unir
LA UNIVERSIDAD DE INTERNET

FORMACIÓN PROFESIONAL

UNIDAD FORMATIVA 1

Cultura DevOps

Introducción a DevOps

Índice

Introducción al DevOps

DevOps es una revolución en la manera de trabajar de equipos y organizaciones: no debe entenderse solo en el contexto técnico, sino también en el cultural, una corriente de cambio en la que prima la colaboración desde la gestión hasta desarrollo y operaciones.

Revisaremos lo que se entiende por DevOps en la actualidad, tanto a nivel técnico como cultural. Una introducción a la cultura Agile expondrá el objetivo de muchas organizaciones IT a día de hoy: el **control total** sobre los procesos de despliegue de nuevas funcionalidades, desde su concepción hasta su puesta en marcha.

A continuación, se hará una revisión de cada una de las unidades formativas de las que se compondrá el curso:

- Construcción de software.
- Virtualización, Docker y Cloud.
- Networking.
- Conceptos de administración de sistemas.
- Herramientas de gestión de ciclo de vida.
- Despliegues y monitorización.
- DevOps en las organizaciones.

Una vez expuestos los términos sobre los que vamos a trabajar, se establecerán las bases técnicas sobre las que se desarrollarán las diferentes prácticas, tanto a nivel individual como de equipo, durante el resto del curso.

Para finalizar la unidad, el alumno deberá responder correctamente las preguntas del test de autoevaluación.

Objetivos

En este tema se pretenden conseguir los siguientes objetivos:

- Comprender qué implica el cambio cultural de una organización para la adopción de la metodología DevOps.
- Conocer las herramientas que ayudan a los equipos DevOps a realizar su trabajo de forma exitosa.
- Analizar los factores que promueven la adopción de DevOps en las organizaciones.
- Cimentar la base tecnológica para la parte práctica.

Cultura DevOps

DevOps no puede entenderse únicamente como un conjunto de tecnologías y mejores prácticas, sino como una **revolución cultural**. Es una corriente de cambio dentro de las organizaciones en la que prima:

- Colaboración e interacción abierta entre los equipos de operaciones y de desarrollo.
- Derribar los silos, barreras artificiales creadas en las organizaciones que frenan la colaboración natural.
- Ciclos cortos de diseño, implementación y despliegue que permiten anticiparse a los problemas y adaptarse al cambio.

Por lo general, solemos utilizar el término **cambio cultural** para implantar procesos de mejora continua, para emplear metodologías ágiles y para adoptar el enfoque DevOps.

Esto obedece a la imposibilidad de adoptar estas transformaciones de forma superficial: estamos hablando de un cambio organizativo muy beneficioso a la vez que radical.

Si bien son las grandes organizaciones las que podrán beneficiarse de estos cambios de forma más evidente en términos cuantitativos, son las nuevas organizaciones las que tienen la posibilidad de abrazar la cultura DevOps y de la colaboración de una forma natural.

En la mayoría de startups, la cultura de la colaboración, la visibilidad en contraposición con el control o la automatización en oposición al trabajo manual, surge de forma natural por la **necesidad de optimizar los recursos.**

Otro caso típico donde podemos ver la afinidad de las nuevas organizaciones con la colaboración y la cultura DevOps es el **Cloud First**: organizaciones donde no se plantea una transición a la nube porque nunca ha existido otra opción. Algunas organizaciones tradicionales han aceptado (mediante la resignación) que son incapaces de innovar de la forma en la que le gustaría debido a su propia inercia, por lo que intentan aprender de estas nuevas organizaciones y utilizarlas como herramientas.

¿A qué llamamos DevOps?

DevOps es un término relativamente nuevo para describir lo que también ha sido llamado como 'administración de sistemas ágiles', y que se centra en el trabajo y la colaboración de los equipos de Desarrollo y Operaciones.

En la actualidad, las empresas se mueven a la velocidad de la nube y es por ello por lo que DevOps se ha convertido en un enfoque cada vez más extendido y popular para la entrega de software.

Los equipos de Desarrollo y Operaciones utilizan esta metodología para **construir, probar, implementar y monitorizar** las aplicaciones debido a que les permite actuar con velocidad, mantener altos índices de calidad y controlar los cambios con suma rapidez.

DevOps es esencial para cualquier empresa que aspire a ser ágil y que pretenda ser capaz de responder rápidamente a las demandas del mercado. Por lo tanto, DevOps es un **enfoque hacia la entrega de software** y promueve una colaboración más estrecha entre las líneas de negocio, desarrollo y operaciones, al tiempo que elimina las barreras entre las partes interesadas y los clientes.

DevOps como herramienta de cambio para las organizaciones

Para ir adentrándonos en la materia necesitamos comprender la necesidad y el valor de DevOps a nivel de negocio, así como los principios de DevOps.

En general, realizar un cambio en el *Business as Usual* siempre es difícil y, por lo general, requiere de una inversión. Así que, cada vez que una organización adopta cualquier nueva tecnología, metodología o enfoque, la adopción de esta tiene que ser impulsada por una **necesidad de negocio**.

En un entorno DevOps, los equipos son responsables de ofrecer nuevas características, pero también estabilidad, escalabilidad, fiabilidad y otro sinfín de características comunes a un software de calidad. Es por ello por lo que las responsabilidades se equilibran de manera más equitativa para garantizar que ambos equipos, desarrollo y operaciones, tengan visibilidad del rendimiento de la aplicación a través de todas las etapas del **ciclo de vida**.

¿Qué impulsa a las empresas hacia la adopción de DevOps?

