



CERTIFIED TECH DEVELOPER

Infraestructura **II**

PRIMER AÑO
TERCER BIMESTRE



Contenido

Módulo 1: Inmersión.....	5
Ejes de la materia.....	5
Infraestructura como código.....	5
Pipelines de CI/CD.....	5
Monitoreo.....	5
Conexión con otras materias	5
Infraestructura I	5
Back End I.....	5
Testing I.....	6
DevOps.....	6
¿Qué es y cómo surge DevOps?	6
Herramientas de DevOps	8
Perfiles en el ecosistema DevOps.....	9
Expectativas en el ecosistema DevOps.....	10
Resumen de la Semana	11
Ejes de la materia.....	11
¿Qué es DevOps?	11
Perfiles de DevOps	11
Expectativas de DevOps	11
Módulo 2: Infraestructura como Código.....	11
Infraestructura como código: El Concepto.....	11
Creación de una máquina virtual en Amazon Web Services (EC2)	12
Antes de la Infraestructura como código	13
Mejoras y soluciones	13
Beneficios de la Infraestructura como código.....	13
Paradigmas para la infraestructura como código	14
Paradigma Imperativo.....	15
Paradigma Declarativo	15
El principio de Idempotencia	15
Ecosistema de herramientas IaC	16
Terraform.....	16
AWS CloudFormation.....	16
Azure Resource Manager	16
Google Cloud Deployment Manager.....	16
Ansible.....	17

CloudFormation.....	17
¿Cómo usarla?	17
¿Dónde la usamos?	19
¿Quién la usa?.....	20
Resumen de la Semana	20
Ansible.....	21
¿Cómo usarla?	21
¿Dónde la usamos?	24
¿Quién la usa?.....	24
Terraform.....	25
¿Cómo usarla?	26
¿Quién la usa?.....	28
Los plugins de Terraform (Clase 10)	29
Domain Specific Language (Clase 10).....	31
Terraform HCL (Clase 10)	31
Ambiente Productivo (Clase 10)	34
Resumen de la semana	40
CloudFormation vs Ansible vs Terraform.....	40
Resumen de la Semana	41
Módulo 3: Pipelines	42
Pipelines ¿qué son y para qué sirven?.....	42
¿Qué tecnologías existen?.....	44
Jenkins	45
Características	45
Instalación.....	45
Configuración	45
Distributivo	45
Plugins.....	46
Jenkinsfile	46
Conociendo Jenkins a través del Jenkinsfile.....	47
El proceso de build	48
Proceso de compilación usando Jenkins	48
Continuous integration	50
Características	50
¿Qué son los triggers?.....	51
Artefacto.....	52

Navegando entre artefactos	52
Administración.....	52
Metadata.....	52
Almacenamiento	53
Accediendo a nuestros artefactos.....	53
Todo alrededor de una filosofía	53
Conclusión.....	53
Pasos para el desarrollo de Software.....	54
El principio de inmutabilidad	54
Resumen de la Semana	55
CD: Despliegue continuo.....	55
Continuous Delivery vs Continuous Deployment	56
¿Una diferencia clave!.....	56
Examinemos ambos procesos	57
Pipelines: CI/CD.....	57
Combinamos Deployments.....	59
¿Cómo hacer un pipeline de CI/CD exitoso?	61
Resumen de la semana	62
Módulo 4: Monitoreo.....	62
¿Qué significa monitorear?.....	62
¿Qué buscamos monitorear?	64
Beneficios del monitoreo.....	64
Monitoreo de infraestructura y de aplicaciones.....	65
Herramientas para monitorear	65
La importancia de las métricas	66
¿Qué es medir?	66
¿Qué son los indicadores?	67
Métricas y degradación.....	67
Tipos de métricas	67
Monitorear infraestructura.....	68
Amazon CloudWatch.....	68
Autoescalado y Elasticidad	71
Resumen de la semana	72

Módulo 1: Inmersión

Ejes de la materia

Infraestructura como código

La infraestructura como código es la gestión de la infraestructura en un modelo descriptivo utilizando las mismas herramientas de versionado que un equipo utiliza para su código fuente. Así como el mismo código fuente genera el mismo código binario, un modelo de infraestructura como código *debe generar el mismo entorno cada vez que se aplica*. La infraestructura como código, en conjunto con los pipelines de despliegue continuo, permite automatizar los despliegues de infraestructura haciéndolos más rápidos y menos propensos a errores, además nos evita depender de un equipo de infraestructura.

Pipelines de CI/CD

La **integración continua (CI)** es una práctica de desarrollo que consiste en integrar el código a un repositorio compartido de manera frecuente, idealmente varias veces al día. Cada integración es verificada por un proceso automatizado permitiendo a los equipos detectar problemas rápidamente.

El **despliegue continuo (CD)** es la capacidad de poner en producción, en manos de los usuarios, cambios de cualquier tipo (nuevas funcionalidades, cambios de configuración, soluciones de errores y experimentos) de manera segura y sostenible. Esto se logra al asegurarnos que el código se encuentra en un estado desplegable incluso al hacer cambios constantes.

Estas dos prácticas se llevan a cabo mediante **pipelines de CI o CD** respectivamente, que son procesos automatizados por los que pasa el código (código fuente o binarios) hasta llegar a su destino final, que puede ser un entorno de pruebas o entorno de producción.

Monitoreo

El monitoreo se divide en dos grandes ramas:

- El **monitoreo de aplicaciones**: Es el proceso de medir la performance, disponibilidad y experiencia de usuario de una aplicación. Estas métricas se utilizan para identificar y resolver problemas en la aplicación antes de que impacten a los usuarios.
- El **monitoreo de servidores**: Es el proceso de ganar visibilidad respecto a la actividad de nuestros servidores, sean físicos o virtuales. Se pueden enfocar en distintas métricas de los servidores, pero las principales son la disponibilidad y la carga.

Conexión con otras materias

Infraestructura I

En Infraestructura I vimos las bases, las diferentes tecnologías que vamos a poder utilizar y nos van a habilitar a desplegar, configurar, y monitorear nuestras aplicaciones.

Back End I

- **Maven**: Esta herramienta de Back End I se puede utilizar manualmente y embebida dentro de nuestros procesos automatizados de build y release, también conocidos como pipelines.
- **Testing y JUnit**: Automatizar el despliegue de una aplicación no se trata solo de instalarla, sino que hay un conjunto de validaciones a ejecutar tanto durante el proceso de

compilación como durante el proceso de liberación para verificar el correcto funcionamiento de la misma. Los procesos modernos de infraestructura son la amalgama de todas estas actividades.

- **REST APIs:** Las APIs son contratos entre nuestros sistemas, formas estandarizadas de intercambiar información, ejecutar acciones y tomar decisiones utilizando mensajes protocolizados. Cuando interactuamos con un proveedor de nube, ya sea por medio de herramientas de scripting o por herramientas de infraestructura como código, lo que está sucediendo por debajo es que estamos consumiendo una o un conjunto de APIs.
- **Docker:** Es nuestro amigo de Introducción a la Informática e Infraestructura I. Esta herramienta nos va a permitir construir nuestras aplicaciones en un formato trasladable y que no dependa de recursos externos al contenedor en sí mismo. Es muy común ver en los procesos de build y release la dockerización de la aplicación en cuestión.
- **Microservicios:** Arquitectura en la que nuestra aplicación se distribuye entre varios componentes más pequeños, especializados, que resuelven problemas específicos. Para poder construir de manera dinámica y ágil estos componentes podemos hacer uso de los procesos de build y release.

Testing I

- **Api testing:** Las pruebas de APIs pueden realizarse de forma automatizada como pasos dentro de un pipeline de CI/CD, facilitando de esa manera su ejecución reiterada (por ejemplo, en cada build).
- **Automatización de pruebas:** En el mundo moderno de infraestructura la automatización de pruebas acelera el proceso de compilación, distribución y despliegue de las aplicaciones, eliminando el factor del error humano y ahorrando el costo producido por la repetibilidad de tareas.

Clase 2: ¿Qué es DevOps?

DevOps

¿Qué es y cómo surge DevOps?

Las empresas de software durante muchos años mantuvieron la cultura de trabajo donde los distintos departamentos involucrados en el desarrollo de un producto estaban aislados y separados uno de los otros, desarrollo, operaciones, control de calidad, herramientas, infraestructura, plataforma, seguridad, entrega. Todos incomunicados avanzando a ciegas pasándose el bastón al siguiente departamento como ocurre en una carrera de relevos hasta llegar a la meta final: la entrega del producto. Esta división e incomunicación hacía de los errores el común denominador generando grandes fricciones, mayores tiempos de desarrollo y por ende altos costos para la empresa. Es por esto que, con el tiempo, ante la necesidad de ser más ágiles, nace DevOps.

DevOps es la combinación de filosofías, prácticas y herramientas que incrementan la velocidad a la que una organización entrega aplicaciones y servicios, permitiendo mejorar los productos a un ritmo más rápido que las organizaciones que usen procesos de desarrollo e infraestructura tradicionales. Esta velocidad les permite a las organizaciones entregar más valor a sus clientes y ser más competitivas en el mercado.

En la cultura DevOps, los equipos de desarrollo (Dev) y operaciones (Ops) no trabajan por separado, sino que se comunican de forma constante. A veces los equipos se fusionan en un único equipo que

trabaja en el ciclo de vida completo de la aplicación, desde el desarrollo y las pruebas hasta el despliegue y la operación. En algunos casos, la calidad y la seguridad también se integran con el desarrollo y las operaciones.



Estos equipos hacen foco en automatizar procesos que históricamente fueron manuales y lentos. Para eso *utilizan un stack de tecnologías y herramientas* que los ayuda a operar y evolucionar aplicaciones de forma rápida y confiable. Además, colaboran a que una sola persona pueda realizar tareas que normalmente requerirían de una combinación de varias personas, como desplegar código o aprovisionar infraestructura.

DevOps significa que *el equipo de desarrollo es responsable del código en todas las etapas*, desde el desarrollo hasta la producción minimizando la fricción entre el desarrollo de nuevas funcionalidades y la operación de software actualmente en producción, reduciendo el tiempo de entrega de una feature y aumentando la frecuencia de los despliegues creando un entorno propicio para experimentar de una manera segura, dándole más agilidad a la organización y permitiéndole ponerse por delante de sus competidores.



Herramientas de DevOps

Implementar una cultura DevOps en una organización requiere del uso de herramientas fundamentales para automatizar procesos, monitorear recursos y lograr la agilidad que se persigue.

¿Cuáles son las herramientas?

CONTROL DE VERSIONES: Es la práctica de llevar un registro y gestionar los cambios que se hacen en el código fuente del software. Para esto existen los sistemas de control de versiones que llevan un registro de todas las modificaciones al código en una base de datos especial. Si se comete un error los desarrolladores pueden deshacer estos cambios y volver a una versión anterior del código.

CONTENEDORES: Un contenedor es unidad estándar de Software que empaqueta el código y a todas sus dependencias para que la aplicación pueda funcionar de forma confiable en distintos entornos de cómputo.

ORQUESTADORES DE CONTENEDORES: Se ocupan del despliegue, gestión, escalamiento, conectividad y disponibilidad de las aplicaciones basadas en contenedores.

MONITOREO DE LAS APLICACIONES: Es el proceso de medir la performance, disponibilidad y experiencia de usuario de una aplicación y usar estos datos para identificar y resolver problemas antes de que impacten a los futuros usuarios.

MONITOREO DE SERVIDORES: Es el proceso de ganar visibilidad respecto a la actividad de los servidores, sean físicos o virtuales, se pueden enfocar en diferentes métricas, pero las principales son la disponibilidad y la carga.

GESTIÓN DE CONFIGURACIÓN: Es un proceso que lleva registro de las distintas configuraciones que un sistema adopta a lo largo de su ciclo de vida.

INTEGRACIÓN CONTINUA (CI). Es una práctica de desarrollo que consiste en integrar el código a un repositorio compartido lo más frecuentemente posible idealmente varias veces al día. Cada iteración es verificada por un proceso automatizado permitiéndole a los equipos detectar problemas rápidamente.

DESPLIEGUE CONTINUO (CD): Es la habilidad de poner en producción, es decir, poner en manos de los usuarios cambios de cualquier tipo de manera segura y sostenible. Esto se logra asegurándonos de que el código siempre se encuentre en un estado desplegable incluso cuando se están haciendo cambios constantemente.

AUTOMATIZACIÓN DE PRUEBAS: Es un software que hace uso de herramientas de automatización para controlar la ejecución de las pruebas. Luego los resultados de estas son comparados con los resultados esperados para determinar si las pruebas han tenido éxito o no. La automatización de pruebas reduce el tiempo de ejecución de estas y minimiza la tasa de error humano.

INFRAESTRUCTURA COMO CÓDIGO: Es la gestión de infraestructura como un modelo descriptivo utilizando las mismas herramientas de versionado que un equipo utiliza para su código fuente. En conjunto con pipelines del pliego continuo permite automatizar los despliegues de infraestructura haciéndolos más rápidos y, al no depender de un equipo humano, es menos propenso a errores.

COMPUTACIÓN EN LA NUBE: Es el uso de servicios de cómputos a través de internet. Esta herramienta permite acelerar la innovación mediante el uso de recursos flexibles y el aprovechamiento de economías de escala. Típicamente sólo se paga por los servidores que se consumen ayudando así, a reducir los costos de operación y haciendo funcionar la infraestructura de una forma más eficiente.

Aunque el uso de esta y otras herramientas es clave para automatizar y monitorear todos los procesos debemos tener en cuenta que DevOps no consiste en sólo utilizar herramientas, el uso de estas da el soporte a las prácticas que se implementan a nivel cultural dentro de una organización y la agilidad viene de utilizar estas herramientas para hacer realidad esas prácticas.

Algunas herramientas de los DevOps son:

- Selenium
- Docker
- Puppet
- GitLab
- Chef
- CloudFormation
- Bamboo
- AWS
- Azure
- Jenkins
- Kubernetes
- Splunk
- Terraform

Perfiles en el ecosistema DevOps

DESARROLLADORES DE APLICACIONES: Son quienes desarrollan la aplicación, los programadores front-end, back-end, mobile, full stack o especializados en una tecnología particular —como Solidity— o plataforma — por ejemplo, Internet de las cosas (IoT)—. En un entorno DevOps es importante que se comuniquen constantemente con los demás roles.

ANALISTAS DE CALIDAD (QA): Son quienes verifican y validan la aplicación. En un entorno DevOps es importante que también se concentren en automatizar pruebas para hacerlas repetibles y confiables.

ANALISTAS DE INFRAESTRUCTURA: Son quienes implementan la infraestructura sobre la cual se ejecutarán las aplicaciones y las bases de datos. También se ocupan del mantenimiento y la evolución de esta infraestructura. Buena parte de las prácticas de DevOps recaen sobre ellos, en especial la comunicación con quienes desarrollan la aplicación. Dado que muchas veces la infraestructura existe en la nube, también se los suele llamar analistas clouds o analistas de nube.

ANALISTA DE REDES: Son quienes implementan la infraestructura sobre la cual se ejecutarán las aplicaciones y las bases de datos. También se ocupan del mantenimiento y la evolución de esta infraestructura. Buena parte de las prácticas de DevOps recaen sobre ellos, en especial la comunicación con quienes desarrollan la aplicación. Dado que muchas veces la infraestructura existe en la nube, también se los suele llamar analistas clouds o analistas de nube.

ANALISTAS DE SEGURIDAD: Son personas que trabajan en la seguridad de la aplicación y de la infraestructura. A veces no se dispone de un empleado por equipo dedicado de forma exclusiva a este rol. En esos casos es importante que todo el equipo reciba entrenamiento en seguridad.

ANALISTAS DE CI/CD: Son quienes mantienen los pipelines de integración y despliegue continuos. En aplicaciones simples es común que esta persona sea la misma que ocupa el rol de analista de infraestructura, pero en aplicaciones más complejas es necesario diferenciar roles.

ARQUITECTOS DE NUBE: Definen la arquitectura del entorno en la nube: la estructura que tendrán los servidores, cómo se interconectan y varios aspectos de seguridad relacionados. También definen quiénes tendrán acceso a los distintos entornos. En organizaciones pequeñas no hay una persona dedicada de forma exclusiva a esto y la función recae sobre el analista de infraestructura.

