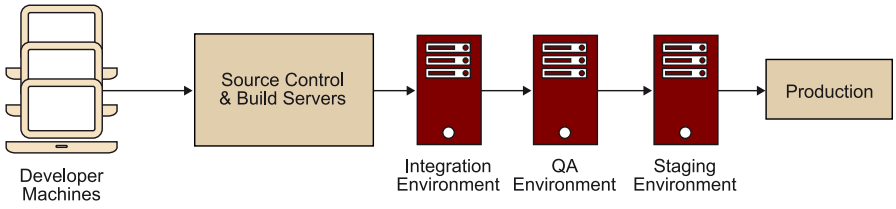
La **gestión de infraestructura** y de los diferentes **entornos** debe dar respuesta al diseño de las diferentes etapas de la entrega continua. Tanto la infraestructura como los diferentes entornos deben ser extremadamente flexibles para adaptarse rápidamente a los cambios que se puedan producir, ya que las etapas de desarrollo son tremendamente dinámicas.

2.6.1. Entornos involucrados en el desarrollo de software

Para dar cobertura al desarrollo basado en DevOps, debemos definir diferentes entornos donde se realizarán distintas tareas de validación que nos permitan progresar al siguiente entorno.

Empecemos viendo una configuración de entornos típica en escenarios DevOps:

Figura 13. Detalle de una configuración de entornos típica



Fuente: <https://www.infoq.com/articles/Continous-Delivery-Patterns>.

1) Desarrollo local

Normalmente empezamos en el ordenador del desarrollador, donde el nuevo código generado se compila localmente y se realiza una prueba inicial. Para esto se utilizan herramientas como Virtualbox o Vagrant, que permiten desplegar el nuevo código en un entorno parecido al entorno de producción. Finalmente, se traslada el nuevo código al entorno de integración.

2) Entorno de integración

En este entorno es donde se guardan los nuevos cambios y se compilan para integrarlos con el resto del código. Para ello, se utilizan herramientas como Git para la gestión del código fuente y Jenkins como servidor de integración. En este paso el propio servidor de integración normalmente incorpora como mínimo una batería de pruebas unitarias.

3) Entorno de test

En este entorno es donde se producen los test de regresión y funcionales. Como hemos visto, con estos test aseguramos que no se ha dañado ninguna funcionalidad disponible en el software y que las nuevas funcionalidades se ajustan a los requerimientos.

4) Entorno de preproducción

El entorno de preproducción o *staging* es una réplica del entorno de producción. En este entorno se dan por probadas las funcionalidades del software y el objetivo es realizar las pruebas de integración con otras piezas de software; probar la instalación, configuración y *scripts* de migración; y realizar test de rendimiento, en concreto pruebas de carga, ya que estas son muy dependientes del entorno.

5) Entorno de producción

Por último, tenemos el entorno de producción, donde los usuarios interactúan con el software desplegado.

2.6.2. Gestión de la infraestructura

Una vez vistos los diferentes entornos, veamos cómo se gestiona mediante código la infraestructura sobre la que corren dichos entornos.

Infrastructure as code es el proceso de gestión y aprovisionamiento de la infraestructura (procesos, servidores bare-metal, servidores virtuales, etc.) y su configuración a través de archivos de definición de infraestructura procesables por una máquina, en lugar de la configuración física del hardware o el uso de configuración interactiva. Los archivos de definición pueden encontrarse en un sistema de control de versiones. IaC se basa en definiciones declarativas, en lugar de procesos manuales de gestión de la infraestructura.

Los enfoques de infraestructura como código se han ido extendiendo con la adopción del *cloud computing*, que a veces se comercializa como «infraestructura como servicio» (IaaS). IaC apoya IaaS, pero son conceptos distintos que no deben confundirse.

El valor de la infraestructura como código puede desglosarse en tres categorías medibles: coste, velocidad y riesgo. La reducción de costes tiene como objetivo ayudar no solo a la empresa financieramente, sino también en términos de personas y esfuerzo, lo que significa que mediante la eliminación del componente manual, las personas son capaces de reorientar sus esfuerzos hacia otras tareas empresariales. La automatización de la infraestructura permite una ejecución más rápida a la hora de configurar su infraestructura y tiene como objetivo proporcionar visibilidad para ayudar a otros equipos a trabajar de manera rápida y eficiente. La automatización elimina el riesgo asociado con el error humano, como la configuración errónea manual, y eliminarlo puede reducir

el tiempo de inactividad y aumentar la fiabilidad. Estos resultados y atributos ayudan a la empresa a avanzar hacia la implementación de una cultura de DevOps, el trabajo combinado de Desarrollo y Operaciones.

Generalmente, hay tres enfoques para IaC: declarativo o funcional, imperativo o procedimental e inteligente o consciente del entorno. El enfoque declarativo se centra en lo que debería ser la configuración final de destino. El imperativo se centra en cómo se va a cambiar la infraestructura para satisfacerla. Y el enfoque inteligente se centra en por qué la configuración debe ser una cierta manera en la consideración de todas las relaciones y dependencias de múltiples aplicaciones que se ejecutan en la misma infraestructura.

El enfoque declarativo define el estado deseado y el sistema ejecuta lo que necesita suceder para lograr ese estado deseado. El imperativo define comandos específicos que deben ejecutarse en el orden adecuado para finalizar con la conclusión deseada. Y el inteligente determina el estado correcto deseado antes de que el sistema ejecute lo que necesita suceder para lograr un estado deseado que no afecta a las aplicaciones dependientes.

La infraestructura como código es clave en la práctica de DevOps. Los desarrolladores se involucran más en la definición de la configuración y los equipos de operaciones se involucran más temprano en el proceso de desarrollo. Las herramientas que utilizan IaC aportan visibilidad al estado y configuración de los servidores y, en última instancia, brindan visibilidad a los usuarios dentro de la empresa, con el objetivo de reunir equipos para maximizar sus esfuerzos. La automatización, en general, tiene como objetivo tomar la confusión y el aspecto propenso a errores de los procesos manuales y hacerlo más eficiente y productivo. También, permitir que el software y las aplicaciones sean creados con flexibilidad, menos tiempo de inactividad y una forma globalmente rentable para la empresa. IaC pretende reducir la complejidad que mata la eficiencia de la configuración manual. La automatización y la colaboración se consideran puntos centrales en DevOps. Las herramientas de automatización de la infraestructura a menudo se incluyen como componentes de una cadena de herramientas de DevOps.

2.6.3. Automatización de la configuración continua

Todas las herramientas de automatización de configuración continua (CCA) pueden considerarse como una extensión de los marcos IaC tradicionales. Aprovecha IaC para cambiar, configurar y automatizar la infraestructura, pero también proporciona visibilidad, eficiencia y flexibilidad en el manejo de su infraestructura.

Un aspecto importante al considerar las herramientas de CCA, si son de código abierto, es el contenido de la comunidad. Como afirma Gartner, el valor de las herramientas de CCA es «tan dependiente del contenido y el apoyo aportados por la comunidad de usuarios como lo es la madurez comercial y

el rendimiento de la herramienta de automatización». Vendedores como Puppet y Chef han creado sus propias comunidades: Chef tiene Chef Community Repository y Puppet tiene PuppetForge; otros proveedores confían en comunidades adyacentes y aprovechan otros marcos IaC, como PowerShell DSC. Aparecen nuevos proveedores que no son impulsados por el contenido, sino por el modelo con la inteligencia del producto para ofrecer contenido. Estos sistemas visuales orientados a objetos funcionan bien para los desarrolladores, pero son especialmente útiles para operaciones orientadas a entornos de producción que valoran versiones de modelos de versiones de contenido. A medida que el campo continúa desarrollándose y cambiando, el contenido basado en la comunidad será cada vez más importante para la forma en que se utilizan las herramientas de IaC.

Las herramientas CCA incluyen: Ansible, CFEngine, Chef, Puppet, Otter o Saltstack.

2.6.4. Detalle de una herramienta: SaltStack

SaltStack es una herramienta de código abierto basada en Python para la gestión de configuración mediante un motor de ejecución remota. Da apoyo al enfoque de «Infraestructura como código» para la implementación del *cloud*.

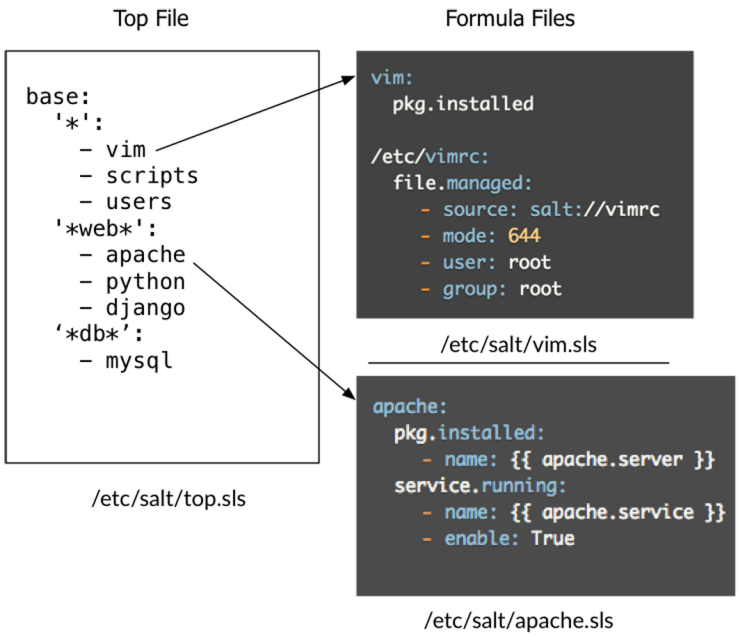
En cuanto a su arquitectura, se puede desplegar en modo *master*: un servidor *master* y sus correspondientes *minions* (agentes en los nodos esclavos); o bien en modo *masterless*. Si se opta por el modo *master*, es posible prescindir de los *minions*, en este caso el *master* se conecta a los nodos esclavos vía Secure Shell o SSH.

En cuanto a la configuración del propio SaltStack para la gestión de los diferentes *minions*, se compone de un fichero principal que hace referencia a otros ficheros de configuración específicos llamados fórmulas.

En el siguiente ejemplo vemos cómo todos los *minions* tienen definidas unas fórmulas por defecto y luego, dependiendo del rol del *minion* en cuestión, se le aplican unas fórmulas u otras. En concreto, a todos los *minions* se les aplican las fórmulas: vim, scripts y users; a los *minions* con rol web: apache, python y django; y a los *minions* con rol db: mysql.

Vemos que en las fórmulas se pueden instalar paquetes y aplicar ficheros de configuración, como en el caso de la fórmula para vim. También se pueden instalar y configurar servicios, como en el caso de apache, donde incluso se marca el servicio para que se ejecute en tiempo de arranque del *minion*.

Figura 14. Fichero principal y fórmulas de SaltStack



Enlace de interés

Si queréis aprender más sobre SaltStack, os recomendamos que lo instaléis y empecéis a practicar. En https://docs.saltstack.com/en/getstarted/fundamentals/install.html encontraréis una guía para la instalación inicial.

2.6.5. Infraestructura como código sobre diversos proveedores *cloud*

En el caso de estar gestionando infraestructura basada en proveedores de *cloud* público existen herramientas *cloud* agnósticas que nos pueden facilitar mucho la tarea.

Una herramienta cuyo uso está muy extendido es Terraform. Dicha herramienta se basa en el concepto de infraestructura inmutable.

El concepto de infraestructura inmutable deriva del concepto de programación «Inmutabilidad», es decir, que una vez se crea o se instancia algo, nunca será modificado. Esto significa que una vez la instancia esté funcionando, esta no se verá modificada. Cuando se requiera la aplicación de algún cambio, será creada una nueva instancia. Con esto conseguimos crear versiones del entorno completo de una forma controlada y segura; además, estas versiones pueden ser desplegadas de forma casi instantánea.

Terraform es una herramienta provista por Hashicorp, que nos permite realizar la codificación de la infraestructura dependiendo de nuestras necesidades, los servicios que queramos prestar y nos presenta una gran variedad de proveedores *cloud* donde alojar nuestra infraestructura.