La respuesta es muy sencilla y esta son los largos plazos de entrega del software hasta su paso a producción, derivados de los sistemas de trabajo tradicionales que:

- Impiden a las empresas brindar servicios de vanguardia.
- Impiden a los equipos IT el trabajar de forma ágil.
- Lastran la experiencia del cliente.

Para mantener el ritmo de las demandas del mercado, los equipos de IT deben construir, implementar, probar y lanzar software en ciclos cada vez más rápidos.

Debido a que DevOps mejora la forma en que una empresa entrega valor a sus clientes, proveedores y socios, podemos afirmar que es un **proceso esencial de negocio** y no solo una capacidad de IT.

Metodologías ágiles

Se suele llamar **metodologías ágiles**, en comparación con la ingeniería de software tradicional, al conjunto de técnicas de desarrollo dedicadas principalmente a los sistemas complejos y al desarrollo de productos con características dinámicas, no deterministas y no lineales, donde las estimaciones precisas, planes estables y predicciones son, a menudo, difíciles de conseguir en etapas tempranas.

Estas técnicas son una aplicación directa del manifiesto Agile, firmado por expertos del desarrollo de software hace veinte años:

- **Individuos e interacciones** sobre procesos y herramientas.
- **Software funcionando** sobre documentación extensiva.
- **Colaboración con el cliente** sobre negociación contractual.
- **Respuesta ante el cambio** sobre seguir un plan.

Muchas veces se hace referencia a los métodos ágiles como métodos de adaptación continua o métodos adaptativos. Estos se centran en aplicar rápidamente los cambios de rumbo en un proyecto, modificando estimaciones, requisitos, alcance, etc., con el objetivo de garantizar un resultado final satisfactorio.

Las **principales metodologías ágiles** son las siguientes:

- Adaptive Software Development (ASD).
- Crystal Clear.
- Feature Driven Development (FDD).
- Kanban.
- Extreme Programming (XP).
- Scrum.

Nos centraremos más adelante en definir Scrum, al ser una de las más empleadas en las organizaciones, aunque esbozaremos las características del resto pues, como ha debido quedar bien claro hasta ahora, **siempre debemos optar por aquello que resulte mejor para el equipo**.

Yendo un paso más allá, es posible que estas metodologías, su estructura o sus eventos, no encajen con la forma de trabajar del equipo. Aunque siempre debemos estar abiertos al cambio, es **muy importante** iterar con cuidado cualquier nueva iniciativa que deseemos implementar, monitorizar su impacto en el ambiente y velocidad del equipo y, si esas mediciones son satisfactorias, añadirlas al flujo de trabajo.

Contenido del curso

Este curso se ha estructurado de forma lineal y en él presentamos los fundamentos de tecnologías y conocimientos que, en opinión del autor, deben acompañar al practicante DevOps en su día a día.

Es importante mantener siempre una actitud abierta, pues lo que se busca es presentar fundamentos y bases, no verdades absolutas que no deban ser rebatidas o aceptadas sin más.

Construcción de software

En esta unidad repasamos los fundamentos de programación y construcción de software.

En nuestro día a día como DevOps, nos podemos encontrar en varios escenarios:

- Desarrollo de software de aplicativos/ operaciones.
- Mantenimiento de software creado por otros equipos.
- Review de software de otros equipos/compañeros.

Todos estos casos se pueden dar en cualquier momento de nuestra carrera, y el factor común en todos los casos es, sin duda, que se requieren **conocimientos de desarrollo y mantenimiento de software**.

Con el objetivo de que el desarrollo de retos y prácticas sea lo más uniforme posible y consistente, con los ejemplos que se verán a lo largo de las diferentes asignaturas, nos centraremos en los siguientes **lenguajes de programación** (entre paréntesis, versión mínima recomendada).

- Java 1.8.
- Python 3.
- JavaScript (ECMAScript 2017).

¿Qué significa la versión mínima?

La evolución a lo largo del tiempo de los lenguajes de programación es algo a lo que tenemos que prestar atención, siempre. En especial, cuando el cambio se produce en una versión *major,* lo que en versionado semántico se entiende como que no se garantiza la retrocompatibilidad: un ejemplo es el paso de Python 2 a 3.

Esta elección condiciona totalmente el abanico de tecnologías que se pueden o podrán emplear en el futuro: frameworks web, tecnologías de testing, herramientas de gestión de ciclo de vida…

Esto no quiere decir que no se deban conocer y manejar otros lenguajes de programación: esto no trae más que ventajas y, en ocasiones, habrá elecciones que resulten idóneas para ciertas operaciones.

Ejemplos de **lenguajes con aplicaciones muy diversas** y gran penetración en la industria:

- Golang.
- Scala.
- Kotlin.
- Groovy.
- Elixir.
- Clojure.
- PHP.
- Ruby.
- Rust.
- R.

Testing

La segunda parte de esta lección estará centrada en cómo dotar al código de validaciones que garanticen que hace lo que tiene que hacer y que lo seguirá haciendo a medida que evolucione.

No solo esto, sino que el uso de tests en nuestro código y aplicación, en las condiciones adecuadas (rápidos, consistentes, fiables), es lo que posibilita una de las características más deseables de toda la cultura DevOps: l**a velocidad**.

Los ciclos rápidos de desarrollo garantizan que se tenga feedback inmediato de posibles errores, lo que permite arreglarlos y seguir iterando hasta llegar a resolver el problema que nos habíamos propuesto.

Veremos una técnica de programación que facilita este tipo de desarrollo conocida como *Desarrollo orientado por tests* o *TDD (*por las siglas en inglés de *Test-Driven Development)*. **TDD es un patrón de diseño** en el que, a grandes rasgos, l**os tests se codifican antes que el código fuente** que los haría pasar: merece la pena dedicar tiempo a repasar este patrón y compararlo con las alternativas de desarrollo usuales.