INGENIEROS DE CONFIABILIDAD DE SITIO (SRE): Son los encargados de diseñar y monitorear el sistema para minimizar las suspensiones de servicio y el tiempo de recuperación de los servicios. Trabaja tanto de forma proactiva como reactiva, respondiendo a incidentes, pero también intentando que no ocurran o vuelvan a ocurrir.

GERENTES DE ENTREGA: En algunos casos no es posible realizar despliegue continuo, por limitaciones del mercado o por la naturaleza del producto —por ejemplo, cada despliegue significa inevitablemente una suspensión temporal del servicio o cada cliente requiere una versión distinta del producto—. En estos casos, el gerente de entregas se ocupa de coordinar la entrega de nuevas versiones del producto a los clientes, llevar registro de qué cliente tiene qué versión del producto y orientar los esfuerzos del equipo hacia la satisfacción de los clientes.



Expectativas en el ecosistema DevOps

DevOps es una cultura que viene a transformar el ecosistema de desarrollo y de operación de software. Es importante que entendamos cómo funciona el equipo de DevOps y ver que principios y acciones necesitamos para desenvolvernos con éxito en esta cultura.

La cultura de DevOps está basada en la automatización, esto implica disminuir los procesos manuales aumentando la velocidad con la que se ejecutan y reduciendo así el esfuerzo necesario para llevarlos a cabo. Esta automatización es responsabilidad de cada una de las personas que trabajan en el equipo.

Una de las claves de la cultura de DevOps es la toma y el análisis de métricas. Estas métricas surgen de monitorear las aplicaciones, la infraestructura y los procesos. Algunas de las métricas más importantes pueden ser reducir tiempos promedio de recuperación de fallos, reducir tasa de errores, reducir tiempos promedios de puesta en producción de funcionalidades, incrementar tiempos promedios entre fallos.

Dentro de una organización la fricción es la fuerza que se opone al progreso de nuestro objetivo y en el caso de DevOps nuestro objetivo es poner software de calidad en las manos de las personas usuarias finales. Algunos elementos que introducen fricción son la necesidad de procesos manuales, la ocurrencia frecuente de errores o, las incompatibilidades entre distintos procesos. La automatización es una forma reducir la fricción, pero también se reduce la fricción simplificando los procesos al mínimo y asegurándonos de que todos sean compatibles entre sí. Por ejemplo, que los entornos de prueba y producción sean idénticos para permitir una transición sin errores de uno al otro.

Trabajar en un entorno DevOps significa que diferentes perfiles, que típicamente pertenecían a diferentes áreas, ahora trabajan en conjunto. Esta colaboración se da gracias a una comunicación fluida y constante entre las personas responsables de todas las actividades. Un equipo de DevOps tiene la capacidad técnica para ocuparse de todas las etapas de la solución que están construyendo, incluyendo el desarrollo, la gestión de calidad, la seguridad, el despliegue y la operación de la solución en un entorno de producción.

Aquí no existe la frase “eso no es mi trabajo”, la solución es responsabilidad de la totalidad del equipo. Al final de cuentas la cultura de DevOps es sostenida por todo el equipo de trabajo por eso es importante que todos podamos asumir la responsabilidad de hacer del equipo un verdadero equipo de DevOps.

Clase 3: Cierre de la Semana

Resumen de la Semana

Ejes de la materia

INFRAESTRUCTURA COMO CÓDIGO: Gestionar la infraestructura como un modelo escrito en código y versionado.

CI/CD

- Integración continua: Integrar a main frecuentemente.
- Despliegue continuo: Desplegar a producción frecuentemente.

MONITOREO

- Monitorear servidores: Disponibilidad, carga.
- Monitorear aplicaciones: Performance, experiencia de usuario.

¿Qué es DevOps?

DevOps es la combinación de filosofías, prácticas y herramientas que incrementan la velocidad a la que una organización entrega aplicaciones y servicios.

Perfiles de DevOps

- Desarrollador
- Analista de calidad
- Analista de infraestructura
- Analista de redes
- Analista de seguridad
- Analista de CI/CD
- Arquitecto de nube
- Ingeniero de confiabilidad de sitio (SRE)
- Gerente de entregas

Expectativas de DevOps

- Automatizar
- Medir y monitorear
- Reducir la fricción
- Aumentar la comunicación
- Ser dueños de la solución

Clase 4: Infraestructura como Código – La Disciplina

Módulo 2: Infraestructura como Código

Infraestructura como código: El Concepto

Empleamos el término Infraestructura como Código (IaC, por sus siglas en inglés) para referirnos a la gestión de la infraestructura a través de templates que tienen la capacidad de ser versionados.

De esta forma vamos a poder automatizar los procesos manuales que se requieren para lograr el objetivo final que buscamos.

Así como cada vez que ejecutamos el código de nuestra aplicación obtenemos el mismo resultado, lo mismo ocurre con la infraestructura: vamos a obtener el mismo resultado de infraestructura desde nuestra IaC. Este concepto es muy importante para implementar la metodología DevOps en nuestra vida profesional, ya que nos permitirá crear un ambiente mucho más rápido y seguro para nuestras aplicaciones.

La infraestructura como código nos permite gestionar la infraestructura de nuestras aplicaciones la cual se realiza a través de automatización, de la administración y el aprovisionamiento de la configuración del hardware físico o de los servicios de nuestro proveedor Cloud. De esta manera, vamos a poder automatizar todas las tareas manuales que se requieren para gestionar y preparar nuestra infraestructura tales como la creación de nuestro servidor para alojar nuestra aplicación, el aprovisionamiento de una base de datos de cualquier tipo de motor y la creación de un cluster para correr nuestros contenedores.

PROCEDIMIENTO DE LA INFRAESTRUCTURA COMO CÓDIGO

1. Analizar qué infraestructura necesitamos según los requisitos de la aplicación
2. Calcular si tenemos los recursos suficientes para ese requerimiento, definimos cuantas réplicas debe haber de nuestra infraestructura
3. Escribir nuestro template
4. Ejecutamos nuestro template en nuestra herramienta “infrastructure as Code” o se lo proveemos a otro equipo para que lo pueda ejecutar cuando lo necesite.
5. Recibiremos feedback de ese equipo o la herramienta de automatización nos avisará que terminó ejecutarse correctamente.

En la infraestructura como código se crean archivos de configuración que contienen las especificaciones que esta necesita, lo cual, facilita la edición y la distribución. Así mismo garantiza que siempre se prepare el mismo entorno. El control de versiones también es un aspecto importante de la infraestructura como código que debería aplicar a los archivos de configuración al igual que a cualquier otro archivo de código fuente de software. Además, la implementación de la infraestructura como código significa que puede dividirla en elementos modulares que se combinarán distintas maneras mediante la automatización.

Por lo general el procedimiento de la infraestructura como código implica los siguientes tres pasos:

1. Definir y describir las especificaciones de las infraestructuras para que pueda ejecutar la aplicación o base de datos.
2. Los archivos que se crean se envían un repositorio de código o de almacenamiento de archivos.
3. La plataforma de IaC toma todas las acciones necesarias para crear y configurar los recursos de infraestructuras.

Creación de una máquina virtual en Amazon Web Services (EC2)

1. Entrar a la consola de AWS
2. Elegir qué tipo de servicio
3. Elegir la región donde queremos trabajar
4. Elegir el tipo de instancia EC2 que queremos implementar
5. Seleccionar los recursos que le dan contexto a este servicio

6. Lanzar el servidor

La IaC es una práctica esencial de DevOps indispensable para un ciclo de vida de entrega de software a ritmo competitivo, permite a los equipos crear y versionar rápidamente la infraestructura de la misma manera en que versionan el código fuente y hacer un seguimiento de estas versiones para evitar inconsistencias que puedan conducir a graves problemas durante el despliegue.

Antes de la Infraestructura como código

¿Cómo iniciamos? ¿Cuáles son los primeros pasos en una implementación? ¿Qué mejoras nos ofrece la infraestructura como código?

Al momento de realizar los análisis para implementar una infraestructura para nuestra aplicación, lo *primero a definir es la arquitectura que necesitamos*: qué servidor es el adecuado o qué base de datos necesitamos. Una vez seleccionado el tipo de servidor, avanzamos en la configuración e instalación de aquello que nuestro sistema operativo requiere para estar operativo.

Paso a paso:

1. Configurar la conexión de la comunicación de los servidores hacia nuestras computadoras, hacia Internet, o bien hacia otros servidores.
2. Instalar las dependencias de la aplicación.
3. Implementar la aplicación o base de datos que va a tener ese servidor.

Mejoras y soluciones

Aunque este proceso se encuentre bien documentado, siempre vamos a encontrar obstáculos: el tiempo que nos ocupa la tarea, posibles incompatibilidades, o inconsistencias en la documentación. La metodología de infraestructura como código nos ofrece una solución a todo esto.

Este proceso que se realiza manualmente comienza a agilizarse ante el vertiginoso avance tecnológico, dando lugar a la automatización de la infraestructura. Como consecuencia, los ciclos de desarrollo poseen mejores tiempos de respuesta ante la mayor disponibilidad y flexibilidad de los ambientes de desarrollo de software.

La infraestructura como código irrumpe para aumentar la calidad y la eficiencia de los equipos de desarrollo.

Beneficios de la Infraestructura como código

1. **REDUCCIÓN DEL ERROR HUMANO:** Minimizamos el riesgo de equivocarnos cuando seguimos una serie de pasos. Mediante procedimientos claros y ordenados podemos evitar guardar una mala configuración o borrar algo que no debíamos. Esto va a aumentar la confianza que tengan en la infraestructura que brindemos.
2. **REPETIBILIDAD Y PREDICTIBILIDAD:** Cuando sabemos que el contexto de nuestra aplicación funciona, vamos a poder repetir la cantidad de pasos que sean necesarios y ser capaces de predecir el resultado, ya que siempre será el mismo. Esto nos da —como resultado— una infraestructura más testeable y estable.
3. **TIEMPOS Y REDUCCIÓN DE DESPERDICIOS:** El encargado de ejecutar nuestra infraestructura va a poder hacerlo en cuestión de minutos y sin necesidad de instalar algún componente extra. ¡Siempre vamos a poder activar el proceso para que haga lo suyo!

4. Al ejecutar las tareas más rápido, vamos a poder ayudar a otros equipos a que también trabajen en menos tiempo y de manera más eficiente.
5. **CONTROL DE VERSIONES:** Nuestra infraestructura se va a encontrar definida en archivos, por lo que vamos a poder versionar —al igual que el código fuente de nuestra aplicación— en templates o plantillas. ¡Saquémosle el jugo a los templates! Podemos utilizar parámetros para escribir nuestro código de la manera más genérica posible. Luego, al ejecutarlo, vamos a poder enviarle datos distintos en forma de parámetros para que nuestro código los reciba.
6. **REDUCCIÓN DE COSTOS:** Al automatizar procesos, podemos enfocarnos en otras tareas y mejorar lo ya hecho. Esto aporta a la flexibilidad de los equipos de infraestructura para abarcar más tareas.
7. **TESTEOS:** La infraestructura como código permite que los equipos de infraestructura puedan realizar pruebas de las aplicaciones en cualquier entorno (incluso producción) al principio del ciclo de desarrollo.
8. **ENTORNOS ESTABLES Y ESCALABLES:** Al evitar configuraciones manuales, la falta de dependencias y al obtener el estado final de infraestructura que necesitamos para nuestras aplicaciones, vamos a ofrecer entornos estables y escalables.
9. **ESTANDARIZACIÓN DE LA CONFIGURACIÓN:** Estandarizar las configuraciones y el despliegue de la infraestructura nos permite evitar cualquier problema de incompatibilidad con nuestra infraestructura y que las se ejecuten con el mejor rendimiento posible.
10. **DOCUMENTACIÓN:** Al aportar a la documentación de los procesos internos de nuestros equipos vamos a mejorar tiempos y costos. Como ya vimos, podemos versionar nuestras automatizaciones. Esta característica nos permite que cada cambio se encuentre documentado, registrado por usuario y con una vuelta atrás (rollback) rápida si encontramos errores en los despliegues, al igual que el código fuente
11. **MÁS RAPIDEZ SIN DESCUIDAR LA SEGURIDAD:** Al momento de mejorar nuestra infraestructura, nunca hay que dejar de pensar en la seguridad que la compone. Al momento de estandarizar la ejecución de la infraestructura, también podemos estandarizar los grupos de seguridad con los permisos mínimos, pero necesarios para que todos los equipos puedan trabajar y evitar tareas manuales por parte de los equipos de seguridad.

Paradigmas para la infraestructura como código

Existen dos paradigmas de programación aplicados a la Infraestructura como código. Al escribir nuestro IaC podemos optar por el paradigma imperativo, que nos posibilita controlar el flujo de trabajo de nuestro código, o bien enfocarnos en el resultado final y en el cambio de nuestra infraestructura, el paradigma declarativo. Es el "cómo" versus el "qué".

Paradigma Imperativo

Utilizamos el paradigma imperativo cuando al escribir nuestro código nos enfocamos en cómo se va a ejecutar a través de diversas operaciones y el flujo de trabajo que va a realizar. Vamos a definir variables constantes y definir decisiones. Las más conocidas son:

- IF
- ELSE
- ELIF (en otros lenguajes se conoce como ELSE IF)
- FOR y FOREACH
- WHILE y DO WHILE
- SWITCH (no existe en todos los lenguajes)
- Manejo de errores con excepciones (TRY/CATCH/FINALLY)

Paradigma imperativo



Consideramos que la utilización de este tipo de controles es imperativo porque estamos controlando de manera explícita nuestro flujo de trabajo dentro del código y qué decisiones se ejecutan según las condiciones que definamos.

Se utilizan estructuras de control o loops para controlar el proceso de nuestro código.

Paradigma Declarativo

Paradigma declarativo



Al utilizar el paradigma declarativo, vamos a trabajar sobre la lógica de **qué se va a ejecutar**, sin indicar los detalles de cómo lo va a hacer. Al utilizar este método, nuestro código va a estar compuesto por un **conjunto de funciones que van a realizar la tarea que definamos**. Es muy importante tener test automatizados para probar nuestro código. Al ejecutarlos y ver los resultados vamos a tener la posibilidad de identificar errores en la lógica de nuestro código. Podemos decir que el enfoque declarativo define el estado final de nuestra infraestructura.

Se utilizan métodos para el proceso de nuestro código y luego se ejecutan pruebas (o tests).

El principio de Idempotencia

La propiedad de obtener siempre un mismo resultado sin importar las veces que ejecutemos una operación es lo que se denomina principio de idempotencia. El principio de idempotencia, dentro del ámbito de infraestructura como código, es la propiedad de poder ejecutar nuestro código las veces que sea necesario **siempre obteniendo el mismo resultado**. Para que nuestra infraestructura siempre sea idempotente utilizaremos herramientas para ese fin como pueden ser, Ansible, Terraform o CloudFormation.

Si separamos la infraestructura como código en etapa estás serían las siguientes:

1. **ORIGEN:** Es el archivo de configuración que en general es un archivo .json o .yaml
2. **PROCESO:** Son las operaciones que realizan las herramientas de IaC en base al archivo origen
3. **DESTINO:** Es el estado final de la infraestructura tal como lo necesitamos.

Al aplicar el principio de idempotencia en estas tres etapas vamos a tener distintos beneficios como, por ejemplo, la capacidad de que si necesitamos modificar algo de nuestra infraestructura sólo debemos modificar el archivo de origen luego, ejecutaremos el proceso de ese archivo de

configuración con la herramienta que estemos utilizando y obtendremos el estado final según las modificaciones que apliquemos.

Es fácilmente documentable ya que no tenemos que revisar nuestra infraestructura para revisar cómo está compuesta o documentar todo después de hacerlo. Tenemos nuestro archivo de configuración con toda la información y, si es necesario, sólo hay que agregar documentación complementaria.

Se pueden aplicar prácticas de desarrollo de software al área de infraestructura algo, que hasta hace poco era muy lejano, como por ejemplo versionar los archivos de configuración para volver a una versión anterior sin trabajo extra y es fácilmente compartible con el resto del equipo.