Enlace de interés

Si queréis saber más sobre Terraform, podéis consultar el siguiente enlace: <https://www.terraform.io/>.

Con herramientas como Terraform, construir nuestra infraestructura se vuelve más fácil y manejable. Permite crear código con toda la infraestructura que necesitemos y conectar las diferentes partes, como si se tratase de asignaciones de variables en un lenguaje de programación.

2.7. Recomendaciones sobre gestión de los datos en entornos DevOps

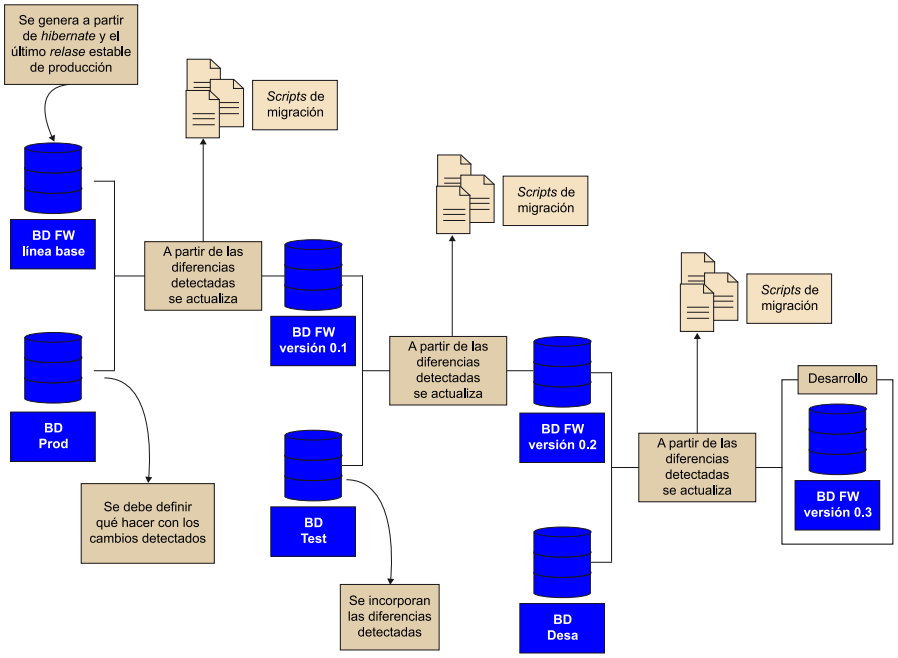
2.7.1. Versionado de la base de datos

Una buena práctica es la automatización de la gestión de las versiones mediante *scripts* de migración; estos:

- Inicializan el esquema, las credenciales y los datos.
- Realizan una copia temporal de tablas para facilitar el *roll-back* de datos.
- Utilizan una de las tablas para almacenar la versión actual de la base de datos.

Veamos un ejemplo del uso de los *scripts* de migración por los que pasamos de la versión base de base de datos FW hasta la versión 0.3 actual de desarrollo.

Figura 15. Ejemplo detallado del uso de *scripts* de migración



Fuente: <http://comunidad.iebschool.com/perezmvictoria/2014/05/01/proceso-del-versionado-para-bases-de-datos/>.

Dicha automatización permite la ejecución de *scripts* sucesivos para versionar hacia adelante (x + 1) o hacia atrás (x).

2.7.2. Objetivo: Evitar la pérdida de datos

En caso de *roll-back*, el objetivo principal es evitar la pérdida de datos. Algunos *scripts* de *roll-back* pueden perder datos. Para evitar esta situación, se puede:

- Grabar y reproducir las transacciones desde el «transaction log».
- Usar despliegues «blue-green». Los usuarios vuelven a la versión «blue» en caso de *roll-back* y se reproducen las transacciones nuevas de «green».
- Despliegues desacoplados de aplicación y datos. Permiten desplegar el cambio a la BD una vez que se ha estabilizado el cambio de versión.

2.7.3. Gestión de los datos de test

Para controlar el rendimiento de los test, tenemos:

- Test unitarios: uso de la base de datos en memoria, por ejemplo usando el patrón DAO.
- Test de aceptación: uso de *data-sets* moderados, inferiores a los de producción.

Acoplamiento entre test y datos: los juegos de pruebas deben inicializar el estado de los datos al comenzar y restaurarlo al acabar. Las estrategias más comunes son:

- **Aislamiento:** los test se pueden ejecutar en cualquier orden o en paralelo. Se puede conseguir utilizando las transacciones de la BD (*roll-backs*). Es la opción más flexible y segura.
- **Adaptativos:** los test calculan los resultados esperados de salida a partir del «estado» encontrado.
- **Secuencia:** cada test genera el estado de entrada del siguiente.

2.7.4. Gestión de datos de test en la Deployment pipeline

1) Sobre los datos para los test de commit

Son datos más bien livianos para que los test sean rápidos. Esto se consigue no sobreelaborando la definición de entradas y salidas para mantener la independencia sobre implementaciones concretas.

2) Sobre los datos para los test de aceptación

Son más realistas y complejos. Para gestionarlos de manera eficiente, se pueden agrupar como:

- Específicos del test: son creados y modificados por el test.
- Referenciados del test: son creados por la *suite* y consultados por el test:
  - Los datos para los test de capacidad.
  - Generales que requiere la aplicación para funcionar e independientes del test.

Cabe indicar que los test de aceptación requieren un gran volumen de datos de entrada y datos referenciados, y estos deben obtenerse a partir de datos y proporciones reales.

3. Infraestructura PaaS privada basada en contenedores: el caso Docker

3.1. Introducción a Docker

Docker es un proyecto de código abierto que permite automatizar el despliegue de aplicaciones dentro de «contenedores». Este contenedor empaqueta todo lo necesario para que uno o más procesos (servicios o aplicaciones) funcionen: código, herramientas del sistema, bibliotecas del sistema, dependencias, etc. Esto garantiza que siempre se podrá ejecutar, independientemente del entorno en el que queramos desplegarlo. No hay que preocuparse de qué software ni versiones tiene nuestra máquina, ya que nuestra aplicación se ejecutará en el contenedor.

3.1.1. Un poco de historia

Salomon Hykes comenzó Docker como un proyecto interno dentro de dotCloud, empresa enfocada a PaaS (plataforma como servicio). Fue liberado como código abierto en marzo de 2013. Con el lanzamiento de la versión 0.9 (en marzo de 2014) Docker dejó de utilizar LXC como entorno de ejecución por defecto y lo reemplazó con su propia librería, libcontainer (escrita en Go), que se encarga de hablar directamente con el *kernel*. Actualmente es uno de los proyectos con más estrellas en GitHub, con miles de *forks* y de colaboradores.

3.1.2. Requisitos mínimos

Docker funciona de forma nativa en entornos Linux a partir de la versión 3.8 del *kernel*. Algunos *kernels* a partir de la versión 2.6.x y posteriores podrían ejecutar Docker, pero los resultados pueden variar, por lo que oficialmente no está soportado. Otro requisito es que solo está preparado para arquitecturas de 64 bits (actualmente x86_64 y amd64). Para usar Docker en entornos Windows o MAC, han creado una herramienta, Boot2Docker, que no es más que una máquina virtual ligera de Linux con Docker ya instalado. Dicha imagen la arrancamos con Virtualbox, Vmware o con la herramienta que tengamos instalada.

Otra manera es con un instalador «todo en uno» para MAC y Windows. Este instalador contiene un cliente para Windows, la imagen de una máquina virtual Linux, Virtualbox y msys-git unix tools.

3.1.3. Características

Las principales características de Docker son:

- Portabilidad: el contenedor Docker podemos desplegarlo en cualquier sistema, sin necesidad de volver a configurarlo o realizar las instalaciones necesarias para que la aplicación funcione, ya que todas las dependencias son empaquetadas con la aplicación en el contenedor.

- Ligereza: los contenedores Docker solo contienen lo que los diferencia del sistema operativo en el que se ejecutan, no se virtualiza un SO completo.

- Autosuficiencia: un contenedor Docker no contiene todo un sistema operativo completo, solo aquellas librerías, archivos y configuraciones necesarias para desplegar las funcionalidades que contenga.

3.1.4. Ventajas y desventajas

Usar contenedores Docker permite a desarrolladores y administradores de sistemas probar aplicaciones o servicios en un entorno seguro e igual al de producción, lo que reduce los tiempos de pruebas y adaptaciones entre los entornos de prueba y producción.

Las principales ventajas de usar contenedores Docker son:

- Las instancias se inician en pocos segundos.

- Son fácilmente replicables.

- Es fácil de automatizar y de integrar en entornos de integración continua.

- Consumen menos recursos que las máquinas virtuales tradicionales.

- Mayor rendimiento que la virtualización tradicional, ya que corre directamente sobre el *kernel* de la máquina en la que se aloja, evitando al *hypervisor*.

- Ocupan mucho menos espacio.

- Permiten aislar las dependencias de una aplicación de las instaladas en el *host*.

- Existe un gran repositorio de imágenes ya creadas sobre miles de aplicaciones, que además pueden modificarse libremente.

Por todo esto, Docker ha entrado con mucha fuerza en el mundo del desarrollo, ya que permite desplegar las aplicaciones en el mismo entorno que tienen en producción o viceversa, permite desarrollarlas en el mismo entorno que tendrán en producción.

No obstante, también tiene algunas desventajas:

- Solo puede usarse de forma nativa en entornos Unix con *kernel* igual o superior a 3.8.
- Solo soporta arquitecturas de 64 bits.
- Como es relativamente nuevo, puede haber errores de código entre versiones.

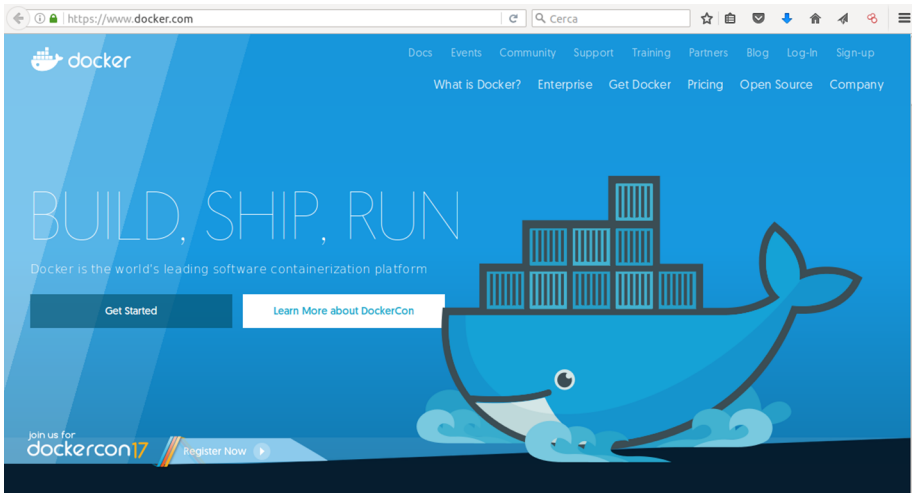
3.1.5. Usos y recomendaciones

El uso de Docker está recomendado:

- En entornos de integración continua, es decir, cuando el paso de desarrollo a producción en un proyecto sea lo más a menudo posible, para así poder detectar fallos cuanto antes.
- Para garantizar la integridad de las aplicaciones en diferentes entornos.
- Cuando necesitemos tener entornos fácilmente desplegables, portables y desechables.
- Cuando necesitemos un entorno fácilmente escalable.

3.1.6. Arquitectura

Docker usa una arquitectura cliente-servidor. El cliente de Docker habla con el demonio de Docker, que hace el trabajo de crear, correr y distribuir los contenedores. Ambos pueden ejecutarse en el mismo sistema, o se puede conectar un cliente a un demonio Docker remoto. El cliente Docker y el demonio se comunican vía *sockets* o a través de una RESTful API (imagen página oficial).