Pero, antes de llegar a eso, se explicará qué tipos de tests existen y nos centraremos en aquellos más aplicables en el día a día, o de más inmediata aplicación en las prácticas de la asignatura.

Veremos que aquí también la elección del lenguaje condiciona las tecnologías y frameworks disponibles. En cualquier caso, los conceptos y patrones se mantienen idénticos, tan solo hay que aprender con qué herramientas debemos trabajar.

Scripting

Se explicará qué es un script, qué nos aporta y cómo nos permite añadir funcionalidad o automatizar tareas comunes/pesadas:

- Facilitar construcciones o despliegues.
- Crear empaquetados complejos.
- Encadenar comandos con parámetros complicados.

Si bien existen lenguajes de *scripting* específicos de plataformas Windows (PowerShell, el más famoso), nos centraremos sobre todo en los scripts de *shell* de Linux, por ser los más comunes en la mayoría de los casos. No obstante, se ofrecerán los recursos necesarios para scripting en Windows.

Virtualización, Cloud y Contenedores

Durante estas semanas, el foco se va poner sobre **dónde** ejecutaremos nuestro software.

La provisión de entornos de compilación y ejecución de software es una tarea que tradicionalmente ha caído del lado de las Operaciones. Si bien esto no deja de ser así, como DevOps tendremos la oportunidad de acercar las aplicaciones y sus requerimientos a los entornos que necesitan para ser productivas.

Lo primero es definir un 'entorno tradicional', aunque realmente es la base de toda la infraestructura de operaciones de los sistemas de todo el mundo: los sistemas físicos.

Un sistema hardware físico va desde una Raspberry a un portátil o a un servidor dentro de un data center. Es un *ordenador* en el concepto más habitual de la palabra, aunque se suele reducir la descripción de un sistema por los recursos que ofrece:

- CPU.
- Memoria.
- Almacenamiento.

Es labor del sistema operativo la administración de estos recursos y, por otro lado, labor de los administradores de sistemas la instalación, mantenimiento y operación de las piezas software necesarias para tareas como:

- Compilación de aplicaciones.
- Entornos de desarrollo.
- Ejecución de servicios.

Virtualización

Un paso más allá a la hora de utilizar los recursos físicos es el de la virtualización: **aprovechar los recursos disponibles** en un servidor o en una red de ordenadores, o incluso en un data center, para generar 'máquinas virtuales' que los aislen y generen un entorno de ejecución o procesamiento.

El software de virtualización (denominado *hipervisor*) crea **una máquina virtual que imita todo el entorno de hardware**. El sistema operativo que está cargado en una máquina virtual es un producto estándar no modificado. Cuando realiza llamadas para recursos del sistema, el software de emulación de hardware captura la llamada del sistema y la redirige para que pueda gestionar estructuras de datos proporcionadas por el hipervisor. Es el propio hipervisor el que realiza las llamadas al hardware físico real, subyacente a toda la aglomeración de software.

La **emulación o imitación de hardware** también es conocida como *virtualización de metal desnudo* (del inglés *bare metal virtualization*), para simbolizar el hecho de que ningún software se encuentra entre el hipervisor y el 'metal' del servidor. Como hemos mencionado, el hipervisor intercepta las llamadas del sistema desde la máquina virtual huésped y coordina el acceso al hardware subyacente directamente.

Cloud Computing

Todos estos recursos de virtualización o ejecución en contenedores tienen algo en común: tienen que 'correr' en algún sitio y, normalmente, ese sitio no puede ser nuestro portátil, pues necesitamos ciertas garantías de:

- Disponibilidad de servicio.
- Capacidad extra de recursos de forma puntual.
- Posibilidad de pedir más recursos a demanda.
- Tolerancia a fallos.

En cualquier entorno, para poder **escalar** necesitamos poder tener más recursos disponibles:

- Si mi aplicación tiene problemas de memoria (o necesita más CPU para hacer un procesamiento), se añaden más recursos a la máquina → **scale-up**.
- Si una máquina no es capaz de atender todas las peticiones de red entrantes, tengo que añadir más máquinas → **scale-out.**

Para resolver estos problemas (y muchos otros) están las *clouds:* proveen recursos y servicios 'en la nube' para eliminar los problemas de:

- Poseer infraestructura.
- Configurarla.
- Mantenerla.
- Actualizarla o mejorarla.

Infraestructura como código

Hemos visto que podemos definir, de forma precisa, el entorno de ejecución de nuestro software.

Además, con el uso de Clouds o Virtualización, podemos también disponibilizar servicios añadidos, como pueden ser:

- Almacenamiento elástico.
- Bases de datos.
- Servicios de logs.
- Redundancia.

Llegados a este punto, nos volvemos a preguntar: ¿se puede controlar toda está necesidad de infraestructura de un modo organizado? ¿Cómo replicaríamos una puesta en marcha en otra cuenta o en otro entorno, o con otros parámetros?

Afortunadamente, existe una tecnología que nos permite **describir** nuestra infraestructura en lenguajes declarativos y, de este modo, guardarlos en un control de versiones junto con nuestra aplicación (o en otro repositorio). Está tecnología recibe el nombre de *infraestructura como código.*

Existen varias alternativas que debemos conocer:

- **Específicas**: relacionadas con las clouds públicas, como Cloudformation para AWS o Azure Resource Manager templates.
- **Genéricas**: como es el caso de Terraform.

Docker

Docker representa otra **tecnología de virtualización**, en este caso, **aplicada al sistema operativo**: es una plataforma de código abierto para desarrollar, distribuir y desplegar aplicaciones en entornos aislados y seguros denominados *contenedores*.