Todo esto nos permite que el proceso sea totalmente automatizable. Conocer el principio idempotencia y los beneficios en su aplicación es fundamental ya que esta es la base teórica de la automatización que permite la IaC.

Ecosistema de herramientas IaC

Al automatizar nuestra infraestructura, es probable que utilicemos distintos proveedores o que usemos una parte cloud y otra parte on-premise (un datacenter propio). Existen herramientas que poseen su propia sintaxis (en general, JSON y YAML) para poder administrar la infraestructura en múltiples proveedores o en uno solo, pero de una manera más eficiente.

¿Por qué hacemos estas distinciones? ¿Porque podemos elegir para nuestra infraestructura la herramienta que nos brinda la mayor eficiencia posible!

Terraform

Terraform es un software de código libre desarrollado por HashiCorp. Es una herramienta *declarativa* de aprovisionamiento y orquestación de infraestructura que permite automatizar el aprovisionamiento de todos los aspectos de la infraestructura, tanto para la nube como la infraestructura on-premise (en los mismos datacenter). Tiene algunas características interesantes, como comprobar el estado de la infraestructura antes de aplicar los cambios. Es la herramienta más popular porque es compatible con todos los proveedores de nube sin realizar modificaciones en nuestros templates.

AWS CloudFormation

AWS CloudFormation es la solución nativa de AWS para aprovisionar recursos en esta nube. En este caso se pueden definir templates en formato JSON o YAML. Se pueden utilizar para crear, actualizar y eliminar recursos las veces que sea necesario. Una ventaja de CloudFormation es que, al ser un servicio propio de Amazon, tiene una integración completa con los demás servicios de AWS, por lo que es nuestra mejor opción si solo utilizamos este proveedor de nube.

Azure Resource Manager

ARM es la herramienta nativa en Azure para implementar infraestructura como código, Azure Resource Manager (ARM Templates). Estas plantillas llevan una sintaxis declarativa en formato JSON, que nos permiten definir los recursos y las propiedades que conforman la infraestructura.

Google Cloud Deployment Manager

Google Cloud Deployment Manager es la herramienta IaC para la plataforma Google Cloud —lo mismo que CloudFormation es para AWS—. Con esta herramienta los usuarios de Google pueden administrar fácilmente mediante archivos de configuración YAML.

Ansible

Ansible es una herramienta de automatización de infraestructuras creada por Red Hat. Ansible modela nuestra infraestructura describiendo cómo se relacionan sus componentes y el sistema entre sí, en lugar de gestionar los sistemas de forma independiente.

Clase 5: Infraestructura como Código en AWS – CloudFormation

CloudFormation

CloudFormation es una herramienta nativa de Amazon Web Services (más conocido como AWS). Pero... ¿por qué es una herramienta tan popular? Porque nos brinda la posibilidad de implementar prácticas de infraestructura como código (IaC) de forma nativa dentro de AWS.

Cloud Formation (CF) nos brinda la posibilidad implementar prácticas de infraestructura como código de forma nativa dentro de AWS, esta herramienta se va a encargar de crear y configurar la infraestructura que definimos previamente en una planilla o template con los requisitos que necesitamos. Esta herramienta nos ofrece algunas ventajas, como crear repositorios con nuestros templates para que sean accesibles o que se puedan realizar entregas rápidas de los recursos de infraestructura.

Lanzada en el 2018, esta herramienta permite aprovisionar la infraestructura necesaria en base a dichos **templates** que son **reutilizables** y **parametrizables** escritos en archivos con formato .json o .yaml. Estos archivos se pueden guardar con cualquier extensión, como: .json, .txt, .yaml, .template y son estos los que utiliza CF para crear los recursos de infraestructura.

Por ejemplo, en una plantilla se puede definir la creación de una instancia Amazon EC2 con todas sus características. Dentro de las particularidades más notables de CF se encuentran:

- Templates parametrizables que pueden levantar distintos servicios únicamente cambiando los parámetros, esto nos permite que sean reutilizables.
- Automatización, ya que se pueden utilizar los templates dentro de pipelines.
- Permite el uso de rollbacks que nos da automáticamente una vuelta para atrás, destruyendo lo que se haya producido en el caso de que falle alguno de los pasos de la creación de recursos de infraestructura.
- El uso de Cloud Formation es totalmente gratuito lo cual hace que el precio sea otra cosa interesante.

CF es una herramienta integral que nos permite administrar nuestra infraestructura en un **entorno controlado** y **predecible**, reduciendo los tiempos de despliegue y, agilizando así, la entrega de recursos.

¿Cómo usarla?

Para introducirnos en el funcionamiento de la herramienta, vamos a tener en cuenta tres conceptos importantes:

- **PLANTILLAS O TEMPLATES:** Es la definición de la arquitectura de nuestra infraestructura y que, gracias a CF, vamos a poder implementar. Es un archivo de texto con formato JSON (JavaScript Object Notation) que describe los recursos que queremos crear junto a sus propiedades.
- **PILAS:** es una unidad que genera CloudFormation para la creación ordenada de los recursos.

- **CAMBIO:** es un resumen de los cambios que se proponen para anticiparnos al resultado final.

Para el ejemplo, nuestro template va a contener una instancia EC2 para alojar nuestro web server y la aplicación WordPress, una base de datos en el servicio de Amazon RDS, un balanceador de carga, una VPC y subnets con dos zonas de disponibilidad distinta, servicios de grupos de seguridad.

Lo primero que vamos a crear es la descripción para que quienes lo reciban sepan a grandes rasgos qué componentes posee seguimos con los parámetros, estos reciben nombres lógicos como ID que presentan recursos.

Los **parámetros** se componen de un tipo lógico y de tres componentes asociados a este tipo de recurso. Tomemos VPC como ejemplo:

```
1  "VpcId" : {
2    "Type" :
3    "AWS::EC2::VPC::Id",
4    "Description" : "VpcId
5    of your existing Virtual
6    Private Cloud (VPC)",
7    "ConstraintDescription"
8    : "must be the VPC Id of an
9    existing Virtual Private Cloud."
10   },
11
```

VpcId es el nombre lógico para definir una virtual Private Cloud donde vamos a encontrar tres componentes asociados:

- **Type** que es el tipo de recurso dentro de AWS, en este caso, una VPC para una instancia EC2.
- **Description** que es la documentación del tipo de recurso a crear
- **ConstraintDescription** qué es la condición necesaria para que se pueda utilizar este recurso.

Continuamos con **Mappings**, una clase de mapeo de cadenas literales que puede recibir como tipo de dato un string o un list con el formato clave valor. En el caso de nuestro ejemplo vamos a mapear la creación de distintas instancias EC2 a un tipo de ami que es la HVM64.

```
1  "Mappings" : {
2    "AWSInstanceType2Arch" : {
3      "t1.micro" : { "Arch"
4      : "HVM64" },
5      "t2.nano" : { "Arch"
6      : "HVM64" },
7
```

Luego crearemos los distintos recursos que necesitamos como el balanceador de carga para tener distintas réplicas de nuestra aplicación y de la base de datos. En este ejemplo crearemos el recurso ELB (ElasticLoadBalancing) haciendo referencia a la subnets de nuestra red privada.

```

1  "Resources" : {
2
3      "ApplicationLoadBalancer"
4  : {
5      "Type" :
6      "AWS::ElasticLoadBalancingV2:
7      :LoadBalancer",
8      "Properties" : {
9          "Subnets" : { "Ref" :
10         "Subnets"}
11     }
12 },

```

En **resources** vamos a poder crear nuestros grupos de seguridad para resguardar la aplicación y la base de datos.

```

1  "WebServerSecurityGroup" :
2  {
3      "Type" :
4      "AWS::EC2::SecurityGroup",
5
6

```

Para finalizar los templates se usan **Outputs** con el objetivo de declarar cuál va a ser el mensaje final que va a visualizar el ejecutor del template. En este ejemplo va a visualizar la ruta expuesta a internet de nuestra aplicación. Como está automatizado genera la ruta a través de los nombres de los distintos recursos que fue creando.

```

1  "Outputs" : {
2      "WebsiteURL" : {
3          "Value" : { "Fn::Join" :
4          ["", ["http://", {
5              "Fn::GetAtt" : [
6                  "ApplicationLoadBalancer",
7                  "DNSName" ]}], "/wordpress"
8          ]}],
9          "Description" :
10         "WordPress Website"
11     }
12 }
13

```

Datos Curiosos

- Con CloudFormation podemos delegar las tareas y controlar los accesos con IAM desde AWS
- Actualmente el 37% del uso de la nube es con AWS
- Podemos versionar nuestros templates utilizando el servicio S3
- En el caso de que CF detecte un error en el proceso, el rollback es automático.
- Forma parte de la certificación “AWS Certified Solutions Architect-Associate” para oficializar nuestros conocimientos en la nube.

¿Dónde la usamos?

CloudFormation se puede utilizar en distintos ámbitos:

- Podemos hacerlo por línea de comando desde nuestros equipos.
- En scripts (como PowerShell).
- En pipelines, como parte de un conjunto de tareas automatizadas y encadenadas entre sí, formando una tubería con un inicio y un fin.

¿Quién la usa?

Las plantillas de CloudFormation posibilitan que los analistas de infraestructura deleguen tareas de creación de recursos a otras áreas, a través del control del mismo código de las automatizaciones. Pero ¿cualquier tipo de usuario puede ejecutarlas?

Consideraciones que hay que tener en cuenta al momento de autorizar usuarios que se encuentran por fuera del área de infraestructura.

¿QUÉ TIPO DE ANALISTA DE INFRAESTRUCTURA SOS SI USAS CF?

Analista de infraestructura de cualquier seniority para ejecutar CF desde cualquier entorno

Analista de infraestructura Ssr o Sr para escribir templates propios y ejecutarlos desde cualquier entorno

Desarrollador de cualquier seniority para ejecutar los templates dados desde la consola web

Líder técnico o desarrollador avanzado para ejecutar un template de CF (con los permisos necesarios de AWS) desde AWS CLI o la consola web.

Clase 6: Cierre de la Semana

Resumen de la Semana

Durante esta semana nos introducimos en el concepto de infraestructura como código y cuáles son las tres herramientas que tenemos que conocer sí o sí, ya que son las más importantes del mercado. También pusimos manos a la obra con la primera de ellas, CloudFormation.

CloudFormation es la herramienta de IaC nativa más popular en el mercado.

Para el mercado de trabajo, las certificaciones de AWS van desde la certificación inicial hasta la ruta de aprendizaje de arquitecto cloud.

¿QUÉ PROBLEMAS RESOLVEMOS?

- Resolvemos gran parte de las tareas de infraestructura de nuestros recursos en AWS.
- Podemos delegar el uso de los templates a través de los roles de permisos IAM.

VERSIONAMIENTO: Conocemos cómo versionar nuestros templates a través de los buckets S3 de AWS.

REUTILIZACIÓN: Podemos elegir un template dado por AWS o crear una versión propia para cubrir nuestros requisitos.

ANALISTAS DE INFRAESTRUCTURA: Evitamos repetir tareas rutinarias y entregamos la infraestructura mucho más rápido.

DOCUMENTACIÓN: Aportamos a la documentación del área de infraestructura.

Ansible

Ansible es un proyecto comunitario open source diseñado para ayudar a las organizaciones a automatizar el aprovisionamiento de infraestructura, la gestión de configuración y el despliegue de aplicaciones.

Ansible fue desarrollada por Michael DeHann, el co-autor de Cobbler (aplicación para aprovisionar servidores) y co-autor del framework Fedora Unified Network Controller para administración remota. Ansible se distingue de Chef y Puppet, quienes dominan el mercado en este momento, por ser más simple aprender y **Agentless**, esto implica que no necesita instalar un agente en un nodo para administrarlo, sino que utiliza SSH en Linux o Unix o, WinRM y PowerShell en Windows para ejecutar comandos dentro de los nodos.

Con Ansible se crean archivos de configuración llamados playbooks, escritos en YAML, que se utilizan para especificar el estado requerido de la infraestructura. Al ejecutarlos, Ansible se ocupa de aprovisionar la infraestructura necesaria para alcanzar el estado descrito. Esto quiere decir que se puede, por ejemplo, crear una máquina virtual en el proveedor de infraestructura -como una instancia EC2 dentro de AWS- aplicando metodologías de infraestructura como código.

Los playbooks se pueden ejecutar localmente en la computadora del analista de infraestructura o en un servidor dedicado a la ejecución de estos.

Ansible no nace originalmente como una herramienta infraestructura como código es la propia comunidad que, ante la necesidad de gestionar la infraestructura en distintos proveedores de Cloud de forma simple y repetible, desarrolla los módulos necesarios para implementar la guía C.

Los objetivos de esta herramienta son:

1. **MINIMALISMO POR NATURALEZA:** Un sistema de gestión no debería imponer dependencias adicionales al entorno.
2. **CONSISTENCIA:** Con Ansible uno puede crear entornos consistentes
3. **SEGURIDAD:** Ansible no instala agentes en nodos, solo requiere que un nodo tenga instalado OpenSSH y Python.
4. **CONFIABILIDAD:** Un playbook en Ansible debería ser indepotente para evitar efectos inesperados en los sistemas a gestionar.
5. **MÍNIMO APRENDIZAJE REQUERIDO:** Los playbooks utilizan un lenguaje fácil y descriptivo basado en YAML y templates de Jinja.

Un dato relevante a mencionar es que vamos a poder parametrizar templates de Ansible con Jinja, una tecnología basada en Python que nos va a permitir crear templates de Script en powershell o base scripting, posibilitando el uso de condiciones propias de programación como condicionales y bucles.

Ansible es una herramienta muy potente que permite gestionar configuraciones, desplegar aplicaciones y manejar infraestructura como código.

¿Cómo usarla?

Ansible es una herramienta que permite gestionar las configuraciones de tu infraestructura. Sus principales ventajas al momento de usarla son:

- No necesita instalación de agentes.
- Su configuración es de fácil lectura.
- Es muy flexible (usa APIs y plugins).
- Es fácil de usar por basarse en YAML.

Veamos cómo podemos ejecutar Ansible dentro de nuestra computadora e impactar los resultados en nuestra cuenta de AWS.

Lo primero que debemos hacer es ejecutar el comando [Ansible-playbook](#) seguido del archivo .yaml que tiene nuestro código. Este tipo de archivos es legible y, para el caso de Ansible, su extensión es menor ya que la mayor parte del código son invocaciones a módulos publicados para toda la comunidad.

Dentro del archivo yaml la estructura nuestro código es la siguiente:

- Hosts. Es donde vamos a ejecutar el Playbook, se declara porque puede ejecutarse remotamente (como en nuestro caso) así que solo debemos escribir localhost.
- Tasks. Es una palabra reservada para indicar que vamos a ejecutar tareas.
- Name. Aquí es donde vamos a indicar lo que vamos a ejecutar, es muy útil porque nos obliga a documentar.
- Nombre módulo. Indicamos el nombre del módulo, los más comunes para AWS son: ec2, ec2_ami, ec2_elb, ec2_tag o ecs_ecr.

```

1 ---
2 - hosts: localhost
3   tasks:
4     - name: ¿Que estamos haciendo?
5       Comentemos brevemente
6         nombre módulo:
7
8

```

Para nuestro ejemplo el playbook que queremos ejecutar quedaría de la siguiente manera: empezamos indicando que el host es localhost, es decir, que lo implementamos en nuestra PC. Seguimos con las tareas, tasks, luego con el nombre de nuestra primera tarea, el contenido es con el objetivo de documentar que es lo que estamos haciendo. Después del name tenemos que indicar que módulo vamos usar, siempre se ejecuta un módulo. Finalmente vamos a ingresar los parámetros para el módulo de ec2: la región de AWS donde trabajaremos, el tipo de instancia, el id de imagen ami -que referencia a un template interno que tiene AWS-, para completarlo tenemos que averiguar cual nos corresponde de acuerdo al tipo de instancia y de región y, para eso, hay que chequearlo en AWS console.