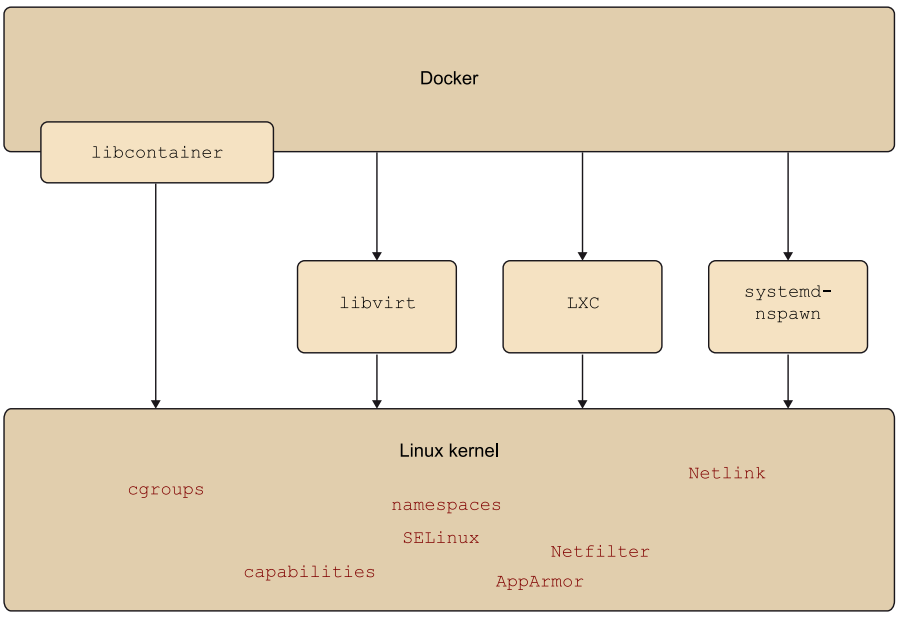


Expliquemos un poco el orden de la arquitectura o funcionamiento de Docker:

- El cliente de Docker (Docker Client) es la principal interfaz de usuario para Docker. Acepta comandos del usuario y se comunica con el demonio Docker.

- El demonio Docker (Docker Engine) corre en una máquina anfitriona (*host*). El usuario no interactúa directamente con el demonio, sino que lo hace a través del cliente Docker.

Figura 16. Representación gráfica de la arquitectura de Docker



Fuente: <https://en.wikipedia.org/wiki/Docker_(software)>.

El demonio Docker levanta los contenedores haciendo uso de las imágenes, que pueden estar en local o en el Docker Registry. Cada contenedor se crea a partir de una imagen y es un entorno aislado y seguro donde se ejecuta nuestra aplicación.

3.1.7. Componentes

Según la documentación oficial, Docker tiene dos componentes principales:

1) Docker. Plataforma *open source* de virtualización con contenedores.

2) Docker Hub. Plataforma de software como servicio (SaaS, *software-as-a-service*) para compartir y administrar contenedores Docker.

Pero también necesitamos conocer otros componentes y conceptos:

1) Docker Engine. Es el demonio que se ejecuta dentro del sistema operativo (Linux) y que expone una API para la gestión de imágenes, contenedores, volúmenes o redes. Sus funciones principales son:

- La creación de imágenes Docker.
- La publicación de imágenes en Docker Registry.
- La descarga de imágenes desde Docker Registry.
- La ejecución de contenedores usando las imágenes.
- La gestión de contenedores en ejecución (pararlo, arrancarlo, ver *logs*, ver estadísticas).

2) Docker Client. Se trata de cualquier software o herramienta que hace uso de la API del demonio Docker, pero suele ser el comando docker, que es la herramienta de línea de comandos para gestionar Docker Engine. Este cliente puede configurarse para hablar con un Docker local o remoto, lo que permite administrar nuestro entorno de desarrollo local como nuestros servidores de producción.

3) Docker Images. Son plantillas de solo lectura que contienen el sistema operativo base (más adelante entraremos en detalle en ello) donde correrá nuestra aplicación, además de las dependencias y el software adicional instalado, necesario para que la aplicación funcione correctamente. Las plantillas son usadas por Docker Engine para crear los contenedores Docker.

4) Docker Registries. Los registros de Docker guardan las imágenes. Pueden ser repositorios públicos o privados. El registro público lo provee el Hub de Docker, que sirve tanto imágenes oficiales como las subidas por usuarios con sus propias aplicaciones y configuraciones. Así, tenemos disponibles para todos los usuarios imágenes oficiales de las principales aplicaciones (MySQL, MongoDB, Apache, Tomcat, etc.) y no oficiales de infinidad de aplicaciones

y configuraciones. DockerHub ha supuesto una gran manera de distribuir las aplicaciones. Es un proyecto *open source* que puede ser instalado en cualquier servidor. Además, nos ofrecen un sistema SaaS de pago.

5) Docker Containers. El contenedor de Docker aloja todo lo necesario para ejecutar un servicio o aplicación. Cada contenedor es creado de una imagen base y es una plataforma aislada. Un contenedor es simplemente un proceso para el sistema operativo, que se aprovecha de él para ejecutar una aplicación. Dicha aplicación solo tiene visibilidad sobre el sistema de ficheros virtual del contenedor.

6) Docker Compose. Es otro proyecto *open source* que permite definir aplicaciones multicontenedor de una manera sencilla. Es una alternativa más cómoda al uso del comando docker run, para trabajar con aplicaciones con varios componentes. Es una buena herramienta para gestionar entornos de desarrollo y de pruebas o para procesos de integración continua.

7) Docker Machine. Es un proyecto *open source* para automatizar la creación de máquinas virtuales con Docker instalado, en entornos Mac, Windows o Linux, pudiendo administrar así un gran número de máquinas Docker. Incluye *drivers* para Virtualbox, que es la opción aconsejada para instalaciones de Docker en local, en vez de instalar Docker directamente en el *host*. Esto simplifica y facilita la creación o la eliminación de una instalación de Docker, facilita la actualización de la versión de Docker o trabajar con distintas instalaciones a la vez.

Usando el comando docker-machine podemos iniciar, inspeccionar, parar y reiniciar un *host* administrado, actualizar el Docker client y el Docker daemon, y configurar un cliente para que hable con el *host* anfitrión. A través de la consola de administración podemos administrar y correr comandos Docker directamente desde el *host*. Este comando docker-machine automáticamente crea *hosts*, instala Docker Engine en ellos y configura los clientes Docker.

3.1.8. Diferencias con las máquinas virtuales

La principal diferencia es que una máquina virtual necesita tener virtualizado todo el sistema operativo, mientras que el contenedor Docker aprovecha el sistema operativo sobre el que se ejecuta, compartiendo el *kernel* e incluso parte de sus bibliotecas. Para el SO anfitrión, cada contenedor no es más que un proceso que corre sobre el *kernel*.

El concepto de contenedor o «virtualización ligera» no es nuevo. En Linux, LXC (Linux Containers) es una tecnología de virtualización a nivel de sistema operativo que utiliza dos características del *kernel*, cgroups (que permite aislar y rastrear el uso de recursos) y namespaces (que permite a los grupos separarse, así no pueden verse unos a otros), para poder ejecutar procesos ligeros independientes en una sola máquina, aislados unos de otros, cada uno con su

propia configuración de red. Ejemplos de esto son las jaulas de FreeBSD, OpenSolaris o Linux Vservers. Otra diferencia es el tamaño: una máquina virtual convencional puede ocupar bastante, sin embargo, los contenedores Docker solo contienen lo que las diferencia del sistema operativo en el que se ejecutan, y ocupan una media de 150-250 Mb. En cuanto a recursos, el consumo de procesador y memoria RAM es mucho menor al no estar todo el sistema operativo virtualizado.

3.2. Instalación de Docker

Podemos distinguir entre la instalación para el desarrollo local y la instalación en servidores en producción. Para los servidores en producción, la mayoría de los proveedores de servicio (AWS, GCE, Azure, Digital Ocean...) disponen de máquinas virtuales con versiones de Docker preinstaladas.

En nuestro caso vamos a instalar Docker en Linux bajo la distribución Ubuntu, pero en la página oficial vienen guías de instalación para múltiples distribuciones Linux (Ubuntu, Red Hat, CentOS, Fedora, Arch Linux, Gentoo, Oracle Linux, etc.).

3.2.1. Instalación en Ubuntu

En la documentación oficial podemos ver las distribuciones de Ubuntu que están soportadas. Aquí vamos a instalar Docker en Ubuntu (LTS).

Primero vamos a comprobar los requisitos, que son arquitectura de 64 bits y un *kernel* igual o superior a 3.10.

```
root@cloud-jguijarro:~# uname -a
Linux cloud-jguijarro 3.13.0-44-generic #73-Ubuntu SMP Tue Dec 16 00:22:43 UTC 2014 x86_64
x86_64 x86_64 GNU/Linux
```

Los pasos que deberemos seguir para la instalación son estos:

1) Actualizamos los paquetes del sistema:

```
root@cloud-jguijarro:~# apt-get update
```

2) Instalamos los paquetes para https y certificados CA:

```
root@cloud-jguijarro:~# apt-get install apt-transport-https ca-certificates
```

3) Añadimos una nueva clave GPG:

```
root@cloud.jguijarro:~# sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
--recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

4) Abrimos /etc/apt/sources.list.d/docker.list. Si apt-get no existe, lo creamos, y si existe, borramos su contenido. Y añadimos el repositorio:

```
deb https://apt.dockerproject.org/repo ubuntu-trusty main
```

5) Actualizamos la lista de paquetes:

```
root@cloud-jguijarro:~# apt-get update
```

6) Si tuviéramos una versión antigua, deberíamos eliminarla:

```
root@cloud-jguijarro:~# apt-get purge lxc-docker
```

7) Comprobamos el candidato para la instalación:

Figura 17. Detalle del paquete candidato para la instalación de Docker

```
root@cloud-jguijarro:~# apt-cache policy docker-engine | more
docker-engine:
  Installat: (cap)
  Candidat:  1.12.3-0~trusty
  Taula de versió:
     1.12.3-0~trusty 0
        500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Package
s
```

Fuente: Jordi Guijarro.

8) La documentación oficial recomienda para nuestra versión de Ubuntu instalar el paquete linux-image-extra, que permite utilizar los *drivers* de almacenamiento aufs (AnotherUnionFS), que es una versión alternativa de UnionFS, un servicio de archivos que implementa una unión para montar sistemas de archivos Linux.

Y también recomienda instalar el paquete apparmor, que es una extensión de seguridad que provee de una variedad de políticas de seguridad para el *kernel* de Linux. Es una alternativa a SELinux.

```
root@cloud-jguijarro:~# apt-get install linux-image-extra-3.13.0.65-generic apparmor
```

9) Instalamos Docker-Engine:

```
root@cloud-jguijarro:~# apt-get install docker-engine
```

10) Iniciamos el demonio Docker:

```
root@cloud-jguijarro:~# service docker start
```

11) Comprobamos que Docker está instalado correctamente:

```
root@cloud-jguijarro:~# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
```

```
c04b14da8d14: Pull complete
Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

3.2.2. Configuración opcional

La documentación oficial nos ofrece una serie de configuraciones y procedimientos para que Ubuntu trabaje mejor con Docker.

1) Crear el grupo Docker si no existe.

El demonio Docker se comunica a través del *socket* de Unix en vez de por un puerto TCP.

Por defecto, el *socket* Unix es propiedad del usuario *root*, por lo que el demonio Docker siempre corre como el usuario *root*.

Para evitar tener que usar sudo cuando se utilice el comando docker, se crea un grupo llamado Docker y se añaden usuarios a él. Cuando el demonio Docker se inicie con un usuario perteneciente al grupo Docker, lo hará con los permisos de lectura y escritura equivalentes a *root*:

```
root@cloud-jguijarro:~# groupadd docker
root@cloud-jguijarro:~# usermod -aG docker user1
```

Reiniciamos la sesión del usuario 'user1' y comprobamos:

```
ubuntu@cloud-jguijarro:~$ docker run hello-world
Hello from Docker. This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that
produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.
To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker Hub account:
https://hub.docker.com
For more examples and ideas, visit:
https://docs.docker.com/engine/userguide/
ubuntu@cloud-jguijarro:~$
```

2) Ajustar la memoria y la *swap*.