Un contenedor es una unidad de software estandarizada que empaqueta el código de una aplicación y todas sus dependencias para permitir su ejecución de manera rápida y consistente, independientemente del entorno en el que se ejecute.

Los contenedores tienen la característica de ser **livianos**, ya que no necesitan la carga adicional de un hipervisor como ocurre con las máquinas virtuales, sino que se ejecutan directamente dentro del núcleo de la máquina host. Esto significa que con un determinado hardware podremos ejecutar mayor número de contenedores que si estuviéramos utilizando varias máquinas virtuales.

La plataforma Docker nos permite empaquetar nuestras aplicaciones con **todo lo necesario para su ejecución (l**ibrerías, código, herramientas, configuraciones), eliminando así dependencias del sistema operativo de la máquina host y facilitando un despliegue muy rápido de nuestras aplicaciones.

Kubernetes

Kubernetes es una herramienta de código abierto que se utiliza para el despliegue y la gestión de contenedores. No se trata de una plataforma de contenerización, sino que ofrece una **gestión de aplicaciones multicontenedor**.

Esta herramienta fue desarrollada inicialmente por Google basándose en soluciones propias con las que la compañía había estado trabajando internamente durante años. La primera versión se presentó en el año 2014 y, desde entonces, ha ido evolucionando continuamente con nuevas funcionalidades.

Entre las principales características y capacidades de Kubernetes podemos encontrar:

- El empaquetado automático de contenedores.
- El descubrimiento de servicios y balanceo de carga.
- El almacenamiento desacoplado.
- La autoreparación.
- La gestión de la configuración de aplicaciones.
- La ejecución por lotes y tareas programadas.
- El escalado horizontal.
- El despliegue de actualizaciones automatizadas y sin cortes de servicio.
- Los rollbacks de despliegue a versiones previas.
- La gestión de los recursos del clúster.

Networking

En el mundo actual, la ubicuidad de las redes de ordenadores hace que frecuentemente nos olvidemos de la complejidad que supone tener tantísimos sistemas interconectados comunicando información de unos a otros a través de internet, redes móviles o incluso en las redes locales de cada hogar.

En la sección dedicada al networking daremos un paseo atrás y estudiaremos algunos **fundamentos de los sistemas de redes** que nos permitirán entender mejor ciertos problemas que, sin duda, tendremos que resolver:

- ¿Necesito usar SSL para la comunicación entre las máquinas de mi red interna?
- ¿Cómo puedo hacer para que mis servicios estén accesibles desde internet?
- ¿Cómo puedo acceder a una máquina en una red privada desde fuera de la misma?

Al terminar la unidad podremos responder mejor a preguntas de este tipo y entenderemos conceptos tan comunes como el *direccionamiento IP*, la *resolución de nombres con DNS* o el concepto de *API REST*, tan común como trascendental en la comunicación entre máquinas en el contexto tecnológico actual.

Conceptos de administración de sistemas

En este punto del curso ya contamos con herramientas suficientes para poder crear software de calidad, construirlo y hasta instalarlo y ejecutarlo en un contenedor, máquina física o virtual.

Pero nos falta una pieza fundamental: poder ejecutar diversos tipos de **operaciones** en los **diferentes sistemas** en los que se ejecuta nuestro software -o el de compañeros de equipo.

Un inciso: el manejo total del sistema operativo, así como el conocimiento de los diferentes comandos disponibles en los intérpretes de comandos (*shell)* más comunes, es algo que queda fuera del ámbito de esta asignatura.

Sistemas operativos

En la grandísima mayoría de los casos, el sistema operativo sobre el que se ejecuten los procesos será algún tipo de sistema Linux.

La razón principal es que Linux es un sistema operativo *libre,* lo que significa que no hay que pagar licencia por su uso: esto libera a las organizaciones de un problema de costes, aunque tiene sus contrapartidas.

¿Qué problemas puede traer el usar un sistema operativo libre para ejecutar nuestras operaciones? **El soporte.** Es por eso que existen distribuciones que ofrecen, a través de un sistema de licencias, diferentes opciones de soporte técnico.

Dentro de Linux existen varios tipos de sistemas, los que se pueden clasificar de muchas formas. Una clasificación podría darse si nos basamos en el sistema de paquetería que manejan:

- RPM: Red Hat Enterprise Linux, **CentOS**, Fedora.
- DEB: **Debian**, Ubuntu*.
- Pacman: Arch, Manjaro.

Si bien existen muchísimos más, los marcados en negrita conviene conocerlos, pues la práctica totalidad de las imágenes Docker o SO derivados en las nubes públicas están basados en ellos.

Reto

Encuentra las principales diferencias entre un sistema CentOS y uno basado en Debian.

Comparte tus conclusiones en el foro del aula y participa en el debate con tus compañeros.

Línea de comandos

Una vez conocemos qué sistema operativo estamos ejecutando, sabremos (más o menos) qué tipo de comandos de sistema tenemos disponibles y también podemos saber cómo maneja ese sistema operativo sus recursos.

Importante: a partir de este momento no existen las interfaces gráficas. El acceso y manejo de los sistemas suele ser mediante terminales de texto, ya sea directamente en la propia máquina o conectándonos vía *ssh.*

Existe un conjunto de comandos y conocimientos que todo DevOps debe poseer. La soltura en el manejo de estos entornos es clave para conseguir datos de nuestro sistema en poco tiempo, a la vez que nos permite crear *scripts* o complejas secuencias de comandos para obtener, por ejemplo:

- Ficheros de log que contienen una determinada respuesta http.
- La lista de ficheros más pesados en una serie de carpetas.
- Conectarnos a un servicio remoto y volcar la información en el sistema.
- Transformar datos de entrada en otros formatos totalmente diferentes.