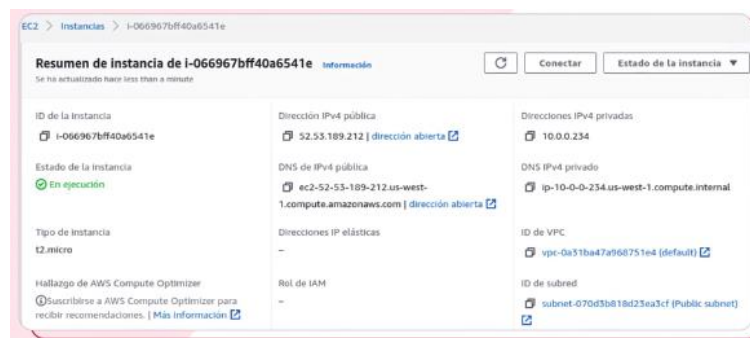
Wait es recomendable con el valor yes, para esperar a que termine toda la ejecución y luego nos de el ok. Wait_timeout es el tiempo en segundos que esperaremos si no responde AWS. Volumes es un parámetro que utilizaremos si le asignamos un disco a nuestra instancia. Vpc_subnet_id es un parámetro que debe estar creado previamente. Lo podemos averiguar dentro del servicio vpc en la opción subnet. Y por último tenemos el parámetro assign_public_ip que servirá para asignarle una IP pública a una instancia.


```

1 ---
2 - hosts: localhost
3   tasks:
4     - name: Creamos nuestro servidor
5       ec2:
6         region: us-west-1
7         instance_type: t2.micro
8         image: ami-0ed05376b59b90e46
9         wait: yes
10        wait_timeout: 500
11        volumes:
12          - device_name: /dev/xvda
13            volume_type: gp2
14            volume_size: 8
15            vpc_subnet_id: subnet-070d3b818d23ea3cf
16            assign_public_ip: yes
17

```

Nuestra implementación en la consola de AWS quedaría de la siguiente manera:



Donde vamos a poder revisar que estos parámetros coinciden con nuestro playbook de Ansible.



El uso de Ansible como herramienta de IaC es más fácil para aquellas personas familiarizadas con Linux ya que hay que ejecutar líneas de comando.

Como gran ventaja, el código Ansible es de muy poca extensión, ya que la habilidad del analista es saber qué módulos tiene que utilizar y conectarse a los recursos necesarios para que impacte lo que está declarado en el código.

Módulos de Ansible¹

Podemos extender el uso de Ansible para automatizar más servicios de AWS.

¹ <https://docs.ansible.com/ansible/latest/collections/community/aws/index.html#>

El corazón de Ansible es la ejecución de archivos de playbooks con las instrucciones necesarias para lograr la infraestructura que queremos, para no tener que escribir la lógica repetidamente para cada caso de uso, y que nuestro código utilice el paradigma declarativo. Para todo eso tenemos a nuestra disposición módulos con problemas comunes ya resueltos.

¿Sabías qué?

- Ansible es una herramienta mantenida por Red Hat, una empresa multinacional dedicada a la distribución de software de código abierto. Su producto más conocido es el sistema operativo Red Hat Enterprise Linux.
- Ansible tiene centenares de módulos que resuelven problemas comunes y son muy útiles. Por ejemplo, la automatización de infraestructura en la nube. La lista de módulos se encuentra en su documentación.
- Existe Ansible Galaxy, una comunidad que permite la distribución de módulos hechos por los usuarios.
- Usar Jinja2 dentro de Ansible nos permite realizar templates de scripts de powershell y bash scripting usando técnicas de programación existentes en Python
- Podemos ejecutar “roles” dentro de nuestros playbooks para crear mini módulos con código que utilizamos habitualmente.

¿Dónde la usamos?

La flexibilidad de Ansible permite hacerlo en diferentes espacios:

- En tu computadora.
- En un servidor que pueda ser usado para ejecutar Ansible.
- En el proyecto de código abierto AWX que podés instalar y usar para administrar tus playbooks.

Esta última opción te brinda una gran ventaja: podés administrar tus automatizaciones y delegar a áreas operativas sin conocimientos en Ansible para que puedan ejecutar tus playbooks según sea necesario. ¡Únicamente te va a preocupar seguir automatizando!

¿Quién la usa?

Ansible es una herramienta muy flexible que nos permite gestionar nuestra infraestructura. Pero, ¿qué conocimientos básicos necesitamos para utilizar esta herramienta?

¿QUÉ TIPO DE ANALISTA DE INFRAESTRUCTURA SOS SI USAS ANSIBLE?

Administras servidores. Ansible surge primero para gestionar configuraciones en servidores y luego se utiliza para gestionar infraestructura en la nube. Si la elegís como herramienta, es probable que también te ocupes mucho de gestionar los servidores.

Utilizas Ansible para todo. Al ser una herramienta muy versátil, si la sabes manejar bien, puedes sacarle el jugo a todas sus posibilidades.

Utilizas YAML. Ansible utiliza YAML como lenguaje para sus playbooks.

Te gusta el software open source. El gran motivo por el que Ansible es lo que es hoy en día es por su comunidad open source. Si elegís Ansible, probablemente te guste el software de código abierto y las comunidades que se forman alrededor de estos.

Pensas de forma descriptiva. Te gusta describir el estado final de la infraestructura y que tu herramienta se ocupe de los pasos necesarios para llegar ahí.

Clase 8: Infraestructura como código – Terraform

Terraform

Terraform es una herramienta de código abierto creada por HashiCorp en 2017 y programada con el lenguaje Golang más conocido como Go. Para el desarrollo de sus archivos de configuración utiliza el paradigma de **programación declarativo**, gracias a su lenguaje nativo llamado HCL (HashiCorp Configuration Language), con el objetivo de describir el estado final de la infraestructura que deseamos crear.

Con Terraform vamos a poder elaborar un plan para alcanzar este estado final y ejecutarlo para suministrar la infraestructura. Debido a la sencilla sintaxis de sus archivos de configuración, es la **herramienta más popular de IaC** en múltiples nubes y también de infraestructura local más conocida como On Premise por lo que, para el agitado ambiente tecnológico que cambia constantemente y ante la necesidad de migrar de un tipo de infraestructura a otra, para lograr la mejor performance para los usuarios, Terraform se convierte en la mejor solución para brindar los ambientes de desarrollo y producción.

Dentro de los beneficios de Terraform podemos identificar:

- La **VELOCIDAD** que aporta para soluciones rápidas por sus automatizaciones y código declarativo.
- La **CONFIANZA** al escribir el código pensando en el estado final de nuestra infraestructura, con Terraform se reduce drásticamente la posibilidad de equivocarnos en grandes volúmenes de recursos para crear.
- Es fácilmente **ADAPTABLE** si nos vemos en la necesidad de migrar de proveedor de infraestructura Cloud. La adaptación del código es muy sencilla con Terraform, modificando pequeñas cosas y sin tener que escribir todo el código desde cero.
- Además, nos ofrece una gran **ESCALABILIDAD** si nuestra infraestructura, en entornos de prueba, es aprobada por todos los equipos de desarrollo. Es muy rápido implementar esos o más recursos en ambientes productivos.
- Su **CÓDIGO ES ABIERTO**, al ser una herramienta Open source nos garantiza una gran comunidad que la mantiene actualizada como herramienta
- Su **INDEPENDENCIA**, como se mencionó, Terraform es independiente de cualquier tipo de proveedor de infraestructura.

Al igual que Ansible, Terraform posee **módulos** que son pequeños recursos con código reutilizable para problemas comunes donde la comunicación es entre módulos, es decir, cada módulo está escrito sobre otro módulo que extiende su uso llamados *módulos hijos*. Estos módulos son **reutilizables** ya que se pueden llamar desde distintos archivos y configuraciones para realizar la misma acción.

Terraform Se puede instalar en nuestras computadoras como los tres proveedores de nube más populares: AWS, Azure, GCP. También ofrece Terraform Cloud que es la venta de una plataforma para ejecutar Terraform.

Terraform es la herramienta más popular para implementar IaC por la gran cantidad de ventajas que nos ofrece siendo la **más destacable** su **independencia** de proveedores de infraestructura.

¿Cómo usarla?²

Recorramos juntos un archivo de configuración estándar de Terraform para crear una instancia con todos los recursos que necesita para funcionar completamente automatizado.

Hagamos foco en:

- La sintaxis de los archivos de configuración.
- De qué manera se ejecutan.
- La flexibilidad del código.
- El tipo de extensión de los archivos.

Crear una VM con Terraform

Como se mencionó anteriormente, Terraform es una herramienta Open Source desarrollada por HashiCorp que se ejecuta a través de escritos en lenguaje HCL y la extensión del **archivo es .tf**, por ejemplo, un archivo para crear una máquina virtual EC2 (que es una instancia de AWS) probablemente se llame ese EC2.tf y dentro va a contener el código para crear dicho recurso.

El tipo de desarrollo en Terraform es **declarativo** porque se utilizan módulos y sólo nos vamos a **enfocar** en el **estado final** que deseamos para nuestra estructura. En el caso de nuestro ejemplo, el código completo se verá de la siguiente manera:

```
1 provider "aws" {
2   shared_credentials_file = "~/aws/credentials"
3   region = "us-west-1"
4 }
5
6 module "vpc" {
7   source = "terraform-aws-modules/vpc/aws"
8
9   name = "mi-vpc"
10  cidr = "10.0.0.0/16"
11
12  azs          = ["us-west-1b", "us-west-1c"]
13  private_subnets = ["10.0.1.0/24", "10.0.2.0/24"]
14  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24"]
15
16  enable_nat_gateway = true
17 }
18
19 module "ec2_cluster" {
20   source = "terraform-aws-modules/ec2-instance/aws"
21   version = "~> 2.0"
22
23   name           = "digitalhouse"
24   instance_count = 1
25   ami            = "ami-0e8637b59b90e46"
26   instance_type  = "t2.micro"
27   vpc_security_group_ids = [module.ssh_security_group.this_security_group_id]
28   subnet_ids     = module.vpc.private_subnets
29 }
30
31 module "ssh_security_group" {
32   source = "terraform-aws-modules/security-group/aws//modules/ssh"
33   version = "~> 3.0"
34
35   name = "ssh-server"
36   description = "Grupo de seguridad"
37   vpc_id = module.vpc.vpc_id
38   ingress_cidr_blocks = ["10.10.0.0/16"]
39 }
40 }
```

En el primer bloque indicamos que proveedor de infraestructura vamos a utilizar, en este caso, es AWS. Dentro de ese **provider**, señalamos donde están las credenciales de nuestra cuenta y qué región de AWS vamos a usar.

```
1 provider "aws" {
2   shared_credentials_file = "~/aws/credentials"
3   region = "us-west-1"
4 }
5
```

En el segundo bloque, vamos a ejecutar el módulo **vpc**. El mismo ya contiene la lógica de cómo crear una red, sólo le tenemos que pasar lo que es particular de nuestra cuenta. En **source** indicamos la

² <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>

dirección del módulo, en *name* escribimos el nombre que va a tener nuestra red vpc y que direcciones IP va a utilizar. También elegimos entre dos zonas de disponibilidad, *azs*, luego señalamos las IPs privadas y las IPs públicas que vamos utilizar y el último parámetro habilita que nuestro servidor está expuesto en internet.

```
1 module "vpc" {
2   source = "terraform-aws-modules/vpc/aws"
3
4   name = "mi-vpc"
5   cidr = "10.0.0.0/16"
6
7   azs          = ["us-west-1b", "us-west-1c"]
8   private_subnets = ["10.0.1.0/24", "10.0.2.0/24"]
9   public_subnets  = ["10.0.101.0/24", "10.0.102.0/24"]
10
11   enable_nat_gateway = true
12 }
```

Seguimos con el segundo módulo al cual le ponemos el nombre EC2, ya que vamos a crear ese tipo de instancia nuestra cuenta. Lo primero es indicar el módulo, luego su versión. Después el nombre de nuestro servidor y la cantidad. Definimos el ami, que es un ID único para el tipo de EC2 que queremos crear, este depende de la región y el tipo de instancia.

En el tipo de instancia para crear usamos *T2.micro* porque es de uso gratuito y no genera gasto en la cuenta y, por último, hacemos referencia al grupo de seguridad creado arriba.

```
1 module "ec2" {
2   source      = "terraform-aws-modules/ec2-instance/aws"
3   version    = "~> 2.0"
4
5   name        = "digitalhouse"
6   instance_count = 1
7
8   ami          = "ami-0ed05376d59b90e46"
9   instance_type = "t2.micro"
10  vpc_security_group_ids = [module.ssh_security_group.this_security_group_id]
11  subnet_ids = module.vpc.private_subnets
12 }
```

Luego vamos a configurar otro módulo para acceder al servidor. SSH es el protocolo utilizado para conectarse remotamente a servidores con sistemas operativos basados en Linux. Primero indicamos que módulo de Terraform vamos a usar, luego la versión del módulo. Señalamos que el recurso a crear se llama *ssh-server*, le agregamos la descripción a modo de documentación, hacemos referencia a que va a usar la VPC creada arriba y, al final, escribimos el rango de IPs que va a utilizar nuestro servidor. *Tiene que coincidir con el cidr nuestra vpc.*

```
1 module "ssh_security_group" {
2   source = "terraform-aws-modules/security-group/aws//modules/ssh"
3   version = "~> 3.0"
4
5   name = "ssh-server"
6   description = "Grupo de seguridad"
7   vpc_id = module.vpc.vpc_id
8
9   ingress_cidr_blocks = ["10.10.0.0/16"]
10 }
11
```

Una vez que tenemos todo nuestro código, en nuestro caso en un archivo que se llama *mi_ec2.tf*, tenemos que **iniciar Terraform** con el comando **terraform init**. Al ejecutar este comando, no va a descargar todos los módulos que declaramos.

```

[awscli@aws:~]$ terraform init
Initializing modules...
Downloading terraform modules/oci2-instance/aws 2.18.0 for oci2...
oci2 is terraform/modules/oci2
Downloading terraform modules/security-group/aws 3.18.0 for oci2-security-group...
oci2-security-group is terraform/modules/oci2-security-group/modules/oci2-security-group
oci2-security-group is terraform/modules/oci2-security-group
Downloading terraform modules/oci2-vg/aws 3.18.0 for oci2-vg...
oci2-vg is terraform/modules/oci2-vg
Initializing the backend...
Initializing provider plugins...
- Finding hashicorp/oci2 versions matching "~> 2.42.0, < 3.15.0, < 3.24.0"...
- Installing hashicorp/oci2 v2.42.0.
- Installed hashicorp/oci2 v2.42.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee it gets the same selection by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
will now work.

If you ever set or change modules or backend configuration for Terraform,
run "terraform init" to reinitialize your configuration. If you forget, other
commands will detect if and remind you to do so if necessary.
[awscli@aws:~]$

```

Ahora podemos **aplicar** los **cambios** en nuestra cuenta, se pueden hacer con **Terraform apply** habilitándonos un largo detalle de lo que va a crear y al final nos hace un resumen de la cantidad total, que en este caso serían 125 recursos. Para finalizar ingresamos yes para confirmar que queremos realizar esta acción y nos va a mostrar la creación de los recursos en tiempo real.

En nuestra cuenta de aws ya podemos ver que hay una máquina virtual nueva, llamada como indicamos en el código, y que está en estado de comprobación, inicializando.

Cuando terminamos la creación en Terraform nos va a mostrar un resumen que coincide con el que mostró al ingresar yes y si echamos un vistazo a nuestra cuenta AWS vamos a ver también que todo está funcionando correctamente.

Nombre	ID de la instancia	Estado de la L...	Tipo de inst...	Comprobación...	Estado de la...
digitalhouse	i-0b11a55f0b7c58b350	En ejecución	t2.micro	Iniciando	Sin alarmas

Para destruir nuestra infraestructura utilizamos el comando **terraform apply -destroy**

¿Sabías que?

- Terraform ofrece una plataforma para que podamos ejecutar nuestras automatizaciones. Se llama Terraform Cloud.
- Podemos tomar un examen para validar nuestros conocimientos en la herramienta. Este tipo de certificaciones oficiales tiene mucho valor en el mercado laboral
- Nos permite automatizar la implementación de infraestructuras en múltiples proveedores de nube.
- HashiCorp promueve la participación de la comunidad. Ofrece cursos y charlas online con novedades y actualizaciones.
- Al igual que en Ansible, podemos crear nuestros propios módulos con las automatizaciones que usamos habitualmente, pero en distintos escenarios.