Dice la documentación que a veces aparece un mensaje de *warning* de la memoria *swap* para Cgroup. Para evitar estos mensajes, activamos la memoria de intercambio en nuestro sistema.

Editamos el fichero /etc/default/grub y actualizamos el valor GRUB_CMDLINE_LINUX:

```
GRUB_DEFAULT=0
GRUB_HIDDEN_TIMEOUT=0
GRUB_HIDDEN_TIMEOUT_QUIET=true

GRUB_TIMEOUT=0
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="console=tty1 console=ttyS0"
GRUB_CMDLINE_LINUX=""
```

Y lo dejamos:

```
GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
```

Luego actualizamos GRUB y reiniciamos el sistema.

```
root@cloud-jguijarro:/home/ubuntu# update-grub
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-3.13.0-65-generic
Found initrd image: /boot/initrd.img-3.13.0-65-generic
done
```

3) Configurar las políticas del *firewall*.

Docker usa un *bridge* para gestionar las redes de contenedores, por lo que si tenemos un *firewall* (UFW) que por defecto elimine el tráfico de *forwarding*, habrá que configurarlo adecuadamente.

Comprobamos el estado de nuestro *firewall*:

```
root@cloud-jguijarro:/home/ubuntu# ufw status
Status: inactive
```

En este caso está inactivo. Si estuviera activo, los pasos que deberíamos seguir serían los siguientes:

a) Editar el fichero /etc/default/ufw y actualizar la siguiente política a "ACCEPT":

```
DEFAULT_FORWARD_POLICY="ACCEPT"
```

b) Reiniciar el *firewall* y permitir las conexiones entrantes en el puerto de Docker:

```
$ sudo ufw reload
$ sudo ufw allow 2375/tcp
```

4) Configurar un servidor DNS para Docker.

Sistemas como Ubuntu generalmente usan la 127.0.0.1 como servidor DNS por defecto en el fichero /etc/resolv.conf. El NetworkManager también configura el dnsmasq para que use el servidor local. Al iniciar contenedores en máquinas con esta configuración, Docker nos mostrará la siguiente advertencia:

```
WARNING: Local (127.0.0.1) DNS resolver found in resolv.conf and containers
can't use it. Using default external servers : [8.8.8.8 8.8.4.4]
```

Esto ocurre porque los contenedores Docker no usan el DNS local, sino que usan uno externo. Para evitar esta advertencia, podemos especificar un servidor DNS para que lo usen los contenedores. Esto se hace editando el fichero /etc/default/docker y especificando uno o más servidores DNS públicos de Google en la siguiente línea:

```
DOCKER_OPTS="--dns 8.8.8.8"
```

Y reiniciamos el demonio Docker.

Si usáramos NetworkManager con dnsmasq, tendríamos que deshabilitarlo comentando en el fichero /etc/NetworkManager/NetworkManager.conf la línea:

```
# dns=dnsmasq
```

Y reiniciando NetworkManager y Docker.

5) Configurar Docker para iniciar en el arranque.

Ubuntu usa systemd como su gestor de arranque. Para configurar el demonio de Docker para arrancar en el inicio:

```
$ systemctl enable docker
```

3.3. Administración de Docker

Ya tenemos instalado Docker en nuestra máquina anfitriona cuyo SO es Ubuntu 14.04 LTS. Si escribimos «docker version» en la línea de comandos, nos da la versión del cliente y del servidor, además de la versión de la API y de Go (lenguaje de programación en el que está escrito Docker):

```
root@cloud-jguijarro:#docker version
Client:
 Version:  1.12.3
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   6b644ec
 Built:    Wed Oct 26 21:44:32 2016
 OS/Arch:  linux/amd64
Server:
 Version:  1.12.3
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   6b644ec
 Built:    Wed Oct 26 21:44:32 2016
 OS/Arch:  linux/amd64
```

Con «docker –v» vemos solo la versión de Docker instalada:

```
root@cloud-jguijarro:~# docker -v
Docker version 1.12.3, build 6b644ec
```

3.3.1. Principales comandos Docker

Antes de empezar a usar Docker en la máquina que hemos preparado, vamos a familiarizarnos con los comandos que nos ofrece. Escribiendo «docker» en la terminal nos aparece una lista de las opciones disponibles:

- attach: para acceder a la consola de un contenedor que está corriendo.
- build: construye un contenedor a partir de un Dockerfile.
- commit: crea una nueva imagen de los cambios de un contenedor.
- cp: copia archivos o carpetas desde el sistema de ficheros de un contenedor al *host*.
- create: crea un nuevo contenedor.
- daemon: crea un proceso demonio.
- diff: comprueba cambios en el sistema de ficheros de un contenedor.
- events: muestra eventos en tiempo real del estado de un contenedor.
- exec: ejecuta un comando en un contenedor activo.
- export: exporta el contenido del sistema de ficheros de un contenedor a un archivo .tar.
- history: muestra el historial de una imagen.
- images: lista las imágenes que tenemos descargadas y disponibles.
- import: crea una nueva imagen del sistema de archivos vacío e importa el contenido de un fichero .tar.
- info: muestra información sobre los contenedores, imágenes, versión de Docker.
- inspect: muestra informaciones de bajo nivel del contenedor o la imagen.
- kill: detiene a un contenedor activo.
- load: carga una imagen desde un archivo .tar.
- login: para registrarse en un servidor de registro de Docker, por defecto «https://index.docker.io/v1/».

- logout: se desconecta del servidor de registro de Docker.
- logs: obtiene los registros de un contenedor.
- network connect: conecta un contenedor a una red.
- network create: crea una nueva red con un nombre especificado por el usuario.
- network disconnect: desconecta un contenedor de una red.
- network inspect: muestra información detallada de una red.
- network ls: lista todas las redes creadas por el usuario.
- network rm: elimina una o más redes.
- pause: pausa todos los procesos dentro de un contenedor.
- port: busca el puerto público, el cual está nateado, y lo hace privado.
- ps: lista los contenedores.
- pull: descarga una imagen o un repositorio del servidor de registros Docker.
- push: envía una imagen o un repositorio al servidor de registros de Docker.
- rename: renombra un contenedor existente.
- restart: reinicia un contenedor activo.
- rm: elimina uno o más contenedores.
- rmi: elimina una o más imágenes.
- run: ejecuta un comando en un nuevo contenedor.
- save: guarda una imagen en un archivo .tar.
- search: busca una imagen en el índice de Docker.
- start: inicia un contenedor detenido.
- stats: muestra el uso de los recursos de los contenedores.
- stop: detiene un contenedor.

- tag: etiqueta una imagen en un repositorio.
- top: busca los procesos en ejecución de un contenedor.
- unpause: reanuda un contenedor pausado.
- update: actualiza la configuración de uno o más contenedores.
- version: muestra la versión de Docker instalada.
- volume create: crea un volumen.
- volume inspect: devuelve información de bajo nivel de un volumen.
- volume ls: lista los volúmenes.
- volume rm: elimina un volumen.
- wait: bloquea hasta detener un contenedor, entonces muestra su código de salida.

Y si queremos saber cómo funciona un comando concreto, debemos escribir «docker COMMAND --help». Veamos, por ejemplo, cómo usar el comando save:

```
ubuntu@cloud-jguijarro:~$ docker save --help
Usage: docker save [OPTIONS] IMAGE [IMAGE...]
Save one or more images to a tar archive (streamed to STDOUT by default)
--help Print usage
-o, --output Write to a file, instead of STDOUT
```

3.3.2. Imágenes

Las imágenes son plantillas de solo lectura que usamos como base para lanzar un contenedor. Una imagen Docker se compone de un sistema de archivos en capas una sobre la otra. En la base tenemos un sistema de archivos de arranque, bootfs (parecido al sistema de archivos de Linux), sobre el que arranca la imagen base. Cada imagen, también conocida como repositorio, es una sucesión de capas. Es decir, al arrancar un contenedor lo hacemos sobre una imagen, a la que llamamos imagen base. Con el contenedor corriendo, cada vez que realizamos un cambio en el contenedor Docker añade una capa encima de la anterior con los cambios, pero dichas modificaciones no serán persistentes, los cambios no los hacemos en la imagen (recordemos que es de solo lectura), por lo que deberemos guardarlos creando una nueva imagen con los cambios.

Docker monta un sistema de ficheros de lectura y escritura en la parte superior de las capas. Aquí es donde los procesos de nuestro contenedor Docker serán ejecutados (container). Cuando Docker crea un contenedor por primera vez, la capa inicial de lectura y escritura está vacía. Cuando se producen cambios, estos se aplican a esta capa, por ejemplo, si desea cambiar un archivo, entonces ese archivo se copia desde la capa de solo lectura inferior a la capa de lectura y escritura. Seguirá existiendo la versión de solo lectura del archivo, pero ahora está oculto debajo de la copia.

3.3.3. Trabajando con imágenes

Como hemos explicado anteriormente, los contenedores se construyen a partir de imágenes, que se pueden encontrar en local, en el Docker Hub o en repositorios privados. Más adelante veremos Docker Hub con un poco más de detalle.

Para comprobar las imágenes que tenemos en local, hemos de ejecutar:

```
docker images
```

Figura 18. Detalle de las imágenes de Docker

Fuente: Jordi Guijarro.

Ahora mismo solo tenemos la imagen que se descargó cuando comprobamos que se había instalado correctamente Docker. Podemos buscar la imagen que queremos con el comando search. Por ejemplo, si queremos saber qué imágenes hay disponibles de Ubuntu.

Vemos una lista con las imágenes disponibles.

Figura 19. Detalle de la búsqueda de la imagen de wordpress

```
root@cloud-jguijarro:~# docker search wordpress
NAME                            DESCRIPTION                                     ST
ARS     OFFICIAL   AUTOMATED
wordpress                       The WordPress rich content management syst...   13
96      [OK]
appcontainers/wordpress         Centos/Debian Based Customizable Wordpress...   34
                  [OK]
bitnami/wordpress               Bitnami Docker Image for WordPress              18
                  [OK]
centurylink/wordpress           Wordpress image with MySQL removed.             12
                  [OK]
kaihofstetter/wordpress-cli     Installs a configured and ready to use Wor...   12
                  [OK]
trafex/wordpress                Wordpress container with Nginx 1.10 & PHP-...   3
                  [OK]
maximiliend/wordpress           A enhanced docker image of WordPress with ...   3
                  [OK]
dsifford/wordpress              A WordPress docker environment that just w...   2
                  [OK]
scjalliance/wordpress           WordPress with GD and FreeType                  2
                  [OK]
dsteinkopf/wordpress            wordpress clone plus some php extensions        1
                  [OK]
ddaishin/wordpress              Wordpress日本語最新版  Xdebug  OSX...
```

Fuente: Jordi Guijarro.

Para descargar una imagen, debemos hacerlo con el comando pull (también podemos hacerlo con el comando run, como veremos más adelante, pero esta es la forma correcta).

Sintaxis:

```
docker pull [options] NAME[:TAG] |
[REGISTRY_HOST[:REGISTRY_PORT]/]NAME[:TAG]
```

Donde las opciones disponibles son:

- -a, --all-tags: descarga todas imágenes etiquetadas en el repositorio.
- --disable-content-trust=true: se salta la verificación de la imagen.
- --help: muestra ayuda sobre el comando.

Si no especificamos el *tag*, se descargará la última versión. Vamos a probarlo.

Figura 20. Detalle de la descarga de una imagen Docker

```
root@cloud-jguijarro:~# docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
aed15891ba52: Pull complete
773ae8583d14: Pull complete
d1d48771f782: Pull complete
cd3d6cd6c0cf: Pull complete
8ff6f8a9120c: Pull complete
Digest: sha256:35bc48a1ca97c3971611dc4662d08d131869daa692acb281c7e9e052924e38b1
Status: Downloaded newer image for ubuntu:latest
```

Fuente: Jordi Guijarro.