Las posibilidades son inabarcables, por lo que veremos los problemas más comunes y se darán herramientas para aprender a resolverlos.

Securización del sistema

Una vez tenemos los conocimientos necesarios para manejarnos dentro de un sistema operativo, deberíamos habernos hecho ya muchas preguntas relativas a la seguridad de los datos y a las operaciones:

- ¿Quién puede acceder al sistema?
- ¿Cómo puedo evitar que los usuarios modifiquen la configuración?
- ¿Cómo evitar que un usuario malintencionado deniegue el acceso?

Repasaremos los conceptos básicos de **control de acceso y perfilado en Linux y cómo funcionan los permisos a nivel de fichero/directorio**.

Esto es solo el principio, donde quedará claro que es una carrera en la que el administrador casi siempre va por detrás de los atacantes, pero no es poco lo que se puede hacer. Al terminar está lección tendremos una idea de los vectores de ataque más comunes y cómo podemos proteger nuestro sistema.

Orquestación de sistemas: Ansible

Si algo falta en esta lección, y a estas alturas ya debería estar más que presente, es la necesidad de **automatizar** todo lo posible el proceso de instalación, configuración y securización de los sistemas.

Ya tengamos una única máquina física o una flota de virtuales, no importa: las operaciones manuales son fuentes de errores humanos, no se pueden verificar ni probar adecuadamente y resulta mucho más difícil auditar un proceso *a posteriori* para saber qué ha podido salir mal (porque algo saldrá mal).

Para esta tarea, una herramienta muy potente y disponible de forma gratuita es Ansible: se trata de una herramienta de automatización de IT, que puede configurar un sistema, desplegar paquetes e, incluso, orquestar tareas avanzadas como la gestión de un despliegue Zero-downtime, por ejemplo.

Está pensada para ser sencilla de usar y está basada en ficheros en formato YAML en los que se especifica qué tareas deben realizarse, en qué orden y con qué parámetros (servidores en los que aplicar los cambios, variables de entorno que inyectar, parametría genérica, etc).

Herramientas de gestión de ciclo de vida

En esta lección se definirá un concepto de suma importancia en DevOps: el ciclo **de vida del software.**

Según Wikipedia, el ciclo de vida está compuesto por una serie de fases, claramente definidas y diferenciadas, que son utilizados por ingenieros y desarrolladores de sistemas para **planificar, diseñar, construir, probar y entregar sistemas de información.**

El objetivo no es otro que producir sistemas de alta calidad que cumplan o excedan las expectativas del cliente, siempre dentro de las restricciones de tiempo y coste.

Si bien existen muchas y muy diferentes maneras de hacerlo, aquellas en las que nos centraremos como DevOps son las que nos permitan hacerlo de manera ágil (por *Agile),* como se ha visto al principio del temario.

Sistemas de control de versiones: git

El eje principal en la gestión de software es su almacenamiento versionado. Y, en esto, la herramienta por excelencia es *git,* con el permiso de otras herramientas que ni siquiera vale la pena mencionar.

La trascendencia de esta herramienta es tal que el concepto de *open source* como lo conocemos **no tendría sentido sin ella**.

Git es un **sistema de control de versiones distribuido** en el que el código fuente se almacena en un servidor central organizado en *repositorios*, pero cada usuario es libre de almacenar una copia local y subirla a donde desee.

Este servidor central almacena una o más *revisiones* del código, que popularmente conocemos como *branches* o ramas. Los usuarios se 'traen' (*pull)* los cambios de esas ramas a sus estaciones de trabajo y hacen lo que tengan que hacer.

Cuando tengo cambios que creo que deben llegar a los demás usuarios, los añadiría (*commit)* a esa rama y después los 'empujaría' (*push)* al servidor central. Si todo va bien y nadie más ha modificado esa rama, el código estará disponible para que cualquiera pueda volver a hacer *pull* y tener mi aportación disponible.

Esto es el flujo habitual y, como veremos, admite multitud de matices y existen una serie de problemas que tendremos que resolver.

Interfaces web

Cuando instalamos *git* en nuestro sistema operativo, lo que realmente estamos es instalando un 'cliente' que se conectará a uno o más servidores de los que traerá el código fuente.

Estos servidores ofrecen interfaces web muy avanzadas para facilitar el uso de la herramienta, enfocadas siempre a la visualización, compartición y aportación de código fuente a los repositorios.

Sin duda, las más comunes son gitlab, Github y Bitbucket. En todos estos casos existen versiones gratuitas y de pago (pensadas para empresas u organizaciones con ánimo de lucro).

Las cuentas gratuitas permiten tener repositorios públicos ilimitados (o un número bastante alto, al menos) y es una práctica usual tener una cuenta en alguno de ellos y subir allí ciertos repositorios personales como, por ejemplo:

- Programas que hemos hecho y de los que nos apetece fardar un poco, aunque también con la esperanza de que alguien encuentre algo que le inspire.
- Katas o similares.
- *Forks* de proyectos *open source*.
- *Pet projects.*

Te recomiendo encarecidamente hacerte una cuenta en alguno de ellos desde ya, aunque creas que no la vas a usar.

Versionado y modelos de ramas

Ahora ya sabemos dónde vamos a guardar nuestro código fuente, sabemos que podemos tener almacenadas versiones del mismo y estamos en condiciones de entregar una nueva funcionalidad.

Llegados a este punto, y si se ha leído algo de literatura al respecto, es posible que exista un poco de confusión en cómo empezar a trabajar con Git en un equipo.