¿Quién la usa?

Terraform es una herramienta que se caracteriza por el apoyo de su comunidad. Además, es una herramienta adaptable, ya que se puede utilizar en cualquier proveedor de infraestructura que utilicemos.

¿QUÉ TIPO DE ANALISTA DE INFRAESTRUCTURA SOS SI USAS TERRAFORM?

Amante de los resultados. Terraform se caracteriza por ser una herramienta que se enfoca en el estado final de la infraestructura sacándole todo el jugo a los módulos.

Te gusta ser parte de una gran comunidad. Terraform es una herramienta con una gran comunidad y totalmente open source.

Sos analista de infraestructura. Terraform no es una herramienta para todo, está enfocado únicamente en el aprovisionamiento de recursos de infraestructura y hay que tener conocimiento profundo en la materia.

Disaster Recovery. Terraform es muy útil para los equipos que deben cumplir con este proceso.

Clase 10: Infraestructura como código – Terraform (continuación)

Los plugins de Terraform (Clase 10)

Como hemos visto, la función principal de Terraform es la de crear, modificar y destruir recursos de infraestructura.

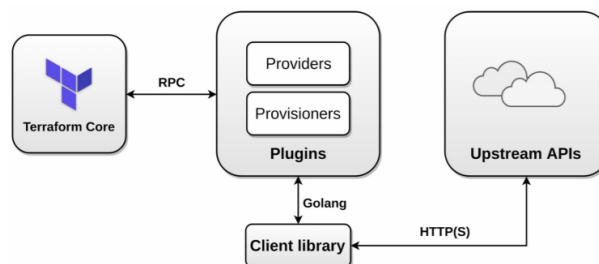
Pero, ¿cómo trabaja este componente realmente? ¿Cómo se comunica con nuestro proveedor de Cloud? ¿Cómo se configura?

Arquitectura

El núcleo de TF está compuesto por varias partes móviles que:

1. Nos proporcionan una capa de abstracción por encima de la API subyacente.
2. Son responsables de interpretar las interacciones de la API y exponer recursos.
3. Soportan múltiples instancias de proveedores cloud.

Si pudiéramos hacer una radiografía de este componente y cómo maneja su flujo de datos, podríamos ver algo similar a lo siguiente:



PLUGINS: Es una aplicación complementaria, generalmente pequeña, que sirve para agregar una funcionalidad extra o adicional (muy específica) a algo ya existente. Los plugins que se utilizan están divididos en: Providers y Provisioners.

PROVIDERS³: Un provider es un plugin “específico” permitirá que nuestro proveedor cloud pueda comprender el idioma en el cual le vamos a hablar. Por ejemplo, para decirle que queremos disponer de un servidor nuevo.

El uso del término “específico” refiere a que existen varios providers, por ejemplo: un provider para AWS, otro para GCP, para Azure, Kubernetes, etc.

PROVISIONER: Un provisioner es un método que se escribe en el código mismo de HCL de Terraform y sirve para saltar cualquier brecha o gap que no pueda ser cubierta por los métodos estándar que Terraform ofrece. Por ejemplo: ejecutar comandos remotos en un servidor.

Nota: de igual manera, Hashicorp, empresa dueña del producto Terraform, recomienda el uso de provisioners solo en casos extremos.

³ <https://registry.terraform.io/browse/providers>

Para esta tarea, existen herramientas de “Configuration Management”, como por ejemplo Ansible o Puppet. Si por alguna razón no se pudiese hacer uso de estas herramientas, Terraform nos ofrece la posibilidad de utilizar este método en su código programable.

```
resource "aws_instance" "web" {  
  # ...  
  provisioner "remote-exec" {  
    inline = [  
      "puppet apply",  
      "consul join ${aws_instance.web.private_ip}",  
    ]  
  }  
}
```

En esta pieza o “snippet”, podemos ver al método “remote-exec” usado para ejecutar comandos remotos

¿CÓMO SE COMUNICA CON LA NUBE?

Del lado de la nube, existe una API especialmente diseñada para saber interpretar los comandos provenientes desde nuestra computadora. Es decir, está a la escucha de nuestras peticiones.

Si el “provider” no existiera, entonces no habría comunicación entre ambas partes. Al ejecutar, por ejemplo, el comando “Terraform plan”, este binario va a buscar al “Provider” que le hemos definido en nuestro módulo de Terraform:

```
# =====  
# Declaramos el Cloud Provider con el que queremos trabajar  
terraform {  
  # Le decimos que queremos:  
  # a. la versión del binario de terraform mayor o igual a 0.12  
  required_version = ">=0.12"  
  required_providers {  
    aws = {  
      # Especificamos desde donde queremos descargar el binario:  
      source = "hashicorp/aws"  
      # Le decimos que solo permitirá:  
      # b. la versión del binario del provider 3.20.0 (con cierta restricción)  
      version = "~> 3.20.0"  
    }  
  }  
}  
# =====
```

Aquí podemos ver que la sentencia “required_providers” está seteada a “aws”, es decir, no me interesa trabajar con Google o con Microsoft, sino, con AWS específicamente.

Luego, mediante la sentencia “source = “hashicorp/aws””, le decimos desde dónde vamos a descargar (algo que se produce de forma automática) este provisioner. Siguiendo con nuestra ilustración, el término “Upstream APIs” se refiere al método que usa el protocolo HTTP para “subir” o “bajar” datos hacia o desde la fuente de origen.

¿POR QUÉ HACE USO DE LA TERMINOLOGÍA HTTP?

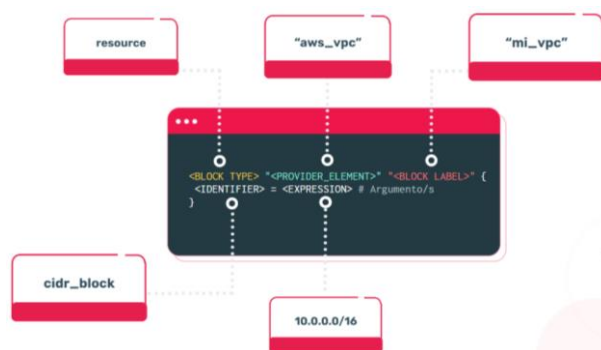
La API que AWS nos ofrece para contactarse con nuestros plugins utiliza las operaciones básicas CRUD (create, read, update, delete). Este modelo es tomado por las operaciones HTTP REST.

CONCLUYENDO

Cuando nos referimos a ejecutar “terraform”, generalmente hablamos de aprovisionamiento para afectar a los objetos de infraestructura reales. Recordemos de clases previas que el binario de Terraform tiene otros subcomandos para una amplia variedad de acciones administrativas: plan, apply, destroy, etc. Pero detrás de todos estos comandos, la arquitectura es la misma.

Domain Specific Language (Clase 10)

HCL es el lenguaje de configuración estructurado que nos permite interactuar con los servicios de infraestructura cloud. Fue creado por Hashicorp para ser amigable para el ser humano y, a su vez, ser interpretado velozmente por una computadora.



Resource: Este bloque iniciará el diálogo con nuestro proveedor cloud. Específicamente estamos diciendo: todo lo que continúa y con lo que voy a trabajar es con un recurso.

aws_vpc: Acá estamos declarando el recurso específico de AWS a utilizar. En este caso, un VPC.

mi_vpc: ¿Cómo lo voy a llamar? ¿De qué forma voy a referenciar si quiero invocar este elemento desde otro módulo?

cidr_block: Un valor definido por HCL. Depende del recurso que declaramos. En este caso, de “aws_vpc”.

10.0.0.0/16: Representa un valor asociado al identificador. Por ejemplo: región = “us-east-1”, donde “us-east-1” es el valor.

Terraform HCL (Clase 10)

Para programar necesitamos entender el código. ¿Qué tenemos que saber de HCL para comenzar a utilizarlo? ¿Cómo lo interpretamos? ¿Cuáles son sus características?

Recordemos que el propósito principal de HCL en Terraform es el de declarar recursos. Esto es importantísimo ya que será la esencia de nuestro código. Mediante este lenguaje le vamos a decir a Terraform cómo administrar una colección determinada de infraestructura.

Semántica y estructura

La sintaxis y estructura del lenguaje HCL de Terraform consta de unos pocos elementos básicos:

```
resource "aws_vpc" "mi_vpc" {
  cidr_block = 10.0.0.0/16
}
```

Analizando la sintaxis general podemos descubrir dos sublenguajes integrados:

- El lenguaje estructural, que define la estructura de configuración jerárquica general y es una serialización de cuerpos, bloques y atributos de HCL.
- El lenguaje de expresión, usado para expresar valores de atributo, ya sea como literales o como derivaciones de otros valores.

En líneas generales estos sublenguajes se usan juntos dentro de los archivos de configuración para describir una configuración general, con el lenguaje estructural.

Tipo de bloque

Tomemos el snippet anterior y veamos detenidamente cada uno de sus componentes:

```
<BLOCK_TYPE> "<PROVIDER_ELEMENT>" "<BLOCK_LABEL>" {
```

Lo primero que necesitamos indicarle es el tipo de bloque que vamos a usar. Los tipos de bloques más comunes que suelen utilizarse son: RESOURCE, VARIABLE y DATA, aunque también existen otros.

Estos bloques “declaran” qué recurso de nuestro proveedor Cloud vamos a usar. Por ejemplo, al escribir:

```
resource "aws_vpc" "mi_vpc"
```

Estamos diciendo que:

1. Necesitamos trabajar con un bloque de tipo “resource”, “RESOURCE”.
2. Que el “resource” a instanciar va a ser “aws_vpc”, “PROVIDER_ELEMENT”.
3. Y que lo queremos llamar “mi_vpc”, “BLOCK_LABEL”.

Argumentos

Seguidamente encontramos los argumentos que asignan un valor a un nombre y aparecen dentro de bloques:

```
# Tipo de Bloque  
  
<IDENTIFIER> = <EXPRESSION> # Argumento/s  
}
```

Es el “contenido” del “block body”. Son los valores y/o funciones que se leen durante el tiempo de ejecución del código y es donde asignamos los valores que queremos para nuestros elementos de infraestructura.

Estos argumentos los podemos dividir en “Identificador” y “Expression”. Ambos están estructurados como llave, valor y están separados por un signo igual.

La llave representa un identificador y el valor es lo que ese identificador almacena.

Un sencillo y claro ejemplo es el de región dentro de provider:

```
region = "us-east-1"
```

Remitiéndonos a nuestro código inicial:

```
cidr_block = 10.0.0.0/16
```

Estamos diciendo que mis argumentos serán:

1. Una llave denominada “cidr_block”
2. Que quiero que la subnet base del VPC que estoy construyendo sea 10.0.0.0/16

Es importante destacar que los argumentos que Terraform acepta no son arbitrarios, sino que está programado para recibir ciertas palabras específicas.

Existen, también, argumentos que son mandatorios, es decir, que deben existir en nuestro código y otros que no. Por ejemplo, “cidr_block” es un argumento requerido porque nuestro VPC requiere que nosotros le digamos en qué red vamos a crearlo.

Variables

HCL usa variables de entrada que sirven como parámetros para un módulo de Terraform.

Declaramos una variable de la siguiente manera:

```
variable "image_id" {  
    type = string  
}
```

La palabra “variable” es un bloque que luego admite solamente un BLOCK_LABEL.

En nuestro caso, “image_id” y que debe ser “único” entre todas las variables ya definidas.

El nombre de una variable puede ser cualquier identificador válido excepto las siguientes: source, version, providers, count, for_each, lifecycle, depends_on, locals que son los denominados meta-argumentos, que no se cubrirán en este curso, pero recomendamos navegarlos.

Dentro de su estructura, también vamos a encontrar argumentos.

Ahora bien, ¿qué estamos diciendo cuando introducimos la palabra “type”? Básicamente, que el contenido va a ser de tipo “string” ya que puede haber casos en los que en lugar de datos de tipo string, podremos tener: number, bool, list, map, tuple, etc.

VARIABLES: UN CASO PRÁCTICO

Supongamos que no queremos que el valor neto de nuestra subnet aparezca “hardcodeado”. Lo que podemos hacer es crear una variable con el dato en sí, y luego, desde, este código, referenciar a esa variable. En nuestro ejemplo inicial:

```
resource "aws_vpc" "mi_vpc" {  
    cidr_block = 10.0.0.0/16  
}
```

Lo podríamos reemplazar, por ejemplo, por los siguientes argumentos:

```
resource "aws_vpc" "mi_vpc" {  
    cidr_block = var.base_cidr_block  
}
```

Para ello alcanza con declarar la variable de la siguiente manera:

```
variable "base_cidr_block" {  
    default = 10.0.0.0/16  
}
```

De esta forma, estamos personalizando aspectos del módulo sin alterar el código fuente del propio módulo.

Otra forma más práctica es definiendo un archivo con el nombre: variables.tf donde declaramos una o más variables logrando modularizar nuestro código.

Nota: es importante que este archivo de variables esté ubicado en el mismo directorio en el que se encuentra nuestro módulo primario.

Sintaxis

PALABRAS RESERVADAS

La mayoría de los lenguajes de programación contienen palabras que no se pueden utilizar como variables de asignación. Dichas palabras se hacen llamar “reservadas” debido a que son utilizadas por el lenguaje para funciones específicas. En el caso de HCL, *Terraform no dispone de palabras reservadas a nivel global.*

COMENTARIOS

Existen tres sintaxis diferentes para comentarios:

1. # comienza un comentario de una sola línea, que termina al final de la línea.
2. // también comienza un comentario de una sola línea, como alternativa a #.
3. /* y */ son delimitadores de inicio y fin de un comentario que puede abarcar varias líneas.

El estilo de comentario # de una sola línea es el estilo de comentario predeterminado y debe usarse en la mayoría de los casos.

No es necesario conocer todos los detalles de la sintaxis HCL para utilizar Terraform. De hecho, solo con comprender su estructura y qué es lo que se quiere lograr ya estamos en condiciones de crear módulos más complejos. El resto consiste en acudir a la documentación oficial para buscar que tipo de recursos preciso, qué argumentos lleva y ¡un poco de imaginación!

Ambiente Productivo (Clase 10)

¿Cómo es un ambiente productivo con el que nos vamos a encontrar en nuestra vida profesional? Un ambiente productivo generalmente está dividido en dos o más subredes donde corren servicios que necesitan ser alcanzados desde internet, mientras hay otros que no deben ser accesibles. En esta ocasión, vamos a levantar una infraestructura donde dispondremos de una red pública y otra privada.

Paso 1

Antes de comenzar a levantar infraestructura debemos tener bien en claro los requerimientos, comprender cuales son los recursos que queremos utilizar. En nuestro caso queremos focalizarnos en la estructura base que soportará los componentes esenciales de red donde nuestras aplicaciones serán montadas.

Pensar de entrada en estos elementos también nos lleva a determinar qué variables vamos a necesitar. Por ejemplo, cómo vamos a requerir un vpc, es normal que necesitemos una subnet, una región, etc.

Paso 2

Una buena práctica en desarrollo es la de incluir un encabezado o header donde se indique de qué se trata el código que se está escribiendo. Ubicamos el propósito de nuestro código, quien lo hizo, fecha y versión. En algunos casos se suele agregar también el repositorio donde se aloja.

```
1 # Propósito: crear infraestructura AWS
2 # Autor: DH
3 # Fecha: 30.07.21
4 # Versión: 1.0
5
```

Paso 3

Creamos nuestro módulo de variables (variables.tf).

```
1 variable "<LABEL>" {
2     description = "una breve descripción
3 de la variable"
4     type        = <el_tipo_de_dato> #
5 puede ser string, number, bool, list,
6 etc.
7     default = "<el_valor_a_asignarle>"
8 }
9
```

Luego vamos a declarar las variables que queremos utilizar, entre los puntos más relevantes encontramos la red 10.0.0.0/24 que es la IP desde donde se comenzarán a segmentar las otras redes internas.