Si nos fijamos en la descarga, vemos 5 líneas que ponen «pull complete». Esto es que la imagen está formada por 5 capas o *layers*. Estas capas pueden ser re-utilizadas por otras imágenes, que evitan así el tener que volver a descargarlas, por ejemplo si descargamos otra imagen de Ubuntu. En la descarga también vemos una línea que pone Digest. Este código sirve si queremos asegurarnos

de usar una versión de la imagen en concreto, y no la última por ejemplo, como sucede si usamos el nombre y el *tag*. Comprobamos las imágenes disponibles ahora.

Figura 21. Detalle de las imágenes de Docker

```
root@cloud-jguijarro:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
ubuntu              latest              e4415b714b62        9 days ago
128.1 MB
hello-world         latest              c54a2cc56cbb        4 months ago
1.848 kB
root@cloud-jguijarro:~#
```

Fuente: Jordi Guijarro.

Para ver los detalles de una imagen, ejecutamos el siguiente comando:

Figura 22. Consulta de los detalles de una imagen de Docker

```
root@cloud-jguijarro:~# docker inspect ubuntu
[
    {
        "Id": "sha256:e4415b714b624040f19f45994b51daed5cbdb00e0eb9a07221ff0bd6bcf55ed7",
        "RepoTags": [
            "ubuntu:latest"
        ],
        "RepoDigests": [
            "ubuntu@sha256:35bc48a1ca97c3971611dc4662d08d131869daa692acb281c7e9e052924e38b1"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2016-11-16T20:58:26.830045089Z",
        "Container": "c450b5337e9f0b1e408f45d76031d56219542c7ebe416c6d872694abcb7e5a33",
        "ContainerConfig": {
            "Hostname": "fb8ca5e4ccd2",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"/bin/bash\"]"
            ],
            "ArgsEscaped": true,
            "Image": "sha256:53ba9532a0d83bd4db49e5dbb11a2db0b5e9004669a030143a2c0a3f251bf900",
```

Fuente: Jordi Guijarro.

Una vez que tenemos disponible una imagen, podemos ejecutar cualquier contenedor.

3.3.4. Ejecución de contenedores

Comando run

Podemos crear un contenedor con el comando run.

Sintaxis:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG....]
```

Entre las opciones se encuentran:

- -a, --attach: para conectarnos a un contenedor que está corriendo.

- -d, --detach: corre un contenedor en segundo plano.
- -i, --interactive: habilita el modo interactivo.
- --name: le pone nombre a un contenedor.

Ahora vamos a ejecutar un contenedor sobre la imagen de ubuntu que tenemos descargada.

```
ubuntu@cloud-jguijarro:~$ docker run ubuntu echo hello world
hello world
```

El comando run primero crea una capa del contenedor sobre la que se puede escribir y, a continuación, ejecuta el comando especificado. Con este comando hemos ejecutado un contenedor, sobre la imagen Ubuntu, que ha ejecutado el comando echo. Cuando ha terminado de ejecutar el comando que le hemos pedido se ha detenido. Los contenedores están diseñados para correr un único servicio, aunque podemos correr más si hiciera falta. Cuando ejecutamos un contenedor con run, debemos especificarle un comando que ejecutar en él, y dicho contenedor solo se ejecuta durante el tiempo que dura el comando que especifiquemos, funciona como un proceso.

Algo que hemos de tener en cuenta es que si la imagen que estamos poniendo en el comando run no la tuviéramos en local, Docker primero la descargaría y la guardaría en local y luego seguiría con la construcción capa por capa del contenedor. Vamos a ver los contenedores que tenemos en nuestro *host* con el comando ps.

Figura 23. Consulta de los contenedores Docker activos en nuestro *host*

```
root@cloud-jguijarro:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
root@cloud-jguijarro:~#
```

Fuente: Jordi Guijarro.

No nos aparece ningún contenedor, y esto es porque el contenedor ya no está activo. Para ver todos los contenedores (activos e inactivos) usamos el flag -a.

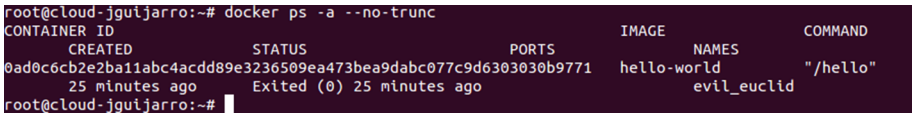
Figura 24. Consulta de todos los contenedores Docker en nuestro *host*

```
root@cloud-jguijarro:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
       PORTS               NAMES
0ad0c6cb2e2b        hello-world         "/hello"            24 minutes ago      Exited (0) 24 minute
s ago                       evil_euclid
```

Fuente: Jordi Guijarro.

Vemos el ID del contenedor, la imagen del contenedor, el comando que se ha ejecutado, cuándo se creó, el estado, el mapeo de puertos y el nombre. Este ID es abreviado; para verlo completo, tenemos que ejecutar:

Figura 25. Consulta de todos los detalles de los contenedores Docker en nuestro *host*



Fuente: Jordi Guijarro.

También vemos que Docker genera automáticamente un nombre al azar por cada contenedor que creamos. Si queremos especificar un nombre en particular, podemos hacerlo con el parámetro --name.

```
ubuntu@cloud-jguijarro:~$ docker run --name micontainer ubuntu echo prueba nombre
```

Con «docker ps –a» veremos el contenedor con el nombre que hemos puesto.

Modo interactivo

Como hemos visto, cuando creamos un contenedor con run, debemos especificar un comando que se va a ejecutar, y cuando se acabe su ejecución el contenedor se detendrá.

Básicamente, el contenedor se crea para ejecutar dicho comando.

Tenemos la opción de ejecutar un contenedor en modo interactivo con los *flags*:

- -t: ejecuta una terminal.
- -i: nos comunicamos con el contenedor en modo interactivo.

Como vemos en la imagen, estamos conectados al contenedor. Si abrimos otra terminal y ejecutamos un docker ps, vemos que tenemos el contenedor corriendo.

Pero si nos salimos de la *shell* del contenedor, este se detendrá:

```
ubuntu@cloud-jguijarro:~$ docker run -it ubuntu /bin/bash
```

O conectarnos a él vía otro terminal:

```
ubuntu@cloud-jguijarro:~$ docker attach tiny_turing
root@dfb581bd638c:/#
```

O ejecutar con exec comandos dentro de un contenedor activo, por ejemplo una *shell*:

```
ubuntu@cloud-jguijarro:~$ docker exec -it dfb /bin/sh
# exit
ubuntu@cloud-jguijarro:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED
```

```
STATUS PORTS NAMES
dfb581bd638c ubuntu "/bin/bash" 3 hours ago
Up 19 seconds tiny_turing
```

Vemos que cuando salimos el contenedor no se detiene. Podemos abrir otra terminal y parar el contenedor con su identificador «docket stop 635cacb8d75c».

Pero no resulta muy funcional un contenedor que se pare cuando terminemos de actuar sobre él. Nos interesa que continúe ejecutándose con el servicio que queramos. Para ello, está el modo detached, que veremos un poco más adelante.

3.3.5. Creando imágenes Docker

Las imágenes, como hemos visto, son plantillas de solo lectura, que usamos de base para lanzar contenedores. Por tanto, lo que hagamos en el contenedor solo persiste en ese contenedor, las modificaciones no las hacemos en la imagen.

Si queremos que dichos cambios sean permanentes, debemos crear una nueva imagen con el contenedor personalizado. Veamos las imágenes que tenemos disponibles:

```
ubuntu@cloud-jguijarro:~$ docker images
```

Vamos a realizar un ejemplo. Tenemos la imagen base de ubuntu. Lanzamos un contenedor con esa imagen base en el modo interactivo que vimos anteriormente e instalamos una serie de paquetes:

```
root@cloud-jguijarro:~# docker run -it --name micontainer ubuntu /bin/bash
root@635cacb8d77c:/# apt-get update
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [94.5 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-security InRelease [94.5 kB]
Get:4 http://archive.ubuntu.com/ubuntu xenial/main Sources [1103 kB]
Get:5 http://archive.ubuntu.com/ubuntu xenial/restricted Sources [5179 B]
...
...
```

Instalamos por ejemplo git.

```
root@635cacb8d77c:/# apt-get install git
```

Ahora vamos a guardar los cambios realizados en la imagen. Tenemos que salir del contenedor y ejecutar el comando «commit». Cuando salimos de un contenedor interactivo este se detiene. Lo vemos con docker ps –a. Ahora mismo este contenedor está formado por la capa con la imagen base y la capa en la que hemos instalado git.

Para poder utilizar esta imagen con los cambios, hemos de crear una nueva imagen, con el comando:

```
docker commit -m "Conf realizada" -a "jguijarro" 635cacb8d77c
git/ubuntu:v1
```

- Con commit creamos una nueva imagen en nuestro repositorio local.
- -m: añadimos un comentario.
- -a: autor de la imagen.
- 635cacb8d77c: identificador del contenedor.
- git/ubuntu:v1: el nombre que le damos a la imagen.

Al crearla nos devuelve un identificador único de imagen. La podemos ver en la lista de imágenes mediante el comando «docker images».

Figura 26. Consulta de las imágenes Docker disponibles en nuestro *host*

```
root@cloud-jguijarro:~# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
git/ubuntu          v1                  d6a5d5bdd503        11 seconds ago      258.1 MB
ubuntu              latest              e4415b714b62        10 days ago         128.1 MB
hello-world         latest              c54a2cc56cbb        4 months ago        1.848 kB
```

Fuente: Jordi Guijarro.

A partir de aquí podemos crear un contenedor con esta nueva imagen como base y ya tendrá instalado git. Lo comprobamos:

Figura 27. Detalle del nuevo contenedor con git instalado

```
root@cloud-jguijarro:~# docker run -it --name git git/ubuntu:v1 /bin/bash
root@dc36b5fc4267:/#
root@dc36b5fc4267:/# dpkg -l | grep git
ii  findutils                4.6.0+git+20160126-2           amd64        utilities for finding fi
les--find, xargs
ii  git                      1:2.7.4-0ubuntu1               amd64        fast, scalable, distribu
ted revision control system
ii  git-man                  1:2.7.4-0ubuntu1               all          fast, scalable, distribu
ted revision control system (manual pages)
ii  libasn1-8-heimdal:amd64  1.7~git20150920+dfsg-4ubuntu1  amd64        Heimdal Kerberos - ASN.1
 library
ii  libgssapi3-heimdal:amd64 1.7~git20150920+dfsg-4ubuntu1  amd64        Heimdal Kerberos - GSSAP
I support library
ii  libhcrypto4-heimdal:amd64 1.7~git20150920+dfsg-4ubuntu1 amd64        Heimdal Kerberos - crypt
o library
ii  libheimbase1-heimdal:amd64 1.7~git20150920+dfsg-4ubuntu1 amd64       Heimdal Kerberos - Base
```

Fuente: Jordi Guijarro.

3.3.6. Detached o Background

Nos permite ejecutar un contenedor en segundo plano y poder correr comandos sobre él en cualquier momento mientras esté en ejecución. Lo hacemos con el *flag* -d. Se dice que es un contenedor demonizado y se ejecutará indefinidamente. Vamos a ver un ejemplo. Primero creamos un contenedor basado en la imagen ubuntu que hemos descargado en modo interactivo:

```
root@cloud-jguijarro:~# docker run -it --name 2plano ubuntu /bin/bash
root@fc6e05dedc96:/#
```

Y le instalamos un servidor web:

```
root@cloud-jguijarro:~# docker run -it --name apache ubuntu /bin/bash
root@2ca26b7f9d5c:/# apt-get install apache2
```

Ahora creamos una nueva imagen del contenedor con Apache instalado:

```
root@cloud-jguijarro:~# docker commit -m "Instalado Apache" -a "Jguijarro"
2ca26b7f9d5c apache/ubuntu:v1
sha256:defab75e4eb991475e943513f52002b76f5eaa65b5d69c998228c469e99093ab
```

Ahora podemos arrancar un contenedor en segundo plano:

```
root@cloud-jguijarro:~# docker run -d --name server apache/ubuntu:v1
/usr/sbin/apache2ctl -DFOREGROUND
9bb15ff58f5da24e2513e69f0c4301577eb10dd07c173729d472a94b8b20234d
```

Vamos a hacer que el servicio Apache no se detenga con -D y que el contenedor se ejecute en segundo plano con -d.

Comprobamos nuestros contenedores:

```
root@cloud-jguijarro:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES
9bb15ff58f5d apache/ubuntu:v1 "/usr/sbin/apache2ctl" 24
seconds ago Up 24 seconds server
```

Podemos ver los procesos que se están ejecutando en un contenedor con «top»:

```
root@cloud-jguijarro:~# docker top web_server
UID PID PPID C
STIME TTY TIME CMD
root 10543 10529 0
15:19 ? 00:00:00 /bin/sh
/usr/sbin/apache2ctl -DFOREGROUND
root 10564 10543 0
15:19 ? 00:00:00
/usr/sbin/apache2 -DFOREGROUND
www-data 10565 10564 0
15:19 ? 00:00:00
/usr/sbin/apache2 -DFOREGROUND
www-data 10566 10564 0
15:19 ? 00:00:00
```

```
/usr/sbin/apache2 -DFOREGROUND
```

Una vez que hemos creado el contenedor y lo tenemos corriendo en segundo plano, podemos conectarnos a él mediante el comando exec, que ejecuta un proceso en el contenedor:

```
root@cloud-jguijarro:~# docker exec -it server /bin/bash
root@5659f2d7e356:/#
```

Vemos que, al salirnos del contenedor, este no se detiene:

```
root@5659f2d7e356:/# exit
exit
root@docker:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES
5659f2d7e356 apache/ubuntu:v1 "/usr/sbin/apache2ctl" 4
minutes ago Up 4 minutes server
```

3.3.7. Mapeo de puertos

Para que la aplicación que nos está sirviendo nuestro contenedor, por ejemplo el servidor web instalado en el contenedor anterior, es necesario mapear los puertos. Es decir, al crear el contenedor, este no está disponible al exterior. Para que lo esté, tenemos que redireccionar el puerto 22 del contenedor a uno de nuestra máquina. Usamos para ello el *flag* -p.

Vamos a ejecutar un contenedor basado en la imagen apache, en segundo plano y con la redirección de puertos activa. Podemos hacer la redirección de puertos con las opciones:

- -p: especificamos a qué puerto queremos la redirección.

- -P: le dice a Docker que si expone algún tipo de puerto haga el *fordwarding* a un puerto aleatorio.

Este se una cuando usamos un Dockerfile.

Lanzamos el contenedor:

```
root@cloud-jguijarro:~# docker run -d -p 8000:80 --name apache_2
apache/ubuntu:v1 /usr/sbin/apache2ctl -D FOREGROUND
```

Comprobamos su estado:

```
root@cloud-jguijarro:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED
```

```
STATUS PORTS NAMES
c1fbe4ea1b9e apache/ubuntu:v1 "/usr/sbin/apache2ctl" 9
seconds ago Up 8 seconds apache_3
1d47feeb39ec apache/ubuntu:v1 "/usr/sbin/apache2ctl" 48
seconds ago Up 47 seconds 0.0.0.0:8000->80/tcp apache_2
```

Vemos que nos indica la redirección. Si accedemos desde nuestra máquina local, veremos el servidor web en el puerto 8000 del anfitrión.

3.3.8. Docker Hub

Aquí vamos a hablar un poco de la plataforma Docker Hub, que sirve como repositorio de imágenes oficiales y de terceros.

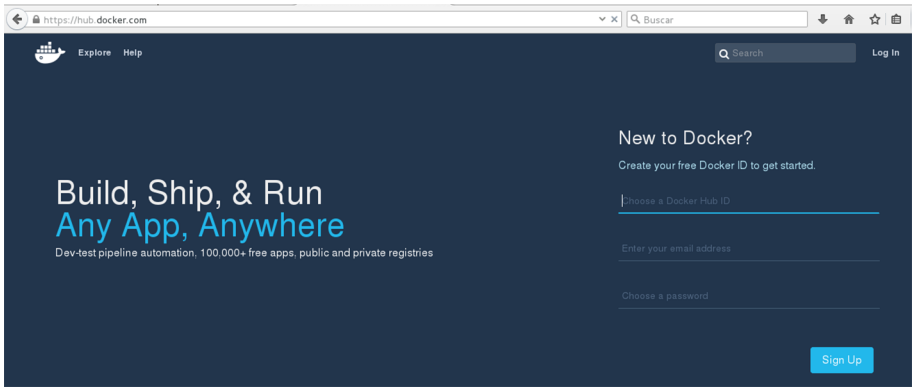
Las características de Docker Hub son:

- Repositorios de imágenes: encuentra, administra, sube y descarga imágenes oficiales y de la comunidad.
- Imágenes automáticas: crea nuevas imágenes cuando haces un cambio en la fuente de Github o BitBucket.
- Webhooks: crea automáticamente imágenes al hacer un *push* a un repositorio.

En Docker Hub podemos tener nuestro propio repositorio de imágenes, público o privado.

Ya hemos visto cómo usar los repositorios desde la línea de comandos, ahora lo vemos desde el sitio web.

Figura 28. Formulario de inscripción al Hub de Docker



Fuente: Jordi Guijarro.

Creamos una cuenta y al acceder vemos que podemos crear nuestro propio repositorio, organización o buscar en los repositorios.

Al explorar vemos primero los repositorios oficiales. Si entramos en el repositorio oficial de nginx por ejemplo, vemos las versiones disponibles para descargar y el comando.

Figura 29. Detalle de las imágenes de nginx para Docker

Fuente: Jordi Guijarro.

3.3.9. *Links*

Para que dos contenedores colaboren entre ellos, podemos abrir puertos y que se comuniquen por ahí, pero Docker nos ofrece la posibilidad de crear *links* entre contenedores.

Primero creamos un contenedor:

```
root@cloud-jguijarro:~# docker run -d --name links01 apache/ubuntu:v1
/usr/sbin/apache2ctl -D FOREGROUND
bcaf034f661cfa05f4f520f42914620b06b8137a9c62a9117997cc69bd952a34
```

No hemos mapeado ningún puerto, el servidor no es accesible desde fuera.

Ahora creamos otro con la opción --link para que Docker cree un túnel entre los dos contenedores:

```
root@cloud-jguijarro:~# docker run -d --name links02 --link links01
apache/ubuntu:v1 /usr/sbin/apache2ctl -D FOREGROUND
848a28b767733f7d791a4e3de0ae88423d1f71213c2dde4eeb6f9feec0fc8d4c
```

Comprobamos los contenedores activos con el comando «docker ps». Este túnel es unidireccional y solo podremos acceder a él desde el que se ha ejecutado con --link al otro:

```
root@cloud-jguijarro:~# docker exec links02 env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=848a28b76773
LINKS01_NAME=/links02/links01
```

```
HOME=/root
```

Vemos que aparecen variables del contenedor links01. Podemos modificar desde links02 las variables de entorno del otro contenedor. Docker internamente gestiona las Ips de los contenedores, añadiéndolas al fichero /etc/hosts de los contenedores.

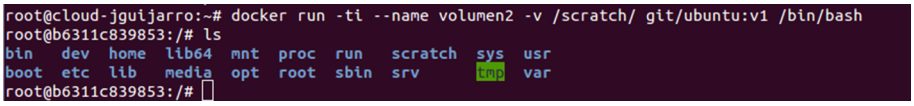
```
root@cloud-jguijarro:~# docker exec links02 cat /etc/hosts
127.0.0.1 localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.6 links01 bcaf034f661c
172.17.0.7 848a28b76773
```

3.3.10. Volúmenes

Podemos montar volúmenes de datos en nuestros contenedores. Con el *flag* -v podemos montar un directorio dentro de nuestro contenedor. Así, podemos incorporar datos que estarían disponibles para nuestro contenedor. Además, seguirá apareciendo después de un reinicio del contenedor, serán persistentes. También podemos compartir volúmenes de datos entre contenedores. Si dichos volúmenes están anclados al sistema operativo anfitrión, no serán eliminados por Docker cuando desaparezca el contenedor.

Vamos a crear un contenedor al que le pasemos un volumen.

Figura 30



Fuente: Jordi Guijarro.

Cuando listamos vemos en la carpeta que encontramos montado en nuestro contenedor el directorio «scratch». Esto es muy útil para pasarle ficheros a los contenedores y que los datos que nos interesen en ellos sean persistentes.

También podemos crear un contenedor que monte un volumen de otro con la opción –volume-from:

```
root@cloud-jguijarro:/volumen01# docker run -d -p 8085:80 --name scratch
--volumes-from ejemplo_volumen git/ubuntu:v1 /sbin/apache2 -D
FOREGROUND
cf51f79ccb03f9042362d9ff962aba3ebd4c5a35ace3514600d42a37a5976036
```

3.3.11. Variables de entorno

Podemos modificar las variables de entorno de un contenedor con el *flag* -e (--env).

También podemos pasarlas desde un fichero externo con ---env-file.

Y podemos ver las variables con:

```
root@cloud-jguijarro:/volumen01# docker exec a60 env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=a6066a7a2383
HOME=/root
```

3.3.12. Configuración de red

Por defecto, los contenedores tienen las conexiones de redes habilitadas. Podemos deshabilitarlas pasando la opción --net none.

Con el comando docker run podemos especificar tres configuraciones referentes a la red:

a) -dns: para configurar un servidor dns en el contenedor:

```
# docker run -i -t -dns="8.8.8.8" ubuntu /bin/bash
```

b) -net: crear una red y conectar el servidor a ella:

```
# docker network create -d overlay mi-red
# docker run -net=mi-red -i -t -d ubuntu
```

c) -add-host: agregando una entrada en el fichero /etc/hosts del contenedor:

```
# docker run -i -t -add-host apache:10.0.0.10 ubuntu /bin/bash
```

3.3.13. Eliminar contenedores

Para eliminar un contenedor podemos hacerlo por el nombre o por el ID. Realmente solo necesitamos los tres primeros dígitos del ID. Para poder eliminarlo debe estar parado. Si no lo estuviera, tendríamos que pararlo con «stop».

Utilizamos el comando rm. Si queremos ahorrarnos el paso de pararlo, usamos -f.

3.3.14. Dockerfile

Un Dockerfile es un archivo legible por el demonio Docker, que contiene una serie de instrucciones para automatizar el proceso de creación de un contenedor.

Construir el contenedor

El comando docker build irá siguiendo las instrucciones del Dockerfile y armando la imagen. El Dockerfile puede encontrarse en el directorio en el que estemos o en un repositorio.

El demonio de Docker es el que se encarga de construir la imagen siguiendo las instrucciones línea por línea y va lanzando los resultados por pantalla. Cada vez que ejecuta una nueva instrucción se hace en una nueva imagen, son imágenes intermedias, hasta que muestra el ID de la imagen resultante de ejecutar todas las instrucciones. El *daemon* irá haciendo una limpieza automática de las imágenes intermedias.

Estas imágenes intermedias se guardan en la caché de Docker. Y ¿para qué sirve que Docker cree una caché de imágenes intermedias? Pues para que, si por alguna razón la creación de la imagen falla (por un comando erróneo por ejemplo), cuando lo corregimos y volvemos a construir la imagen a partir del dockerfile, el demonio no tiene que iniciar todo el proceso, sino que usará las imágenes intermedias y continuará en el punto donde falló.

Dockerfile comandos

1) FROM

Indicamos una imagen base para construir el contenedor y opcionalmente un *tag* (si no la indicamos, docker asumirá «latest» por defecto, es decir, buscará la última versión).

Lo que hace docker al leer esto es buscar en la máquina local una imagen que se llama; si no la encuentra, la descargará de los repositorios.

Sintaxis:

```
FROM <imagen>
FROM <imagen>:<tag>
```

2) MAINTAINER

Es información del creador y mantenedor de la imagen, usuario, correo, etc.

Sintaxis:

```
MAINTAINER <nombre> <correo> <cualquier_info>
```

3) RUN

Ejecuta directamente comandos dentro del contenedor y luego aplica/persiste los cambios creando una nueva capa encima de la anterior con los cambios producidos de la ejecución del comando, y se hace un commit de los resultados. Posteriormente, sigue con la siguiente instrucción.

Sintaxis:

```
RUN <comando> -> modo shell, /bin/sh -c RUN ["ejecutable","parámetro1","parámetro2"] -> modo
ejecución, que permite correr comandos en imágenes base que no tengan /bin/sh o hacer uso
de otra shell.
```

4) ENV

Establece variables de entorno del contenedor. Dichas variables se pasarán a todas las instrucciones RUN que se ejecuten posteriores a la declaración de las variables. Podemos especificar varias variables de entorno en una sola instrucción ENV.

Sintaxis:

```
ENV <key> <value> <key> <value>
ENV <key>=<value> <key>=<value>
```

Si queremos sustituir una variable aunque esté definida en el Dockerfile, al ejecutar un contenedor podemos especificarla y tomará dicho valor, tenga el que tenga en el Dockerfile.

