

Capítulo 6: Despliegue

Víctor Iranzo

27 de mayo de 2018

1. Integración continua

1.1. ¿Qué es la integración continua?

La integración continua (cuyas siglas en inglés son CI) tienen el objetivo de mantener a todos los miembros de un equipo sincronizados con los cambios del resto mediante integraciones de las modificaciones hechas con el código fuente. El proceso de integración del cambio puede pasar fases donde se verifica su correcto funcionamiento, como la ejecución de pruebas o la compilación.

Durante este proceso se crean artefactos sobre los que se ejecutarán las validaciones. Los artefactos contruidos deben ser lo más parecidos a los que más tarde se despliegan en la versión de producción. Si se reutilizan estos artefactos, nos aseguraremos de que el artefacto sobre al que se han hecho las pruebas coincide con el que se publica.

Entre los beneficios de la integración continua encontramos: una más rápida retroalimentación sobre la calidad de un cambio, la capacidad de automatización la generación de artefactos o la trazabilidad que existe entre el código y un artefacto (como el código está bajo control de versiones, en cualquier momento se puede volver a construir un artefacto de una versión concreta).

1.2. ¿Estás aplicando realmente CI?

Hay muchas organizaciones que presumen de hacer integración continua cuando realmente no la están haciendo. Jez Humble establece tres preguntas para saberlo:

- ¿Realizas e integras tus cambios diariamente? Comprueba que tus cambios se integran con los del resto o harás que más adelante sea más difícil. Si empleas ramas para implementar una funcionalidad, haz que su tiempo de vida sea breve e integra cuanto antes en la rama principal.
- ¿Tienes una batería de pruebas para validar tus cambios? Si los cambios no se validan, estamos simplemente integrando código que compila pero que tal vez no cumple con su comportamiento esperado.
- Si la compilación falla, ¿arreglarla es la primera prioridad del equipo? Si la compilación no se arregla cuanto antes, el número de cambios a integrar aumentará progresivamente y luego será más costosa su integración.

1.3. Relación entre microservicios y CI

El objetivo principal es poder desplegar un servicio de forma independiente al resto. Teniendo esto en cuenta, ¿cómo podemos relacionar las compilaciones de la integración continua y el código fuente con cada servicio?

La opción más simple es establecer una sola compilación y un único repositorio para todo el código. Un simple cambio lanzaría la compilación de toda la solución, pero esto puede resultar útil si estoy modificando más de un servicio. Esta solución solo es aconsejable en casos muy específicos como proyectos que acaban de comenzar y con un único equipo. Por lo general, no es aconsejable porque despliega artefactos que no son necesarios, aumentando el coste y tiempo de cada compilación.

Otra aproximación es tener un único repositorio para todos los servicios y diferentes compilaciones para cada servicio. El problema de esta aproximación es que puede conducirnos a la mala práctica de hacer cambios que modifiquen código de diferentes servicios, que se puede traducir en aumentar el acoplamiento entre ellos.

La solución más idónea pasa por tener una compilación y un repositorio diferente para cada servicio. De esta forma se evita juntar en uno cambios que afecten a diferentes servicios y se genera un único artefacto por compilación (con la ejecución de sus respectivas pruebas). También hay una mayor correspondencia entre los servicios y los equipos encargados de cada uno porque cada equipo realiza cambios en un único repositorio.

1.4. Entrega continua

Se pueden obtener mayores beneficios de la integración continua a través de las compilaciones por fases o build pipelines. En este tipo de compilaciones se separa por pasos el proceso: la ejecución de las pruebas unitarias, la de las pruebas de rendimiento... El beneficio de hacer esto es que se obtiene una retroalimentación más rápida si se ejecutan de forma separada acciones rápidas de otras más pesadas.

Desde el punto de vista de la pipeline se puede observar como el artefacto generado para la compilación pasa de un paso al siguiente, lo que se traduce en que el código está más cerca de poderse llevar a producción. Los pasos de la pipeline pueden ser manuales, como las pruebas de aceptación de un usuario, o estar automatizados, como la ejecución de tests unitarios. Si nos situamos en el contexto de los microservicios, queremos tener una pipeline para cada servicio con el objetivo de poder desplegarlos de forma independiente.

La entrega continua se basa en dos ideas: la primera, poder obtener feedback de todos y cada uno de los cambios realizados a través de su paso por el pipeline, y la segunda, que es poder tratar a cada uno de los cambios como candidatos a salir a producción. Entre las ventajas de su uso destaca que se hace más tangible el proceso de calidad y publicación del software y que se reduce el tiempo entre versiones de producción porque se puede publicar una por cada cambio hecho.