Esto es perfectamente entendible y, desde luego, no existe una respuesta única: depende del tamaño del equipo, de la tipología de los ciclos de *release*, del tiempo que se necesita para pasar tests o para desplegar….

En todo proyecto Git hay una rama principal: ***main/master*** (usaremos el nombre *main* a partir de ahora, pues es el nombre oficial en Github desde 2020). Está rama es en la que se supone que se debe encontrar siempre la fuente de verdad de nuestro código fuente.

Cuando los despliegues de versiones no son constantes/relativamente frecuentes, lo habitual (no tiene por qué existir) es que exista una rama llamada ***develop*** en la que se trabajará durante un tiempo, para proceder a 'lanzarla' (***release***) y marcarla con un número de versión.

Adicionalmente, se suelen usar una serie de ramas con una nomenclatura específica:

- ***feature/ -*** Cuando se trabaja en una nueva característica del software (*feature)* durante una release.
- ***hotfix/ -*** Cuando se arregla un error presente en producción (rama main).
- ***bugfix/*** - Cuando se arregla un error introducido en una release, antes de que esta llegue a main (y, por tanto, a producción).

Queda claro que esto puede volverse bastante lioso, pero, afortunadamente, existen unos patrones de trabajo llamados ***modelos de ramas*** (*branching models)* que, con una serie de reglas bien definidas, tratan de poner un poco de orden en el proceso: Git Flow, GitHub Flow, Microsoft Flow…

El problema que tenemos ahora es: ¿cómo podemos anotar esas diferentes versiones de modo que tengan cierta coherencia y sigan algún tipo de estándar?

La respuesta nos la da el **versionado semántico,** con el que, a grandes rasgos, cada versión de nuestro software estará en el formato *x.y.z*

- ***x (major) -*** Suele marcar una familia o línea de versiones concreta, su incremento no garantiza la retrocompatibilidad con la anterior.
- ***y (minor) -*** Dentro de una línea de versiones, marcan incrementos de funcionalidad que son retrocompatibles entre sí.
- ***z (patch)*** - Dentro de una versión concreta, marcan incrementos sin funcionalidades relevantes, usualmente correcciones de errores.

Pull Requests & Code Reviews

Afortunadamente, no estamos solos en el mundo. No somos, o no deberíamos ser, los únicos responsables del código fuente.

Una de las grandes ventajas del control de versiones distribuido (VCS por sus siglas en inglés) es la facilidad que aporta al trabajo en equipo y, concretamente, a la revisión por parte de compañeros del código que aportamos para cumplir una funcionalidad concreta.

El mecanismo de *pull request (o merge request)* es aquel por el que solicitamos aportar código a una rama 'principal' de desarrollo, de la cual parte el trabajo del resto de componentes del equipo.

Una vez esté hecha está solicitud, el resto del equipo puede **revisar** el trabajo, inspeccionar visualmente el código y aportar pistas o consejos que lo puedan hacer mejor.

Podrían incluso rechazar la solicitud si no se cumplen una serie de requisitos de integración, como no tener una cobertura mínima de tests, no seguir los estándares de escritura del código, no estar documentado…

Este proceso por el cual una solicitud pasa por la revisión de nuestros pares y, tras un periplo, llega a formar parte de una rama de integración, se llama ***code review*** **(peer review)**. Supone otra de las razones por las que herramientas web como las mencionadas son tan potentes y tan necesarias: **democratizan el código fuente** y ayudan a que todo el equipo sienta que ha aportado algo en cada nueva funcionalidad que se entrega.

Publicación de código

Una vez que tenemos generada una versión de nuestro software y hemos usado las herramientas de construcción apropiadas para obtener un ejecutable, surgen más preguntas:

- ¿Dónde podemos almacenar sucesivas versiones del mismo software?
- Si mi software tiene dependencias de otras piezas, ¿dónde se almacenan?
- ¿Puedo llevar el ejecutable a su destino o necesito una pieza intermedia?

Las herramientas que cumplen estas características son los **almacenes de artefactos**.

Existen varios tipos de almacenes según el tipo de paquete en el que se especializan: Docker Registry, Maven Central, PyPi… También existen herramientas que los aúnan todos tras una interfaz común, como Artifactory o Nexus.

La elección del tipo de herramienta dependerá totalmente de cada caso de uso: diferentes organizaciones pueden tener requisitos cambiantes de conectividad, seguridad, etc.

Integración Continua y Despliegue Continuo (CICD)

Tenemos ya casi todas las piezas en su sitio. Sabemos construir el software, etiquetar el resultado y 'subirlo' a un almacén de artefactos. Nos falta desplegar y listo, ¿verdad?

La verdad es que hasta ahora todo el proceso se ha hecho de modo manual, y cualquier proceso manual es tendiente a fallos: el error humano es el más difícil de evitar, por lo que c**abe preguntarse si es posible automatizar este proceso.**

La respuesta es obviamente afirmativa: existen soluciones que se encargan de automatizar una serie de pasos por nosotros, son capaces de enviar feedback de cada paso que se da y tienen la capacidad de tomar decisiones en base a datos:

- Saber cuándo deben empezar a construir un ejecutable.
- Saber si un ejecutable cumple los mínimos exigibles para promocionar a un entorno diferente (pruebas, QA, producción…).
- Sabe enviar un correo al responsable de una nueva característica, por qué rompe tests de integración, así como detener el proceso de release.
- Saber si debe hacer rollback de una release porque está fallando algo.

Son muchas las ventajas de tener sistemas de **integración continua**, llamados así porque su labor es la de escuchar eventos en un VCS y, en caso de encontrar cambios en una o más ramas, empezar a ejecutar una serie de pasos:

- Tests unitarios y de integración.
- Análisis de seguridad y calidad de código.
- Empaquetado.
- Publicación de resultado en un almacén.