Encontramos también dos subnets: pública y privada. Recordemos que queremos preparar nuestra red de forma segura, una buena práctica es la de separar redes, instalando servicios que consideremos críticos, como una base de datos y una red aislada de internet cuyo tráfico puede salir, pero no entrar si la conexión no es establecida desde el mismo origen.

```
1 variable "aws_region_id" {
2     description = "la región"
3     type        = string
4     default = "us-east-1"
5 }
6 variable "main_vpc_cidr" {
7     description = "Nuestro Security Group"
8     type        = string
9     default = "10.0.0.0/24"
10 }
11
12 variable "public_subnets" {
13     description = "subnet con acceso a internet"
14     type        = string
15     default = "10.0.0.128/26"
16 }
17
18 variable "private_subnets" {
19     description = "subnet sin acceso a internet"
20     type        = string
21     default = "10.0.0.192/26"
22 }
```

Paso 4

Ahora vamos a crear el módulo de proveedores cloud (providers.tf), lo que acá estamos diciendo es: vamos a trabajar con AWS, por lo tanto, vamos a utilizar ese proveedor. Entonces, cuando

ejecutemos **terraform init**, terraform va a leer este módulo y va a saber que tiene que descargarse desde “hashicorp/aws” el provider AWS.

En este módulo estamos trabajando con dos archivos: el binario de Terraform y el binario del Provider. Nosotros vamos a decirle que queremos trabajar con la versión del binario de Terraform mayor o igual a la versión 0.12. ¿Por qué? Básicamente porque hay un quiebre en la madurez del producto a partir de la 0.12, nuevas funciones, corrección de problemas, etc.

```
1 terraform {
2   required_version = ">=0.12"
3   required_providers {
4     aws = {
5       source = "hashicorp/aws"
6       version = "~> 3.20.0"
7     }
8   }
9 }
10
```

Luego debemos declarar nuevamente el bloque Provider y le pasamos un elemento llamado AWS. Esto es así porque necesitamos declarar la región en la que toda esta infraestructura se levantará.

```
1 provider "aws" {
2   region = "us-east-1"
3 }
4
5
```

Sino detallamos la región a la hora de ejecutar Terraform plan, nos solicitaría entrar manualmente a la misma lo que de alguna manera rompería un poco con nuestra automatización.

Paso 5

Creamos el vpc en AWS archivo main.tf, para esto vamos a invocar al elemento aws_vpc que nos permitirá definir un vpc nuevo.

Example Usage

Basic usage:

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}
```

Basic usage with tags:

```
resource "aws_vpc" "main" {
  cidr_block      = "10.0.0.0/16"
  instance_tenancy = "default"

  tags = {
```

Pero ¿cómo llegamos a esta conclusión? ¿cómo sabemos que para crear un nuevo vpc necesitamos al elemento “aws_vpc y no otro? La respuesta rápida y simple es porque la documentación oficial lo dice así. De hecho, una práctica común a medida que vamos codeando es la de tener al lado el sitio de la documentación abierto e ir buscando referencias de lo que queremos hacer.

Acto seguido vamos a definir el identificador (cidr_block), el cual es requerido. ¿Por qué es requerido? Nuevamente podemos decir que es porque la documentación así lo dice, es de esta manera como AWS definió que se cree un vpc.

Paso 6

Continuando con nuestra aventura tenemos el bloque correspondiente a la creación de un internet Gateway, en términos de AWS, un Internet Gateway es una puerta de enlace desde nuestro VPC hacia internet. El recurso se llama `aws_internet_gateway` al que le asignamos un label o alias por medio del cual, podremos hacer llamadas desde otros bloques o módulos de Terraform en caso de que queramos hacer uso de `aws_internet_gateway`.

```
1 resource "aws_internet_gateway" "IGW" {
2   vpc_id = aws_vpc.Main.id
3   tags = {
4     Name = "IGW"
5   }
6 }
7
```

Luego del label main vemos un `.id` ¿Qué significa esto? El ID es un valor alfanumérico único que identifica el recurso. ¿Por qué ponemos un ID? Resulta que es parte de la extensibilidad que tiene en particular este argumento, de igual forma, obtenemos su valor llamado al archivo de variables `"variables.tf"` y le damos un tag.

Fijémonos un detalle, el patrón de implementación, el resource tag está dentro del mismo bloque `resource aws_internet_gateway`, es decir, que Terraform HCL nos *permite anidar bloques dependiendo de la necesidad*.

Paso 7

Comenzamos con la creación de las subnets, recordemos que vamos a levantar dos subredes una pública y otra privada. En terminología AWS una subnet pública no es más que una subnet que puede conectarse internet y que también puede recibir tráfico entrante. En este caso, vemos que no sólo debemos setear un identificador llamado `cidr_block` sino también otros llamado `vpc_id`. Esto es así porque tenemos que decirle a AWS en qué vpc se encontrará esta subnet que estamos creando.

```
1 resource "aws_subnet" "public_subnets" {
2   vpc_id = aws_vpc.Main.id
3   cidr_block = var.public_subnets
4   tags = {
5     Name = "Public Subnet"
6   }
7 }
8
```

Ahora vamos con la creación de la subnet privada la cual contendrá servicios que generalmente no queremos que sean accesibles vía internet, por ejemplo, una base de datos. Para la subnet privada empleamos la misma lógica que con la subnet pública.

```

1 resource "aws_subnet" "private_subnets" {
2   vpc_id = aws_vpc.Main.id
3   cidr_block = var.private_subnets
4   tags = {
5     Name = "Private Subnet"
6   }
7 }
8

```

Paso 8

Las cosas comienzan a ponerse interesantes. Cuando levantamos un vpc por primera vez este viene con una tabla de ruteo que, por default, rutea todo el tráfico proveniente de todas la subnets asociadas. Como nosotros queremos que sólo el tráfico de la subnet pública vaya a internet no es óptimo que utilicemos este default route table. Por lo tanto, vamos a crear una tabla de ruteo específica para esta subnet.

De acuerdo a la documentación oficial, esto se logra con el recurso “aws_route_table”, el cual requiere los siguientes argumentos de forma obligatoria: vpc_id y cidr_block.

```

1 resource "aws_route_table" "Public_RT" {
2   vpc_id = aws_vpc.Main.id
3   route {
4     cidr_block = "0.0.0.0/0"
5     gateway_id =
6     aws_internet_gateway.IGW.id
7   }
8   tags = {
9     Name = "Tabla de Ruteo Pública"
10  }
11 }
12

```

Hacemos exactamente lo mismo con nuestra subnet privada, la diferencia es que la subnet no estará asociada a la tabla de ruteo que tiene declarada la salida directa a internet, sino que tiene asociado un nat_gateway.

```

1 resource "aws_route_table" "Private_RT" {
2   vpc_id = aws_vpc.Main.id
3   route {
4     cidr_block = "0.0.0.0/0"
5     nat_gateway_id =
6     aws_internet_gateway.NAT_GW.id
7   }
8   tags = {
9     Name = "Tabla de Ruteo Privada"
10  }
11 }
12

```

Paso 9

Ya hemos asociado la tabla de ruteo con nuestro Internet Gateway. Ahora debemos asociar nuestra tabla de ruteo con la subnet pública de tal forma que, la subnet, sepa qué ruta seguir para salir a internet. Entonces, nuevamente declaramos el bloque resource y nos traemos el elemento “aws_route_table_association” que nos va a realizar la vinculación.

Vemos que toma un identificador requerido llamado “route_table_id” que le vamos a dar el valor del ID de la tabla de ruteo que creamos y qué llamamos “public_RT.id”. También le vamos a dar el identificador no requerido “subnet_id” para pasarle el ID de la subnet y así, tener los elementos que vamos a asociar.

```
1 resource "aws_route_table_association"
2   "Public_RT_Association" {
3     subnet_id =
4     aws_subnet.public_subnets.id
5     route_table_id =
6     aws_route_table.Public_RT.id
7   }
8
```

Seguimos con asociación de la subnet privada con su propia tabla de ruteo. Como esta subnet no debe salir a internet, se define con la red 0.0.0.0/0 que significa que no hay direcciones asignadas y se interpreta como todas las redes, es decir, también internet.

```
1 resource "aws_route_table_association"
2   "Private_RT_Association" {
3     subnet_id =
4     aws_subnet.private_subnets.id
5     route_table_id =
6     aws_route_table.Private_RT.id
7   }
8
```

Paso 10

Nos falta una dirección IP estática, para ello generamos una eip y se lo daremos al NatGateway.

```
1 resource "aws_eip" "NAT_EIP" {
2   vpc = false
3   tags = {
4     Name = "NAT con elastic IP"
5   }
6 }
7
```

Paso 11

Para finalizar, asociamos el NatGateway con la EIP que creamos previamente.