```
docker run -env <key>=<valor>
```

También podemos pasar variables de entorno con el comando «docker run» utilizando el parámetro -e para especificar que dichas variables solo se utilizarán en tiempo de ejecución.

Podemos usar estas variables en otras instrucciones llamándolas con $nombre_var o $ {nombre_var}. Por ejemplo:

```
ENV DESTINO_DIR /opt/app
WORKDIR $DESTINO_DIR
```

5) ADD

Esta instrucción copia los archivos o directorios de una ubicación especificada en <fuente> y los agrega al sistema de archivos del contenedor en la ruta especificada en <destino>.

En fuente podemos poner una URL, Docker se encargará de descargar el archivo y copiarlo en el destino. Si el archivo origen está comprimido, lo descomprime en el destino como si usáramos «tar –x».

Sintaxis:

```
ADD <fuente>..<destino>
ADD ["fuente",..."destino"]
```

El parámetro <fuente> acepta caracteres comodín tipo ?,*, etc. Una de las cosas que debemos tener en cuenta (entre otras → https://docs.docker.com/engine/reference/builder/#add) es que el <origen> debe estar donde esté el Dockerfile, no se pueden añadir archivos desde fuera del directorio de construcción.

Si el destino no existe, Docker creará la ruta completa incluyendo cualquier subdirectorio. Los nuevos archivos y directorios se crearán con los permisos 0755 y un UID y GID de 0.

También hemos de tener en cuenta que si los archivos o directorios agregados por una instrucción ADD cambian, entonces se invalida la caché para las siguientes instrucciones del Dockerfile.

6) COPY

Es igual que ADD, solo que NO admite URL remotas y archivos comprimidos como lo hace ADD.

7) WORKDIR

Permite especificar en qué directorio se va a ejecutar una instrucción RUN, CMD o ENTRYPOINT.

Puede ser usada varias veces dentro de un Dockerfile. Si se da una ruta relativa, esta será la ruta relativa de la instrucción WORKDIR anterior.

Podemos usar variables de entorno previamente configuradas, por ejemplo:

```
ENV rutadir /ruta
WORKDIR $rutadir
```

8) USER

Sirve para configurar el nombre de usuario que utilizar cuando se lanza un contenedor y para la ejecución de cualquier instrucción RUN, CMD o ENTRYPOINT posteriores.

9) VOLUME

Crea un punto de montaje con un nombre especificado que permite compartir dicho punto de montaje con otros contenedores o con la máquina anfitriona. Es un directorio dentro de uno o más contenedores que no utiliza el sistema de archivos del contenedor, aunque se integra en él para proporcionar varias funcionalidades útiles con el fin de que los datos sean persistentes y se puedan compartir con facilidad.

Esto se hace para que cuando usemos el contenedor podamos tener acceso externo a un determinado directorio del contenedor.

Las características de estos volúmenes son:

- Los volúmenes pueden ser compartidos y reutilizados entre los contenedores.
- Un contenedor no tiene que estar en ejecución para compartir sus volúmenes.
- Los cambios en un volumen se hacen directamente.
- Los cambios en un volumen no se incluirán al actualizar una imagen.
- Los volúmenes persisten incluso cuando dejan de usarlos los contenedores. Esto permite añadir datos, BBDD o cualquier otro contenido sin comprometer la imagen. El valor puede ser pasado en formato JSON o como un argumento, y se pueden especificar varios volúmenes.

```
VOLUME ["/var/tmp"]
VOLUME /var/tmp
```

10) LABEL

LABEL añade metadatos a una imagen Docker. Se escribe en el formato etiqueta=“valor”. Se pueden añadir varios metadatos separados por un espacio en blanco.

```
LABEL version="1.0"
LABEL localizacion="Barbate" tipo="BBDD"
```

Podemos inspeccionar las etiquetas en una imagen usando el comando docker inspect. $ docker inspect <nombre_imagen>/<tag>.

11) STOPSIGNAL

Le indica al sistema una señal que será enviada al contenedor para salir. Puede ser un número válido permitido por el *kernel* (por ejemplo, 9) o un nombre de señal en el formato SIGNAME (por ejemplo, SIGKILL).

12) ARG

Define una variable que podemos pasar cuando estemos construyendo la imagen con el comando docker build, usando el *flag* --build-arg <varname>=<value>. Si especificamos un argumento en la construcción que no está definido en el Dockerfile, nos dará un error.

El autor del Dockerfile puede definir una o más variables. Y también puede definir un valor por defecto para una variable, que se usará si en la construcción no se especifica otro.

```
ARG user1
ARG user1=someuser ARG user2
```

Se deben usar estas variables de la siguiente manera:

```
docker build -build-arg <variable>=<valor> ....
```

Docker tiene un conjunto de variables predefinidas que pueden usarse en la construcción de la imagen sin que estén declaradas en el Dockerfile:

```
HTTP_PROXY http_proxy
HTTPS_PROXY https_proxy
FTP_PROXY ftp_proxy
NO_PROXY no_proxy
```

13) ONBUILD

Añade *triggers* a las imágenes. Un disparador se utiliza cuando se usa una imagen como base de otra imagen. El disparador inserta una nueva instrucción en el proceso de construcción como si se especificara inmediatamente después de la instrucción FROM.

Por ejemplo, tenemos un Dockerfile con la instrucción ONBUILD, y creamos una imagen a partir de este Dockerfile, por ejemplo, IMAGEN_padre. Si escribimos un nuevo Dockerfile, y la sentencia FROM apunta a IMAGEN_PADRE,

cuando construyamos una imagen a partir de este Dockerfile, IMAGEN_HIJO, veremos en la creación que se ejecuta el ONBUILD que teníamos en el primer Dockerfile.

Los disparadores ONBUILD se ejecutan en el orden especificado en la imagen padre y solo se heredan una vez; si construimos otra imagen a partir de la IMAGEN_HIJO, los disparadores no serán ejecutados en la construcción de la IMAGEN_NIETO.

Hay algunas instrucciones que no se pueden utilizar en ONBUILD, como son FROM, MAINTAINER y ONBUILD, para evitar recursividad.

Un ejemplo de uso:

```
FROM Ubuntu:14.04
MAINTAINER mcgomez
RUN apt-get update && apt-get install -y apache2
ONBUILD ADD ./var/www/
EXPOSE 80
CMD ["D","FOREGROUND"]
```

14) EXPOSE

Se utiliza para asociar puertos, permitiéndonos exponer un contenedor al exterior (internet, *host*, etc.). Esta instrucción le especifica a Docker que el contenedor escucha en los puertos especificados. Pero EXPOSE no hace que los puertos puedan ser accedidos desde el *host*, para ello debemos mapear los puertos usando la opción -p en docker run.

Por ejemplo:

```
EXPOSE 80 443
docker run centos:centos7 -p 8080:80
```

15) CMD

Esta instrucción es similar al comando RUN pero con la diferencia de que se ejecuta cuando instanciamos o arrancamos el contenedor, no en la construcción de la imagen.

Solo puede existir una única instrucción CMD por cada Dockerfile y puede ser útil para ejecutar servicios que ya estén instalados o para correr archivos ejecutables especificando su ubicación.

16) ENTRYPOINT

Cualquier argumento que pasemos en la línea de comandos mediante docker run será anexado después de todos los elementos especificados mediante la instrucción ENTRYPOINT, y anulará cualquier elemento especificado con CMD. Esto permite pasar cualquier argumento al punto de entrada.

Sintaxis:

```
ENTRYPOINT ["ejecutable","parámetro1","parámetro2"] -> forma de ejecución ENTRYPOINT comando
parámetro1 parámetro 2 -> forma shell
```

ENTRYPOINT nos permite indicar qué comando se va a ejecutar al iniciar el contenedor, pero en este caso el usuario no puede indicar otro comando al iniciar el contenedor. Si usamos esta opción, es porque no esperamos que el usuario ejecute otro comando que el especificado.

17) ARCHIVO DOCKERIGNORE

Es recomendable colocar cada Dockerfile en un directorio limpio, en el que solo agreguemos los ficheros que sean necesarios. Pero es posible que tengamos algún archivo en dicho directorio que cumpla alguna función pero que no queremos que sea agregado a la imagen. Para esto usamos .dockerignore, para que docker build excluya esos archivos durante la creación de la imagen.

Un ejemplo de .dockerignore:

```
*/prueba*
*/*/prueba
prueba?
```

Ejemplo de Dockerfile

Vamos a lanzar un contenedor haciendo uso del siguiente Dockerfile, que hemos colocado en una carpeta llamada /wordpress.

```
FROM debian #FROM nos indica la imagen base a partir de la cual crearemos la imagen con
"wordpress" que construirá el Dockerfile.
MAINTAINER UOC <info@uoc.edu>
ENV HOME /root #ENV HOME: Establecerá nuestro directorio "HOME" donde relizaremos los comandos "RUN".
RUN apt-get update
RUN apt-get install -y nano wget curl unzip lynx apache2 php5 libapache2- mod-php5 php5-mysql
RUN echo "mysql-server mysql-server/root_password password root" | debconf-set-selections
RUN echo "mysql-server mysql-server/root_password_again password root" | debconf-set-selections
RUN apt-get install -y mysql-server
ADD https://es.wordpress.org/wordpress-4.2.2-es_ES.zip /var/www/wordpress.zip #ADD nos permite
añadir un archivo al contenedor, en este caso nos estamos bajando Wordpress
ENV HOME /var/www/html/
RUN rm /var/www/html/index.html
```

```
RUN unzip /var/www/wordpress.zip -d /var/www/
RUN cp -r /var/www/wordpress/* /var/www/html/
RUN chown -R www-data:www-data /var/www/html/
RUN rm /var/www/wordpress.zip
ADD /script.sh /script.sh
RUN ./script.sh
#ADD nos añadirá en este caso la configuración de la base de datos que una vez realizada el
despliegue tendremos que poner para su utilización.
EXPOSE 80 #EXPOSE indica los puertos TCP/IP por los que se pueden acceder a los servicios del
contenedor 80 "HTTP".
CMD ["/bin/bash"] #CMD nos establece el comando del proceso de inicio que se usará.
ENTRYPOINT ["./script.sh"]
```

También colocamos en la misma carpeta el *script* donde vamos a establecer una comunicación con la base de datos y nos va a crear una base de datos "wordpress", usuario "wordpress" y contraseña "wordpress":