2. Artefactos

2.1. Artefactos específicos de la plataforma

Algunas tecnología tienen artefactos propios como son los archivos JAR en Java. Desde el punto de vista de los microservicios, estos artefactos no son suficientes para levantar uno de ellos. Necesitamos instalar y configurar más software para el despliegue del servicio y algunos frameworks como Puppet o Chef pueden ayudar.

Como su nombre indica, estos artefactos son específicos de una pila tecnológica y esto puede hacer que el despliegue de un sistema basado en servicios sea más costoso si se emplean en cada uno tecnologías distintas. Para facilitarlo, podemos apoyarnos en las herramientas que hemos citado antes porque soportan diferentes tecnologías.

2.2. Artefactos del sistema operativo

Una forma de evitar los problemas asociados a artefactos específicos de la tecnología empleada es emplear los del sistema operativo. Por ejemplo, en Ubuntu se puede emplear un paquete DEB o en Windows un MSI. Usar estos paquetes nos puede ahorrar trabajo, que de otra forma se haría con frameworks como Puppet o Chef, porque se delega en el gesto de paquetes del SO.

El principal problema es si necesitamos desplegar nuestro sistema en diferentes sistemas operativos. Si esto pasa, haría falta un mayor esfuerzo para producir los paquetes en cada SO. Si no es el caso, esta aproximación puede simplificar mucho el despliegue y evitar complejos scripts para esta tarea.

2.3. Imágenes propias

Uno de los principales problemas en herramientas como Puppet, Chef o Ansible es el tiempo que tardan en ejecutarse scripts de configuración. Estas herramientas son capaces de detectar si el software que necesitan instalar está ya presente en la máquina donde corren.

Puede parecer razonable mantener una máquina durante mucho tiempo para reducir el tiempo de despliegue, ya que al hacerlo el software necesario no haría falta volverlo a instalar. Sin embargo, no es una buena práctica alargar la vida de estas máquinas porque sino los cambios en la configuración del despliegue no se aplicarían. En definitiva, estas herramientas pueden suponer un problema si necesitamos dar retroalimentación rápidamente sobre un artefacto.

Para solucionarlo, una aproximación válida es, en lugar de hacer un script para instalar el software del que depende el servicio en cada despliegue, restaurar una imagen con todo el software necesario ya instalado. Todas las plataformas de virtualización permiten construir imágenes propias. El despliegue se simplifica a simplemente restaurar la imagen con el software necesario e instalar la versión del servicio a desplegar.

Aunque nos ahorremos tiempo durante el despliegue, construir una imagen puede llevarnos largo tiempo. Además, si depende de mucho software para funcionar, la imagen puede ser de un tamaño considerable. Por estos motivos, la tecnología que más se emplea es la basada en contenedores.

2.3.1. Packer

Una herramienta que puede resultar de utilidad a la hora de crear imágenes es Packer. Con Packer, empleando scripts de configuración de diferentes tecnologías (Chef, Ansible, Puppet...) se pueden crear imágenes para diferentes plataformas (VMWare, AWS, Vagrant, Linux...).

2.3.2. Imágenes como artefactos

Organizaciones como Netflix van más allá y construyen la imagen durante la compilación con el servicio ya instalado en ella. De esta forma, el despliegue es más rápido porque en la imagen ya está instalado el servicio.

3. Entornos y configuración de servicios

A medida que nuestro artefacto pasa de una etapa del pipeline a otra, va a desplegarse en diferentes entornos, por ejemplo, uno para pruebas unitarias, otro para pruebas de aceptación... Cada entorno se emplea con una finalidad concreta, pero la diferencia entre los entornos puede traer problemas si por ejemplo se producen errores con una configuración y no con otra.

A medida que el cambio avanza en la pipeline, queremos que el entorno donde se hacen las validaciones sea más parecido a uno de producción. Así podremos capturar lo antes posibles problemas asociados a diferencias entre entornos.

Algunos entornos pueden requerir de ficheros de configuración adicionales. Idealmente, estos no deberían existir para reducir las diferencias entre los entornos. Para añadir esta configuración adicional al artefacto podemos seguir dos aproximaciones: construir un artefacto diferente para cada entorno o construir un único artefacto y gestionar la configuración de forma separada.

En cuanto a la primera, rompe con uno de los conceptos clave de la entrega continua ya que no todos los artefactos construidos podrían ser candidatos para ser desplegados. Además, no podríamos saber si las validaciones hechas sobre uno de los artefactos se cumplen también en el que luego saliera a producción.