En el caso de que, además, la herramienta sea capaz de desplegar automáticamente a entornos productivos y monitorizar el resultado para dar marcha atrás en caso de errores, estaríamos hablando de **sistemas de despliegue continuo**.

Jenkins

Jenkins es una herramienta open source de automatización que usaremos para aprender los básicos de los sistemas de CICD. Entre sus características más interesantes, destacaremos:

- Extensiva comunidad, al tratarse de un proyecto open source con muchísima actividad y formar parte de la lista de proyectos de CD Foundation.
- Extensible vía plugins, con más de 1500 ya disponibles y la posibilidad de crear uno nuevo para nuestras necesidades particulares.
- Las pipelines se programan en lenguaje Groovy, por lo que son una parte más de nuestro proyecto y, como tal, se guardan en control de versiones.
- Por último, se pueden importar/exportar pipelines para compartir funcionalidades con otros usuarios, aumentando de forma dramática la escalabilidad de los proyectos.

Despliegues y monitorización

Está será la sección en la que veremos las diferentes maneras de desplegar nuestro software: qué maneras existen para que pueda empezar a hacer aquello para lo que lo hemos programado, ya sea en entornos de desarrollo, de calidad (QA), productivos…

Hay muchos tipos de aplicaciones posibles y cada una de ellas tendrá su propia manera de ser desplegada:

- Aplicaciones accesibles vía endpoints HTTP.
- Algoritmos de procesamiento de datos.
- Modificaciones a una base de datos.
- Apps móviles.

Nos vamos a centrar ahora en un tipo concreto, bastante común en la industria y que merece la pena ser estudiado con detalle: **las aplicaciones web**, o aplicaciones que son accesibles mediante endpoints HTTP.

A grandes rasgos, lo que veremos será:

- Cómo configurar el despliegue y diferentes técnicas de despliegue disponibles en Cloud:
    - Traditional deploy.
    - Canary.
    - Blue-Green.
- Cómo funciona un balanceador de carga.
- Cómo monitorizar un servicio en marcha.

Logging, Tracing y recolección de métricas

Estas técnicas son formas **complementarias** de obtener datos de lo que está ocurriendo en servicios que están ejecutándose en nuestra infraestructura.

- *Logging* se refiere a la emisión de trazas que nuestra aplicación escribe durante su operación. Estas trazas suelen ser escritas durante la fase de implementación con el objetivo de poder saber qué datos llegan a diferentes secciones del código o tener un detalle de los mensajes de error.
- *Tracing* es una técnica algo más avanzada y compleja en la que, mediante la propagación de una serie de identificadores a lo largo del camino de ejecución del código de nuestra aplicación, podemos trazar el camino que sigue una petición cualquiera, saber por qué métodos ha pasado, cuánto tiempo ha estado en cada uno, etc. Es una técnica muy potente pero que necesita que el framework en cuestión lo provea.
- La *recolección de métricas* es conceptualmente similar al *logging,* pero lo que se almacena son datos cualitativos de la ejecución: tiempo en ejecutar ciertas funciones o cálculos, número de veces que se llama a un endpoint…

Todos estos datos tienen algo en común y es que pueden guardarse en ficheros de sistema para su posterior explotación o pueden enviarse a diferentes servicios de recolección y proceso de datos, que se encargarán de su almacenamiento, clasificación y visualización. Ejemplo de estos sistemas son:

- Logstash.
- Fluentd.
- DataDog.
- Prometheus.
- Kibana.
- Elastic Search*.
- (AWS) CloudTrail, CloudWatch.

DevOps en la organización

Dentro de organizaciones o equipos de cierto tamaño, el concepto de DevOps como revolución de la manera de trabajar de los equipos tiene que evolucionar para poder escalar adecuadamente junto a la empresa.

Es probable que cada subequipo siga organizándose de la misma forma, pero, a medida que la complejidad tecnológica aumenta, más probable es que existan sistemas/plataformas/tecnologías que dependen de equipos centrales o *core*, para concentrar todo el ciclo de vida en un único equipo o área.

Un área de Platform/Core/Global Services presenta una oportunidad única de aumentar el impacto de DevOps en las organizaciones. Dentro de esta área global es usual que haya equipos encargados de:

- Mantener una o más tecnologías 'troncales', que forman el pilar sobre el que se sustenta una compañía tecnológica.
- Ofrecer herramientas como servicio, accesibles a todos los equipos y que proporcionan Control de Versiones, CICD, métricas, logging...
- Crear y mantener líneas de contacto con diferentes áreas o geografías, para poder alinear objetivos y estrategias, aportar ayuda puntual.
- Medir el impacto de las estrategias globales y ayudar a mejorar el trabajo de equipos en base a datos.

Adopción DevOps

Vamos a centrarnos en estos últimos conceptos, ya que constituyen las metas y visiones de *Adopción DevOps* o *DevRel*.

Es necesario que existan áreas/personas con que trabajen de forma transversal, en colaboración con aquellos equipos donde se produce valor para la organización: productos, tecnologías *core*, I+D, ingeniería de datos…

Los objetivos principales de Adopción DevOps podrían ser:

- Asegurar la alineación con los objetivos de la organización.
- Servir como enlaces entre global y local.
- Aumentar el nivel DevOps de equipos locales.
- Servir como agentes del cambio.

Herramientas de Gestión Agile

Todos los equipos necesitan llevar un control de las tareas en curso y cierta gestión de su capacidad de abordar nuevas tareas. Pero, a medida que aumenta el tamaño de una organización y que aumentan...