```
#!/bin/bash
#Iniciamos el servicio mysql
/etc/init.d/mysql start
#Guardamos en variables los datos de la base de datos
DB_ROOT="root" DB_ROOT_PASS="root" DB_NAME="wordpress" DB_USER="wordpress" DB_PASS="wordpress"
#Nos conectamos a la BBDD como root y creamos el usuario sql
mysql -u ${DB_ROOT} -p${DB_ROOT_PASS} -e "CREATE USER '${DB_USER}';"
#Creamos la base de datos
mysql -u ${DB_ROOT} -p${DB_ROOT_PASS} -e "CREATE DATABASE ${DB_NAME};"
#Le damos permisos al usuario sobre la base de datos y le ponemos contraseña
mysql -u ${DB_ROOT} -p${DB_ROOT_PASS} -e "GRANT ALL ON ${DB_NAME}.* TO $ {DB_USER} $

#Reiniciamos los servicios
/etc/init.d/apache2 start
/bin/bash
```

Nos situamos en la carpeta donde tenemos el Dockerfile y creamos la imagen a partir de nuestro Dockerfile con el siguiente comando:

```
root@cloud-jguijarro:~# docker build -f dockerfile -t wordpress.
```

Ya tenemos creada nuestra imagen. Podemos listarla como una más:

```
root@cloud-jguijarro:~#docker images
```

El siguiente paso es correr la imagen en nuestro puerto 80, para lo que usamos el siguiente comando:

```
root@cloud-jguijarro:~#docker run -d -p80:80 -i myblog
```

Limitación de recursos

Podemos limitar los recursos usados por los contenedores.

Algunos ejemplos de creación de contenedores con limitación de recursos serían estos:

```
root@cloud-jguijarro:/# docker run -i -t -m 500m ubuntu /bin/bash root@fbc27c5774aa:/#
```

Hemos creado un contenedor con un límite de memoria de 500 megas.

Inicio automático

Para iniciar un contenedor y hacerlo disponible desde el inicio del sistema anfitrión, usamos --restart, que admite tres valores:

- -no: valor predeterminado, no estará disponible al reiniciar el sistema anfitrión.
- -on-failure[max-retries]: reinicia si ha ocurrido un fallo y podemos indicarle los intentos.
- -always: reinicia el contenedor siempre.

Ejemplo:

```
root@cloud-jguijarro:/# docker run -i -t --restart=always ubuntu /bin/bash
```

4. Infraestructura PaaS pública: Cloud9, Heroku

Nota

En este apartado hablaremos de dos de las soluciones PaaS más populares para desarrolladores basados en el *cloud*: Cloud9 y Heroku.

Hasta el momento hemos visto entornos de aplicaciones completamente administrados en nuestros servidores, pero vivimos en una época en la que cada vez aparecen más herramientas de desarrollo en la nube y dotan a los desarrolladores de la capacidad de utilizar infraestructuras PaaS (*platform-as-a-service*), donde el entorno viene dado y solo tendremos que centrarnos en el desarrollo de la aplicación.

La filosofía del desarrollo basado en infraestructuras PaaS reduce la complejidad del modelo, ya que la gestión y el mantenimiento, así como la escalabilidad de la infraestructura serán delegadas a terceros. En ningún caso esto querrá decir que como desarrolladores no hayamos de tener en cuenta la infraestructura, pero permitirá que centremos nuestros esfuerzos en optimizar nuestras aplicaciones olvidándonos de la administración a más bajo nivel.

En este momento ya seríamos capaces de nombrar las singularidades y ventajas que un sistema PaaS basado en VPS puede ofrecer:

- **Configuraciones asistidas del servidor:** en pocos pasos podemos disponer de servidores totalmente configurados para ejecutar los lenguajes de programación más comunes.

- **Proyectos en la nube:** podemos acceder al desarrollo de nuestros códigos por internet mediante una URL real, superando las limitaciones del desarrollo en local.

- **Trabajos colaborativos:** los entornos de trabajo son accesibles desde internet, lo que permite a grupos de trabajo poder acceder al servidor para realizar múltiples tareas de forma simultánea.

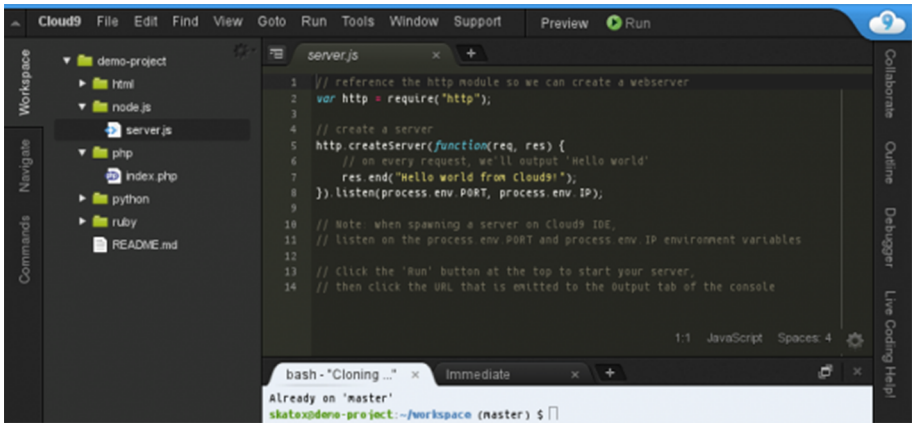
- **Posibilidad de deslocalización de los equipos de trabajo:** cualquier ordenador con acceso a internet se convierte en una estación de trabajo; algunas de las herramientas más famosas ya incorporan IDE completos, con lo que no hace falta instalar ningún otro software para trabajar en el código del servidor.

4.1. Cloud9

Cloud9 se presenta como una aplicación en nuestro navegador en la que dispondremos de un IDE completo donde además de poder desarrollar nuestras aplicaciones online podremos administrar nuestros entornos de trabajo.

Dentro de los planes de precios encontraremos una opción gratuita en la que dispondremos de un entorno de trabajo con los recursos suficientes donde probar nuestras aplicaciones.

Figura 31. Detalle de la interfaz web de Cloud9

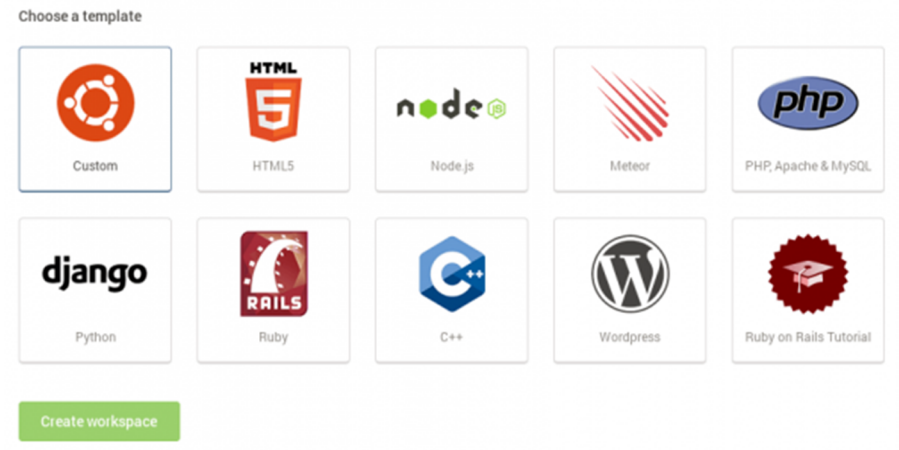


Fuente: Joan Caparrós.

La interfaz web nos permitirá crear y editar en tiempo real nuestras aplicaciones, al tiempo que nos dará acceso a un terminal que nos permitirá instalar dependencias y servicios (servidor web, *framework*, base de datos, etc.) como si de una VPS se tratara.

El uso de Cloud9 es muy sencillo, la configuración de nuestro entorno de trabajo (*workspace*) es muy visual y dispondremos de configuraciones ya establecidas para disponer de entornos HTML5, NodeJS, Meteor, LAMP, Django, Rails, C++, entre otros. Después del paso inicial de configuración del entorno, ya solo se deberá incluir un repositorio de nuestro sistema de control de versiones favorito y, tras ejecutar los servicios necesarios, ya podremos acceder a la aplicación desde el navegador.

Figura 32. Detalle de las aplicaciones disponibles en Cloud9



Fuente: Joan Caparrós.

4.2. Heroku

Heroku es otra solución PaaS en donde albergar nuestras aplicaciones en la nube. Para ello, la plataforma dispone de una opción gratuita con la que dispondremos de los recursos mínimos para poder ejecutar el código desarrollado.

La plataforma dispone de configuraciones básicas para dar soporte al despliegue de aplicaciones desarrolladas en NodeJS, Rails, Java, PHP, Python, Go, Scala y Clojure. Una vez definido el lenguaje e introducido sus dependencias, el sistema asignará un servidor ligero llamado dynos, donde ejecutará el código indicado por el usuario.

Para configuraciones avanzadas se deberá instalar Heroku Toobelt, la aplicación para poder manejar Heroku desde consola, disponible para Windows, Mac OS y Debian/Ubuntu.

Los despliegues en Heroku están directamente ligados al código almacenado en el sistema de control de versiones, Heroku Git, Github o Dropbox. Una vez configurada la plataforma, las actualizaciones irán ligadas a las modificaciones del código subido al repositorio remoto vinculado al proyecto; en cada *push* el sistema reiniciará nuestra aplicación incorporando los últimos cambios aplicados.

La escalabilidad del proyecto dependerá de los dynos implicados en el proyecto, pero es posible ampliar los recursos destinados mediante la contratación de horas de uso de dynos extras.

En el caso de necesitar mejorar los servicios que acompañan a nuestra aplicación –como gestión de *logs*, *testing*, tareas programadas, entre otros–, Heroku dispone de *addons* (podéis consultar la lista completa de *addons* disponibles desde la web https://addons.heroku.com/).

A continuación mostraremos los pasos necesarios para el despliegue de una aplicación dentro de la plataforma Heroku:

- Creamos una cuenta en Heroku: https://signup.heroku.com/.
- Instalamos Heroku Toobelt: https://toolbelt.heroku.com/.
- Iniciamos sesión en Heroku mediante el terminal: heroku login.
- Creamos una aplicación, mediante la herramienta web o ejecutando las siguientes instrucciones:
  - heroku create.
  - heroku apps:rename nombreApp.

  - heroku open.

- Procedemos a la sincronización del repositorio git vinculado con los últimos cambios que aplicar mediante el comando *push*: git push origin master.

- La aplicación quedará desplegada en la dirección https://nombreApp.herokuapp.com.

Bibliografía

1. Bibliografía utilizada (libros, revistas, capítulos de libros u otras fuentes de interés)

Además de los enlaces proveídos dentro de los propios materiales como recursos adicionales, añadimos las siguientes publicaciones:

Effective DevOps: <http://shop.oreilly.com/product/0636920039846.do>.

The DevOps 2.0 Toolkit: <https://leanpub.com/the-devops-2-toolkit>.

The Phoenix Project: <https://www.amazon.com/Phoenix-Project-DevOps-Helping-Business/dp/0988262592>.

The DevOps Handbook: <https://www.amazon.com/gp/product/1942788002/ref=pd_sim_14_2?ie=UTF8&psc=1&refRID=YW60HZBXV9XNQAJC9CPR>.

The Official (ISC)2 Guide to the CCSP CBK 2nd Edition: <https://www.amazon.com/gp/product/1119276721/ref=pd_sbs_14_3/157-2087627-3975059?ie=UTF8&psc=1&refRID=5MR8CP23FF8YSNSFWZZR>.

Terraform: Up and Running:<https://www.amazon.com/Terraform-Running-Writing-Infrastructure-Code/dp/1491977086>

2. Referencias

[1] **James P. Womack; Daniel T. Jones; Daniel Roos** (1990). «The Machine That Changed the World».

[2] **Ward Cunningham** (1992, 26 de marzo). «The WyCash Portfolio Management System».

[3] **Kent Beck y otros** (2001; febrero). *Manifesto for Agile Software Development* [en línea]. <http://agilemanifesto.org>.

[4] **Brian Marick** (2003, 22 de agosto). *Agile testing directions: tests and examples* [en línea]. <http://www.exampler.com/old-blog/2003/08/22/>.