La segunda aproximación reduce las diferencias entre los entornos donde se instala el artefacto, por lo que es una mejor opción. En caso de que la

configuración sea muy extensa, en lugar de almacenar la configuración a través de ficheros se puede emplear un servicio especial para proveerla.

3.1. Definición del entorno

3.2. Interfaz para el despliegue

4. Alojamiento de servicios

Una pregunta habitual es cuántos servicios se deben desplegar por máquina. Vamos a introducir el concepto de host como una unidad genérica de aislamiento, un sistema operativo donde se pueden instalar y ejecutar servicios. Si desplegamos directamente sobre máquinas, entonces un host será el equivalente a una de ellas. En cambio, si empleamos virtualización, cada máquina puede contener más de un host que a su vez pueden contener uno o más servicios.

4.1. Múltiples servicios por host

Desde el punto de vista de gestionar servicios, es una aproximación simple y poco costosa al emplear un menor número de máquinas. Además, puede facilitar el trabajo de los desarrolladores. Sin embargo, cuenta con algunas desventajas:

- La monitorización resulta más difícil: responder preguntas como la CPU que emplea cada servicio puede ser difícil porque todos se encuentran en la misma máquina.
- Posibles efectos secundarios: un servicio que atiende muchas peticiones puede acaparar recursos que son necesarios para el resto de microservicios allí alojados.
- Un único punto de fallo: el fallo del host donde se alojan los servicios puede afectar muy negativamente al sistema porque todos sus servicios dejan de estar disponibles.
- Despliegue más complejo: el despliegue de un nuevo servicio no debe afectar a los ya alojados. Cada uno de ellos puede depender de software distinto y no podemos controlar completamente los efectos de su instalación. Además, es importante mantener la idea de que cada servicio ha de desplegarse de forma independiente.

- Reduce la autonomía de los equipos: si cada servicio lo gestiona un equipo, la gestión del host donde estos se alojan requiere la colaboración de todos los involucrados.
- No se pueden emplear algunos artefactos: no se pueden emplear artefactos como imágenes para el despliegue de cada servicio porque distintos servicios se ejecutan en el mismo sistema operativo.
- Necesidades distintas se tratan de la misma manera: en un mismo host, necesidades como la escalabilidad han de ser tratadas igual para todos los servicios porque todos ellos comparten una única configuración del host.

4.2. Contenedores de aplicaciones

Es un caso particular del modelo de múltiples servicios por host pero con ciertos beneficios para la gestión de los servicios alojados como herramientas para la monitorización. Algunos ejemplos de este modelo son IIS en aplicaciones .NET o contenedores servlet en Java.

Entre las desventajas de emplearlos encontramos las siguientes:

- Limitan la tecnología que se puede emplear: en los ejemplos que hemos puesto, a .NET o Java.
- Ralentizan el tiempo de ejecución: esto puede afectar a la retroalimentación que el desarrollador tiene del servicio que implementa.
- El uso de recursos es más complejo: en un mismo proceso se ejecutan las diversas aplicaciones como hilos.

En definitiva, es mucho mejor modelo emplear contenedores independientes para cada microservicio y tecnologías como las que hemos citado ya están recorriendo ese camino con frameworks como Nancy o Jetty.

4.3. Un servicio por host

Con este modelo se remedian los problemas de monitorización, efectos secundarios entre servicios y único punto de fallo que caracterizaban al modelo de más de un servicio por host. Además, los servicios pueden escalar independientemente de otros y la seguridad se aplica más fácilmente a nivel de host según las necesidades de cada servicio.

Como consecuencia de esta aproximación tendremos que mantener más hosts, aumentando el coste de su gestión. Aún así, a la hora de razonar resulta más sencillo tener un único servicio por host.

4.4. Plataforma como servicio (PaaS)

La plataforma como servicio (PaaS) es un nivel de abstracción más alto que el de un host. Se basa en delegar el aprovisionamiento y ejecución de un artefacto a otra organización, que también se encarga de escalar el sistema según la carga de cada nodo. Heroku es una de estas plataformas.

Cuando estas soluciones no funcionan correctamente en un servicio desplegado, es difícil tomar el control para tratar de solucionarlo. Por eso, el artefacto a desplegar es muy sensible a cómo está implementado, y cuanto más se salga del estándar que la plataforma espera, más probabilidad hay de que no funcione correctamente.

5. Automatización del despliegue

6. Tecnologías para el despliegue

6.1. Virtualización tradicional

6.2. Vagrant

6.3. Contenedores Linux

6.4. Docker