- Las dependencias con otros equipos/herramientas,
- La necesidad de prever la capacidad a futuro,
- Y el tamaño del backlog,

Se hace más patente la necesidad de usar herramientas que permitan gestionar de forma eficiente las tareas en curso y las que vendrán, para poder tener:

- Una visión holística de hacia dónde va la organización.
- Los focos de los diferentes equipos de la misma.
- Identificar puntos de bloqueo y puntos fuertes.
- Organizar recursos de forma eficiente.

Todo esto se puede conseguir con un **uso adecuado de herramientas de gestión de equipos.** Si estos equipos usan alguna disciplina Agile (como, por ejemplo, Scrum), entonces existen herramientas específicas para esto, como por ejemplo JIRA o, en menor medida, Trello.

En Scrum, el trabajo de los equipos se organiza en **sprints**, usualmente de una o dos semanas, en los que se trabaja en una serie de tareas que han sido acordadas por el equipo al inicio del sprint, en base a su capacidad y a la velocidad percibida.

Esas tareas o *historias de usuario* son las partes que componen una *feature*, una nueva característica que vamos a añadir a nuestro producto o nuestra aplicación, y cuya descomposición en tareas corresponde únicamente al equipo.

Existen múltiples variantes de todo esto y, además, hay una serie de eventos que acompañan al proceso para poder tener feedback de cómo avanzan los sprints, valorar esa velocidad, etc. Pero, al final, la base de todo es que **es el equipo el que se organiza** y esto es justamente lo que hace de este modelo algo totalmente compatible con la cultura DevOps.

Preparación a los retos del curso

A la vista del contenido del curso, debe quedar claro que se necesita cubrir un amplio abanico de habilidades.

La idea de las actividades prácticas es la de añadir funcionalidad de **manera incremental,** creando las partes de una aplicación compleja poco a poco, añadiendo valor en cada iteración y entendiendo cuándo vale la pena conseguir un resultado 'como sea' y cuándo es conveniente parar y echar la vista atrás.

No obstante, debemos aprovechar cada iteración para detenernos a evaluar cuál es la mejor solución posible: en ocasiones no quedará más remedio que hacerlo de cero, pero la mayoría de las veces tendremos que evaluar si existe algo que se pueda aprovechar, ya sea una solución comercial o una herramienta open source.

Herramientas útiles

Algunas herramientas que se considera que ayudarán en la realización de retos, actividades y prácticas son:

- Línea de comandos.
- Navegador y extensiones.
- Editores e IDEs.

Línea de comandos

El manejo con soltura de diferentes intérpretes de línea de comandos se considera una aptitud muy valiosa. En concreto, el manejo de Bash y similares (*zsh,* por ejemplo) o PowerShell **proporcionan velocidad y control al ingeniero DevOps** de una manera que pocas otras herramientas consiguen igualar.

Esto es así pues muchas veces las herramientas gráficas, al esconder la complejidad de las tecnologías que utilizan, hacen que nos olvidemos de cómo funcionan realmente por debajo de las mismas. Ejemplos de herramientas o comandos que se recomienda conocer son:

- git
- curl / wget
- sed / awk
- Diferentes comandos del sistema operativo actual.

Navegador y extensiones

Si bien puede parecer algo que para gustos, colores, la elección de un navegador u otro tiene **más trascendencia de lo que parece,** en especial cuando estamos desarrollando aplicaciones web.

Pero no solo en aplicaciones web, pues aquellos que proporcionan sistemas/tiendas de extensiones (que ahora mismo son todos, pero algunos llevan años haciéndolo y se nota en la variedad) pueden convertir un navegador web en auténticos entornos de trabajo, en aplicaciones especializadas en seguridad o, simplemente, en una aplicación para surfear por internet de un modo totalmente adaptado a nuestros gustos personales.

Aplicaciones a mencionar son:

- Extensiones para conectarnos a *proxies,* como FoxyProxy de Firefox.
- Consolas de desarrollador para ejecutar comandos JavaScript o inspeccionar contenidos HTML (Chrome y Firefox).
- ScreenCasting.
- *AdBlockers* / gestores de seguridad.

Editores e IDEs

Lo primero es comentar que cada experto tendrá sus preferencias en cuanto a su herramienta preferida para editar código, ejecutar aplicaciones, etc. Hay mucha variabilidad, pero lo más destacable es la soltura de manejo que cada persona tenga con las herramientas con las que desarrolle su profesión.

Con todo, se recomienda tener soltura en al menos una herramienta de las siguientes para la edición de código fuente (aunque provean de muchas más cualidades):

- **vim / emacs:** al margen de la eterna pelea sobre cuál de las dos es la mejor, estas dos herramientas proporcionan una potencia de edición prácticamente ilimitada a cualquiera que se tome el tiempo necesario para aprender todos sus entresijos. No es tarea fácil, pero vale la pena.
- **Eclipse / IntelliJ (y derivados):** estas herramientas, altamente especializadas, suelen ser la ventana habitual para desarrolladores Java / Android. Pero también existen variedades especializadas en Go, Python... muy potentes y, además, altamente extensibles. Su problema suele ser el consumo de recursos, ya que es bastante alto.
- Visual Studio / VSCode: alternativas mucho más ligeras a las anteriores, pero con una variedad de plugins tan descomunal que permiten lo mismo, sino más a costa de necesitar más tiempo para llegar a tener un entorno con el que nos sintamos cómodos.

Bibliografía y webgrafía básica

- Pro Git (free online version) (2nd Edition 2014) - Scott Chacon & Ben Press.

LA UNIVERSIDAD EN INTERNET | FORMACIÓN PROFESIONAL

PROEDUCA