```
1 resource "aws_nat_gateway" "NAT_GW" {
2   allocation_id = aws_eip.NAT_EIP.id
3   subnet_id =
4   aws_subnet.public_subnets.id
5   tags = {
6     Name = "NAT Gateway + EIP alocadas a
7   la subnet pública"
8   }
9 }
10
```

Toda infraestructura requiere de cimientos sólidos para mantenerse estable y segura durante su ciclo de vida, es por eso, que la etapa de diseño es muy importante. Requiere comprender bien qué

es lo que vamos a hacer antes de hacerlo, identificar cuáles son los elementos que necesitaremos, como conjugarlos, como asegurarlos, etc.

Clase 9: Cierre de la semana

Resumen de la semana

Durante esta semana conocimos dos herramientas nuevas: Ansible y Terraform. Con ambas creamos los mismos recursos en nuestra cuenta de Amazon Web Services. Si sumamos CloudFormation, podemos decir que conocemos las tres herramientas más demandadas en el mercado.

EJECUCIÓN REMOTA: Con CloudFormation no necesitamos conectarnos a AWS, ¡ya estamos dentro! Con Ansible y Terraform, al ejecutar desde fuera de nuestra cuenta, aprendimos cómo conectarnos desde nuestras computadoras y sin contraseñas

OPENSOURCE Y COMUNIDAD: Los módulos usados en Ansible y Terraform, en su mayoría son apoyados por la comunidad de dos herramientas que comparten su código fuente.

ESTADO DE LA INFRAESTRUCTURA: Con Terraform, aprendimos uno de sus mayores beneficios y lo pusimos en práctica: destruir la infraestructura para corregir y mejorar ¡es tan fácil como crearla!

Clase 10: Infraestructura como código – Terraform (continuación)

CloudFormation vs Ansible vs Terraform

Las tres herramientas más usadas y conocidas en el mercado para implementar infraestructura como código son Cloud Formation, Ansible y Terraform. Veamos cuál de estas tres es la mejor.

De las tres herramientas Ansible es la que posee la infraestructura más simple y fácil de usar, además es agentless, esto quiere decir que no necesita instalar un agente en un nodo para administrarlo.

Hay que darle también mérito a Cloud Formation, en cuanto a la facilidad para aprender a usarla ya que posee una gran documentación en la web oficial con ejemplos muy práctico. Tomando en cuenta el aprovisionamiento de infraestructura, Cloud Formation y Terraform tienen una ventaja sobre Ansible, ya que esta última antes de transformarse y abarcar otras funcionalidades, fue concebida inicialmente como una herramienta de gestión de configuraciones, es decir, estaba enfocada únicamente en realizar configuraciones masivas en servidores.



Ahora, en cuanto a los servidores y las aplicaciones que nos brindan las herramientas, la ganadora es Ansible. Esta herramienta creada con la idea de aplicar configuraciones masivas nos va a permitir realizar ambas tareas: crear tanto el servidor como servicios y aplicaciones que expongan a ese servidor.

Con Terraform, por ejemplo, solo vamos a poder realizar la creación del servidor. Si evaluamos su modularidad, es decir la facilidad para agrupar código, Terraform es la herramienta que aventaja a

las otras dos ya que puede dividir código, no sólo en distintos archivos, sino también en distintos bloques que utilizan un recurso cloud.

De las tres herramientas, las más populares y las que más soporte de su comunidad tienen son Terraform y Ansible, al ser opensource son fácilmente adaptables si nos vemos en la necesidad de migrar de proveedor de infraestructura Cloud. Además, garantizan una constante actualización y mantenimiento por parte de sus comunidades adecuando sus usos según las necesidades del mercado.

Dos de ellas son productos que poseen la mayor solidez y madurez: Cloud Formatios y Ansible por una sencilla razón, ambas se crearon antes que Terraform, lo que implica que pasaron por una mayor cantidad de situaciones, versiones, transformaciones y actualizaciones antes que Terraform.

Si tomamos en consideración la ejecución desde cualquier plataforma, Terraform es claramente la ganadora ya que puede ejecutarse desde cualquier sistema operativo, ofrece un archivo binario para distintas plataformas. Mientras que Ansible no funciona nativamente en Windows, necesita una capa intermedia para poderse ejecutar. Y Cloud Formation solo funcionan Amazon web Services.

La gran ventaja de Cloud Formation es la integración en AWS, servicio de computación en la nube que ocupa el 33% del mercado lo que nos brinda la posibilidad de implementar prácticas de infraestructura como código de forma nativa, beneficiándose de todo su ecosistema.

CloudFormation, Terraform y Ansible son las herramientas más potentes, versátiles y populares usadas en el mercado. Cada una de ellas presenta sus ventajas y desventajas. Lo importante es conocerlas, aprender a usarlas y elegir cual es la más adecuada acorde a las necesidades del proyecto.

Clase 12: Cierre de la Semana 4

Resumen de la Semana

¿QUÉ APRENDIMOS?: Nos adentramos más en el manejo de IAC utilizando Terraform. Aprendimos sobre su funcionamiento interno, la manera en que se comunica con la nube, vimos cómo se estructura su lenguaje de programación y trabajamos levantando recursos de Infraestructura en AWS.

DOCUMENTACIÓN: No necesitamos ser expertos para comenzar a escribir código y levantar recursos en la nube. ¡La documentación de Terraform es nuestra aliada!

HCL: Descubrimos que el esqueleto del lenguaje es sumamente sencillo, ya que está compuesto únicamente por unos pocos elementos para su estructuración.

API: Vimos que Terraform utiliza proveedores, que son plugins que se comunican con la nube. Esto, por su lado, nos ofrecen unas APIs para que podamos hablarle.

MODULARIZACIÓN: La legibilidad de código es importante, para ello, podemos segmentar nuestro código en distintos archivos o módulos que serán invocados e interpretados por Terraform en tiempo de ejecución.

DECLARATIVIDAD: Ahondamos en la esencia misma de IaC mediante la sencilla declaración de recursos de infraestructura.

Módulo 3: Pipelines

Pipelines ¿qué son y para qué sirven?

Imaginemos una fábrica que produce un bien físico. Necesita materia prima, ciertas máquinas, personal que las controle y procesos que definan, de forma lógica, la secuencia de pasos que se deben ejecutar para lograr el producto que finalmente saldrá de la fábrica y llegará a algún mercado para su consumo. En la infraestructura como código estos procesos lógicos que permiten la elaboración de un producto son los que se conocen como pipelines.

Un *pipeline es uno o más procesos automatizados que sirven para construir o implementar cierto código de software de forma rápida, escalable, flexible y segura* minimizando errores humanos que derivan de la tarea manual. Un ejemplo sería el siguiente: tenemos una plataforma Cloud que hospeda un sitio web y queremos implementar cierta feature, una nueva opción. Para ello necesitamos los siguientes elementos: una nueva pieza de software, los pasos que el pipeline realizará, accesos apropiados al ambiente de testing y el destino donde esta feature será implementada.

Vía GIT pusheamos el código ya revisado por alguien más a un repositorio remoto, un lugar con una carpeta en otro servidor. Nuestro producto de pipeline recibe el código y este es interpretado según una serie de reglas preestablecidas.

¿Cuáles podrían ser estas reglas?

La **PRIMERA** es validar que exista un motor de Software que pueda compilar el código, es decir, si nuestro código de inputs es HCL entonces vamos a precisar el binario de Terraform para que puede interpretarlo correctamente. Si nuestro código viene hecho en GO vamos a necesitar tener instalado el compilador de Go.

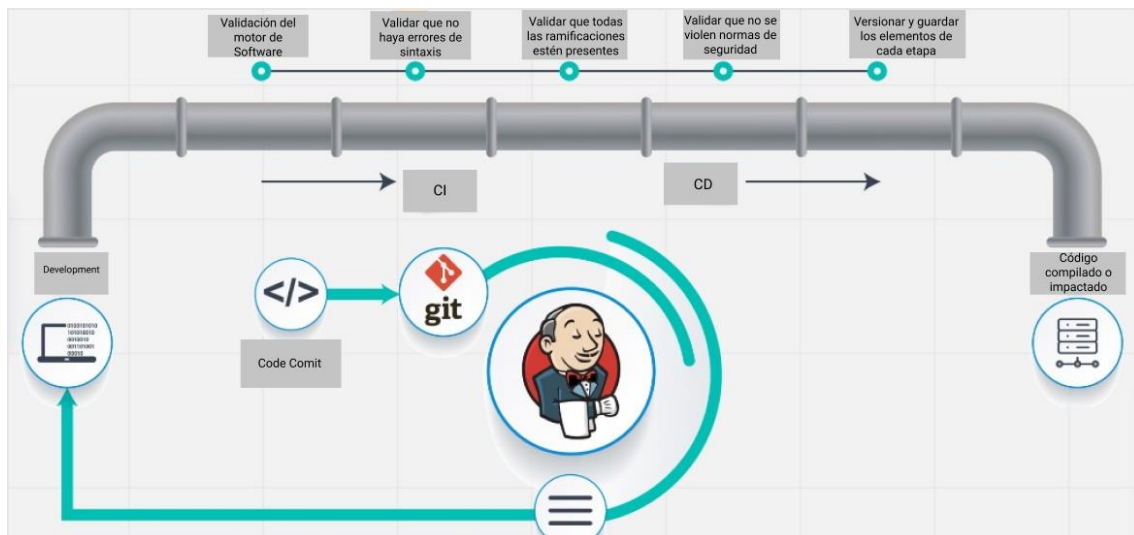
La **SEGUNDA** es validar que no existan errores de sintaxis.

La **TERCERA** validar que todos los módulos, referencias o ramificaciones estén presentes.

La **CUARTA** validar que no se estén violando condiciones específicas de seguridad.

La **QUINTA**, versionar y guardar el o los elementos resultantes de cada etapa.

Nuestro código pasará por cada una de estas etapas y puede, o no, ir arrojando un resultado que luego será consumido por la próxima etapa del pipeline. Finalmente, el código puede ser compilado, impactado en el sitio web que será actualizado automáticamente.



Es importante destacar que según la madurez del proceso y del equipo, la salida de nuestro pipeline puede no impactar ni siquiera en nuestro ambiente de testing, sino que se frena el proceso un paso antes y se requiere una acción manual para realizar el pasaje al sitio web.

Los pipelines dentro de la infraestructura moderna nos brindan una serie de ventajas como:

- **VELOCIDAD:** Al ser generalmente un proceso automatizado donde no hay que ejecutar comandos o hacer clic en opciones determinadas, permite correr a la velocidad de procesamiento que se haya destinado a tal proceso.
- **CONSISTENCIA:** Siempre vamos a obtener el mismo resultado logrando la integridad del producto final.
- **SEGURIDAD:** Con el proceso automatizado y en un ambiente controlado vamos a tener la certeza de que no estamos introduciendo ninguna vulnerabilidad.
- **VERSIONADO:** Como mencionamos anteriormente podemos ir guardando los elementos resultantes por cada etapa o iteración con determinados tags o números de versión.

El *uso de pipelines es fundamental* en cualquier proceso de infraestructura como código y en esencia toda persona que trabaja en software tarde o temprano se va a encontrar con este tipo arquitectura de Release.

En resumen, *pipelines es una práctica que nos sirve para agilizar los procesos de desarrollo e implementación de software* que se vale de automatizar pasos repetitivos según un procedimiento preestablecido. Esta arquitectura es bastante común en el desarrollo de software porque tiene la propiedad de agilizar los procesos de build, deploy y release.

Para hacernos una idea más concreta, lo podemos pensar como una serie de comandos encadenados donde cada comando está en un nivel determinado. Cuando el comando finaliza su ejecución, el resultante pasa al siguiente nivel donde otro lo espera para ejecutar una función predeterminada.

Por ejemplo:

1. Una etapa puede ser el chequeo de que aquello que ingresó en el proceso cumpla con ciertos requisitos iniciales de seguridad. Si estos requisitos no satisfacen ciertas reglas predefinidas de antemano, entonces no puede continuar adelante.

2. En una etapa posterior se puede, por ejemplo, compilar el código.
3. En la siguiente puede ser almacenado en algún lugar específico. Así, sucesivamente, hasta llegar al final de la línea.

Un pipeline no es un proceso monolítico, sino que puede consistir en una o varias etapas que están encadenadas o secuenciadas y que van transformando a nuestro elemento inicial de acuerdo con ciertas reglas hasta llegar al producto final.

¿Qué tecnologías existen?⁴

Cuando hablamos de “collaboration” en términos de archivos, quizás en lo primero que pensamos es en Dropbox o Google Drive. Estos sistemas son fáciles de utilizar, poseen una interfaz intuitiva y permiten controlar archivos. Sin embargo, es bastante difícil adaptarlos a las necesidades con las que nos podemos encontrar al trabajar en tecnología. La industria del software requiere de productos o soluciones que puedan:

- Actualizar código en tiempo real.
- Controlar conflictos en el código.
- Disponer de “Roadmaps” para planificar y determinar milestones.
- Sincronizar versiones de archivos.
- Almacenar grandes cantidades de archivos.
- Automatizar procesos de compilación de archivos.

Este tipo de requerimientos son contemplados por productos muy específicos creados para tal función.

<ul style="list-style-type: none"> • Es un servicio de repositorio y control de versiones para colaboración en proyectos de código de software • Tiene una característica llamada github issues que funciona como herramienta de ticketing • Dispone de boards para hacer seguimiento sobre "issues" creados 	<ul style="list-style-type: none"> • Al igual que GitHub es una herramienta de repositorios que permite a los usuarios poder colaborar en un proyecto de software • Se ha hecho tan popular que existen imágenes de Docker certificadas de tal forma que se puede descargar y utilizar localmente • Posee una característica llamada CI que permite crear y hacer uso de pipelines para actualizar el código de nuestro proyecto 	<ul style="list-style-type: none"> • Repositorio que permite colaborar en proyectos de software. Es muy similar a los anteriores pero sin opción free. • Al ser un producto de Atlassian, se integra muy bien a JIRA y todos los productos de la misma línea. No necesita de procesos complejos de integración • Dispone de boards para hacer seguimientos sobre "issues" creados
		

Todos estos productos disponen de flujos de trabajo altamente configurables según las necesidades del usuario. Pero, ¿qué son los flujos de trabajo? ¿De qué se tratan? Los flujos de trabajo dan cuenta de las características innatas o facilidades que estos sistemas nos ofrecen:

- Almacenamiento de código
- Versionado

⁴ <https://github.com/about> - <https://about.gitlab.com/> - <https://bitbucket.org/>

- CI/CD
- Control de cambios
- Colaboración y revisión

Jenkins

Jenkins es un servidor diseñado para la integración continua (CI). Es gratuito y de código abierto (open source). ¡Y se ha convertido en el software más utilizado para esta tarea!

¿Qué podemos hacer en Jenkins? Nos permite organizar una cadena de acciones que ayudan a lograr el proceso de integración continua (¡y mucho más!) de manera automatizada.

Características

- **AUTOMATIZACIÓN:** Es el software de automatización más usado para la integración continua.
- **COMUNIDAD:** La comunidad ha desarrollado más de 15000 plugins para agilizar las tareas de los equipos de desarrollo.
- **TAREAS:** Jenkins puede orquestar cualquier tipo de proceso y ejecutar tareas manuales, periódicas o automáticas.
- **FÁCIL DE USAR:** Su uso es sencillo y puede aumentar su capacidad de cómputo añadiendo nuevos agentes o servidores

Instalación

La instalación varía según el sistema operativo. Cualquiera es válida tanto para servidores como para nuestras computadoras personales. Las más comunes son:

- Correr Jenkins como contenedor de Docker o bien usar la imagen oficial para ejecutarla en Kubernetes.
- Descargar un MSI para instalar en Windows 10 o Windows Server.
- Se puede utilizar en distribuciones basadas en Red Hat, como RHEL, CentOS o Fedora a través de RPM.
- Implementarlo en Ubuntu/Debian a través de un paquete “.deb”.

Configuración

En la sección de configuración podemos personalizar varias opciones:

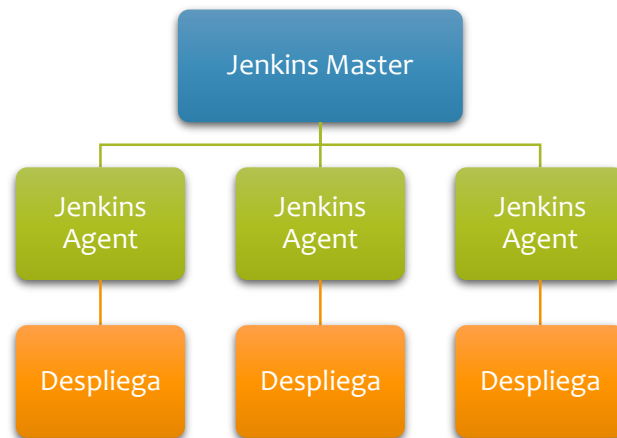
SMTP Y OTROS CANALES: Podemos configurar a Jenkins para que se comunique con nosotros por mail, cargando la información de nuestro SMTP o WebHooks para usar canales como Slack.

VARIABLES DE ENTORNO: En nuestra instancia de Jenkins podemos guardar información sensible (por ejemplo, los datos de conexión a nuestros servidores como par de llaves para ssh).

SEGURIDAD PRIMERO: Podemos configurar los roles y permisos de nuestro Jenkins para limitar el acceso de usuarios. Es importante que cada usuario tenga los permisos mínimos para realizar sus tareas.

Distributivo

Jenkins puede distribuir el trabajo en varias máquinas, lo que ayuda a realizar las compilaciones, pruebas e implementaciones en varias plataformas con mayor rapidez.



Plugins

Con cientos de complementos en el Centro de Actualización, Jenkins se integra con prácticamente todas las herramientas en la cadena de la integración y entrega continua. Algunos ejemplos son:

- Docker
- Pipelines
- Slack Notifications
- Azure

Jenkinsfile

Al crear un archivo llamado Jenkinsfile podemos definir nuestro pipeline a través de código escrito en Groovy (un derivado de Java).

Esto nos brinda algunos beneficios:

- Extender el uso de los plugins.
- Personalizar algunos usos que no se encuentran en plugins.
- Que nuestro pipeline tenga la misma lógica que una aplicación a través de estructuras de control.

```
pipeline {  
    // Podemos usar información cifrada como claves a través de código  
    environment {  
        AWS_ACCESS_KEY_ID = credentials('jenkins-aws-secret-key-id')  
        AWS_SECRET_ACCESS_KEY = credentials('jenkins-aws-secret-access-key')  
    }  
}
```

Jenkinsfile: Dividimos en pasos

Los pipelines se encuentran divididos en distintos pasos que se encadenan entre sí para obtener el resultado esperado al final del pipeline. Los llamamos Stages.

```

pipeline {
  stages {
    stage('Stage 1') {
    }
    stage('Stage 2') {
    }
  }
}

pipeline {
  stages {
    stage('Stage 1') {
      steps {
        sh("kubectl --kubeconfig
$MY_KUBECONFIG get pods")
      }
    }
  }
}

```

Conociendo Jenkins a través del Jenkinsfile

Un Jenkinsfile es un archivo que contiene las reglas o pasos a ser ejecutados en el pipeline. Este puede ser declarativo o con script, el archivo reside en el repositorio junto al resto del código.

Jenkinsfile declarativo

Este archivo está compuesto por varios bloques. El pipeline es el bloque principal donde residen el resto de los bloques. El bloque stages engloba un subconjunto de tareas que se realizan a través de todo el ciclo del pipeline: construir, probar, desplegar, etcétera. Es utilizado por varios plugins para visualizar o mostrar el estado y progreso del proceso. Cada una de las tareas que componen una etapa se denominan steps, básicamente cada paso le dice a Jenkins que hacer en cada uno de los puntos concretos del ciclo a realizar.

Veamos un Jenkinsfile simple:

Acá podemos ver todos los bloques mencionados. Dentro del bloque pipeline se encuentran todos los sub bloques y además una línea “agent any” que hace referencia a agente en que se quiere ejecutar ese pipeline. Luego tenemos el bloque stage que contiene las distintas etapas del pipeline, en este caso hay tres etapas bien definidas: build, test y deployed. Dentro de cada uno de estos bloques están las acciones a llevar a cabo.

```

1 pipeline {
2   agent any
3
4   stages {
5     stage('Build') {
6       steps {
7         echo 'Building...'
8       }
9     }
10    stage('Test') {
11      steps {
12        echo 'Testing...'
13      }
14    }
15    stage('Deploy') {
16      steps {
17        echo 'Deploying...'
18      }
19    }
20  }
21 }

```

En este ejemplo solo colocamos un texto en la salida, pero en el bloque step podemos ejecutar comandos usando sh comando o incluso hacer uso de distintos plugins. Esto se implementa en los pipelines.

Veamos una vez más el logo de Jenkins, que es un mayordomo, y justamente podemos pensar a Jenkins de ese modo. Como alguien que hace lo que se le solicita. En este caso lleva a cabo la ejecución del pipeline, o sea, construye nuestro software, lo testea y lo despliega.



Para poder hacer todo esto Jenkins necesita saber qué es exactamente lo que queremos que haga, es ahí donde entran juego el Jenkins File. Este archivo es una guía que Jenkins tiene que seguir para que todo salga como queremos. Por ejemplo, en esta guía podríamos solicitar que construya el software para obtener un ejecutable para Windows y para Linux. Luego que se ejecuten los test para verificar que el software hace todo lo necesario para su correcto funcionamiento y, además, que no tenga errores. A continuación, si los tests son exitosos queremos que suba los ejecutables a un repositorio para su almacenamiento y finalmente, deseamos que ese ejecutable sea descargado en el servidor correspondiente y se ejecute, teniendo así, una nueva versión del Software funcionando. ¡Y todo esto de manera automática!

Si bien hemos usado un Jenkinsfile declarativo, debemos tener en cuenta que no es la única forma ya que también Jenkinsfile scripted. Estos ofrecen un mayor control y mayor poder a la hora de indicar los pasos de un pipeline. Sin embargo, esto tiene la desventaja de que está en un lenguaje de programación poco utilizado y además suele ser más complejo, con una curva aprendizaje mayor a la de los Jenkinsfile declarativos.

Cabe destacar también que estos archivos están siempre con el código fuente del software lo que ofrece todas las ventajas del uso del controlador de versiones como la colaboración entre el equipo, transparencia, versionado, etcétera.

Según los principios de DevOps tenemos que eliminar la dependencia de las personas que desarrollan hacia las personas encargadas de operaciones o Sysadmins. Esto permite que quienes desarrollan pueden desplegar sus aplicaciones de forma automática y sin intervenciones de un Sysadmins lo que agiliza el proceso de desarrollo de software.

Clase 14: Pipelines – Build & continuos integration

El proceso de build

El pipeline es un componente fundamental en el desarrollo de software automatizado. Si bien el término se ha utilizado para describir muchos aspectos diferentes de la informática, en la industria de DevOps usamos pipelines para ilustrar las amplias aplicaciones de comportamientos y procesos involucrados en integración continua.

Pensemos entonces en los pipelines y sus procesos. Podemos reconocer distintas etapas que conforman un pipeline. Vamos a dedicarnos a una de ellas: el proceso de build o construcción de una aplicación. Es decir, la transformación del código fuente en una aplicación funcional para ser ejecutada por usuarios.

Comencemos el recorrido por este proceso. Veremos cómo se organiza, la manera en que se compone, y cómo se integra con Jenkins como herramienta de integración continua.

Proceso de compilación usando Jenkins

Vamos a hablar del proceso de compilación build de una aplicación utilizando Jenkins como herramienta de automatización.

Cuando hablamos de build nos referimos al proceso de construcción de una aplicación, es decir, a la transformación del código fuente a una aplicación funcional para ser ejecutada por usuarios. Este proceso no implica para algunos lenguajes de programación como python o JavaScript que conforman el grupo de lenguajes interpretados en estos, el código fuente es interpretado tal cual como está al momento de la ejecución del programa.

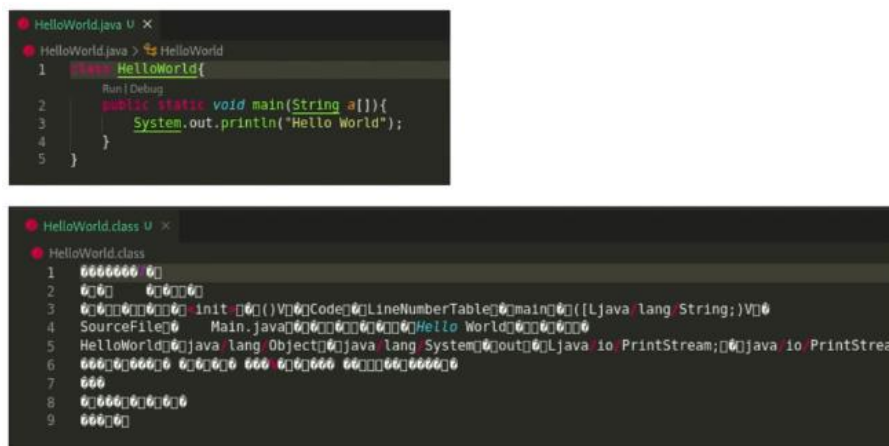
Otro grupo de lenguajes son los combinables que, si tienen como condición necesaria tener una etapa de compilación de la aplicación para poder ser utilizadas, algunos ejemplos son C, C++, C#, Java.

Cuando escribimos nuestro código Java lo hacemos en archivos con extensión .Java el cual lo entendemos únicamente cuando son abiertos con programas llamados IDE que no son más que entorno de desarrollo y no son aptos para el uso de usuarios finales. Para que una aplicación Java pueda ser utilizada, este archivo .java debe transformarse en un archivo ejecutable que tiene extensión .class, en este proceso el archivo no pierde su nombre solo se modifica la extensión. Pero entonces **¿POR QUÉ SE COMPILAN LOS PROGRAMAS?** La compilación es necesaria para que la máquina comprenda nuestro código. Para esto el programa que estamos desarrollando se transforma a *Bytecode*, un lenguaje de máquina que el procesador de la computadora o servidor puede comprender y ejecutar.

Un código Java que imprima hello-world se ve así:

```
class HelloWorld{
    public static void main(String a[]){
        System.out.println("Hello World");
    }
}
```

El código fuente se duplica al ser compilado en un archivo con el mismo nombre, pero con la extensión .class, esta extensión de archivo compilado puede variar según el lenguaje o la finalidad de la aplicación. Comparando nuestros archivos se ven así:



The image contains two screenshots of code editors. The top screenshot shows a file named 'HelloWorld.java' with the following code:

```
1 class HelloWorld{
2     public static void main(String a[]){
3         System.out.println("Hello World");
4     }
5 }
```

The bottom screenshot shows a file named 'HelloWorld.class' containing the compiled bytecode, which is a series of hexadecimal characters and some keywords like 'init', 'main', 'SourceFile', 'HelloWorld', 'java lang Object', 'java lang System', 'java io PrintStream', and 'java io PrintStream'.

Este proceso de Build, construcción o compilación es automatizable dentro de la metodología DevOps. Como todos los procesos tenemos tres etapas donde nuestra entrada, la etapa de codificación, es cuando el programador está escribiendo el programa. El proceso es la compilación y la salida es el entregable para las pruebas.

Con Jenkins tenemos la posibilidad de hacer todo el ciclo DevOps con sus 8 etapas. Esta herramienta, aunque no es la única que permite hacer esto, sí es una de las más populares. Para centrarnos en la etapa de compilación de una aplicación Java hacemos lo siguiente:

1. Creamos una tarea de tipo estilo libre, la tenemos que nombrar de la forma más descriptiva posible.
2. Dentro del grupo de un proyecto de estilo libre vamos a escribir una breve descripción
3. Elegimos el tipo de ejecución que necesitamos para nuestro proyecto
4. Hacemos clic en guardar.

Ya tenemos nuestro proyecto configurado, ahora podemos ejecutarlo cada vez que lo necesitamos con la opción construir ahora”. Esto va a dar como resultado una ejecución donde el ID se crea automáticamente. Al observar la salida podemos ver que está marcada como exitosa y vemos lo que escribimos en nuestro código, además del script que ejecutamos para automatizar esta tarea.



Comprender el concepto de build, construcción o compilación es fundamental para realizar entregables a nuestros clientes con los equipos dedicados a testear las soluciones. Además, es importante saber que este proceso de transformación es totalmente automatizable con herramientas como Jenkins.

Continuous integration

Pensemos en una aplicación cuyo código está almacenado en un repositorio remoto. Los desarrolladores suben cambios al código todos los días, varias veces al día. Por cada envío al repo se ejecutan un conjunto de pasos automáticos que terminan construyendo una aplicación.

Estos pasos no solo arman la aplicación, ¡también la testean! Entonces...

La integración continua (CI) es una práctica de desarrollo que requiere que los desarrolladores integren código en un repositorio compartido varias veces al día. Al integrar contenido con cierta frecuencia, detectamos errores rápidamente y los localizamos de manera sencilla.

Características

ENTREGAS MÁS RÁPIDAS: La integración continua nos ayuda a entregar actualizaciones a nuestros clientes con mayor rapidez y frecuencia al realizar como práctica continuos posteos de código

AUTOMATIZACION: Construir software significa también compilarlo, testearlo, corregir errores y volver a testear. El uso de herramientas y procesos automatizados permite ahorrar esfuerzos en todos estos procesos.

MAYOR VISIBILIDAD: El simple hecho de liberar código más rápido y de forma continua, ya sea con actualizaciones o correcciones, permite a nuestros clientes ver que el proyecto se está moviendo (y no tener que esperar a su finalización para acceder a los resultados).

FACILIDAD DE DEBUGGING: Con testeos frecuentes podemos encontrar y solucionar errores mucho antes que se conviertan en problemas mayores.

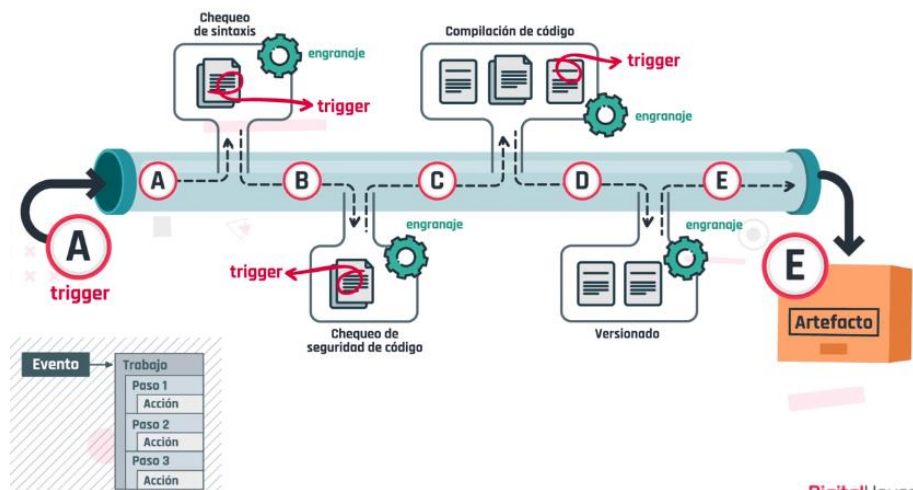
¿Qué son los triggers?

Un trigger es un “disparador”: algo que hace que una primera cosa active una segunda y así sucesivamente. Los triggers son parte de la familia de la “integración continua” o “CI” y se utilizan para ejecutar un pipeline mediante una llamada de una API u otro proceso en línea. Son sumamente útiles para concatenar procesos que, en líneas generales, constituirán un flujo de trabajo que servirá para construir algo.

Pensemos en este ejemplo: tenemos cientos de fichas de dominó dispuestas una tras otra en serie. Si le damos un pequeño golpecito a la primera ficha que se encuentra en el extremo esto generará una reacción en cadena haciendo que todas, y cada una de las piezas, comiencen a caer una tras otra. La acción de empujar la primera pieza es un trigger.

Un *trigger es un disparador de eventos*, un evento es la aparición de un estímulo que puede dar comienzo una serie estados posteriores. Un objeto pasará de un estado a otro dentro de nuestro pipeline.

Un trigger se va a usar para iniciar un pipeline de forma automática y va a decidir qué código ejecutar cuando se produce un evento específico. En general, cuando hablamos de procesos por eventos hay un bucle principal que escucha los nuevos eventos entrantes que son disparados por el trigger. Si bien esta palabra no es más que un concepto hay que destacar que su importancia subyace en lo que representa: *una forma de concatenar procesos*.



Según como un pipeline esté configurado pueden existir los siguientes tipos de triggers:

BASADOS EN CI: Estos triggers hacen que se ejecute un pipeline cada vez que se envía una actualización a un Branch específico o cuando se envían etiquetas específicas.

BASADOS EN PIPELINES: Disparan un pipeline habiendo completado un pipeline anterior.

SCHEDULED: Disparan un pipeline de acuerdo a un evento programado.

EXTERNOS: Refiere la integración con distintos productos. Generalmente mediante el uso de APIs.}

Los triggers difieren en la forma de uso, implementación y configuración según cada producto de CI. Sin embargo, su fundamento es el mismo en todos.

Artefacto

Varias veces hemos tenido que compilar código fuente para obtener un producto utilizable. Esos archivos resultantes del proceso de compilación llevan un nombre puntual: ¡Artefactos!

El artefacto es un *objeto o serie de objetos que se produce a partir de estas compilaciones*. Estos resultantes pueden, a su vez, utilizarse para una segunda y/o última compilación. Tenemos como ejemplo de ello archivos binarios: dll, jar, war, ear, msi, archivos exe, etc.

Navegando entre artefactos

Las aplicaciones suelen estar fragmentadas en distintos componentes individuales que luego se pueden ensamblar hacia un producto completo. Esto generalmente sucede durante la fase de compilación, cuando son creados fragmentos más pequeños de la misma. Esto suele hacerse para tener un uso más eficiente de los recursos, reducir los tiempos de compilación, hacer un mejor seguimiento para la depuración binaria, etc.

El tipo de artefacto o artifact va a depender del producto con el que se programe la aplicación.

Veamos el tipo de artifact por producto:

- Java: jar, ear, war, etc.
- .Net: dll, exe, etc.
- Python: .py, sdist.

También pueden existir tipos de artefactos no relacionados con el producto específico, pero necesarios para el funcionamiento. Por ejemplo, archivos, YAML, XML, TXT, MD, lib, so, bin, etc.

¡Un momento! Para complejizar un poco más el panorama, un artefacto también puede ser un script, un diagrama, un modelo de datos, etc.

Pero entonces, ¿un artifact es cualquier tipo de archivo? En términos estrictos, es todo aquello que resulta de un proceso de build.

Administración

Un producto de repositorios de artefactos nos permite básicamente efectuar las siguientes operaciones: mover, copiar, eliminar ... artefactos para mantener la consistencia en cada repo.

Cuando un artefacto se mueve, copia o elimina, el producto debe actualizar automáticamente los llamados “descriptores de metadatos”.

Ejemplo de ello pueden ser: maven-metadata.xml, RubyGems, Npm, etc.

Metadata

¡Los metadatos son datos acerca de los datos! Cada artefacto contendrá metadatos que son esenciales para la reutilización de código.

Una característica existente es la de permitir a los desarrolladores compartir código y utilizar componentes de terceros. En este sentido, los metadatos cumplen un rol clave en la colaboración. ¿Por qué?

Por ejemplo, al chequear los datos asociados a cada artefacto podemos ver si fueron alterados. Esto será de utilidad para validar su descripción, por ejemplo, la versión de un producto. Podremos

saber -acerca de ese cambio- quien lo hizo, cuándo, a qué hora exactamente, qué dependencias tiene, etc.

Sin embargo, la combinación de muchos tipos de metadatos puede llegar a complejizar el proceso de colaboración. ¡Una buena estrategia inicial es vital para evitar caer en este tipo de situaciones!

Almacenamiento

¿Dónde se almacena todo este código y sus respectivos artefactos?

En este punto está claro que el código suele almacenarse en sistemas de control de versiones como GitHub, GitLab o BitBucket.

Pero este no es el caso para los artefactos que pueden -por ejemplo- ser archivos binarios. Y quizás no tiene mucho sentido utilizar un repositorio de proyectos para almacenar archivos binarios que son ilegibles para el ser humano y pueden ser bastante grandes en tamaño.

Es aquí donde los repositorios de binarios son una parte tan vital del proceso de integración continua.

El repositorio binario puede permitir alojar todo esto en un solo lugar haciendo que su administración sea más simple. Ejemplo de productos típicos que podemos encontrar en el mercado hoy día son: Artifactory, Nexus, Harbor, etc.

También vamos a encontrar aquellos que son basados en Cloud: Azure Artifact, Artifact Registry, etc.

Accediendo a nuestros artefactos

Como todo producto de software, su sintaxis variará, pero su propiedad será la misma: ofrecer una forma sencilla de realizar consultas que especifique un criterio de búsqueda, filtros, opciones de clasificación y parámetros de salida.

Esto se logra mediante la exposición de una API RESTful, ya que siendo objetos que deban utilizarse de inmediato para proporcionar datos de salida, el tiempo de acceso y respuesta debe ser extremadamente rápido y con bajo consumo de memoria.

Todo alrededor de una filosofía

La importancia de un repo de artefactos se puede entender en relación con la filosofía DevOps: “desarrollar mejores aplicaciones, más rápido y con constantes entregas de funciones o productos de software”.

Una práctica común en integración continua es la de construir el binario una sola vez, subirlo a un repositorio de binarios y luego llamarlo desde allí para implementarlo en los diferentes entornos. De esa manera nos aseguramos de que el código base que ya funciona en el ambiente de desarrollo es la misma base que se introdujo en producción.

Conclusión

Los artefactos son subproductos de un proceso de desarrollo de software. Son los componentes con los que un software está hecho.

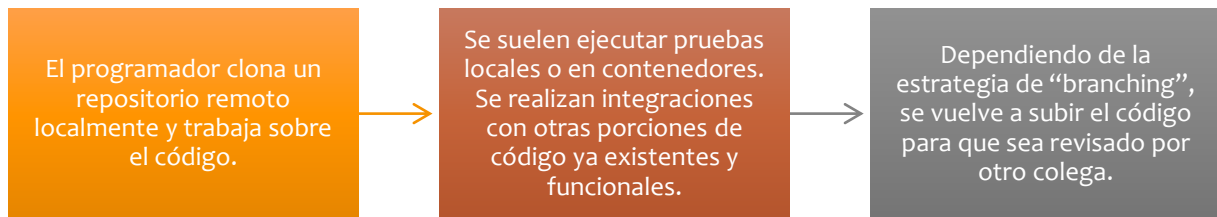
Son sumamente necesarios durante el desarrollo, las operaciones diarias, el mantenimiento y la actualización del software.

Su correcta administración es fundamental para el correcto funcionamiento de un producto gestionado mediante una filosofía de DevOps.

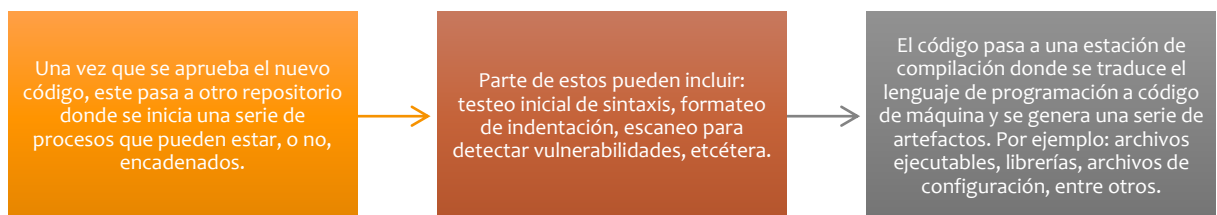
Pasos para el desarrollo de Software

Codificación

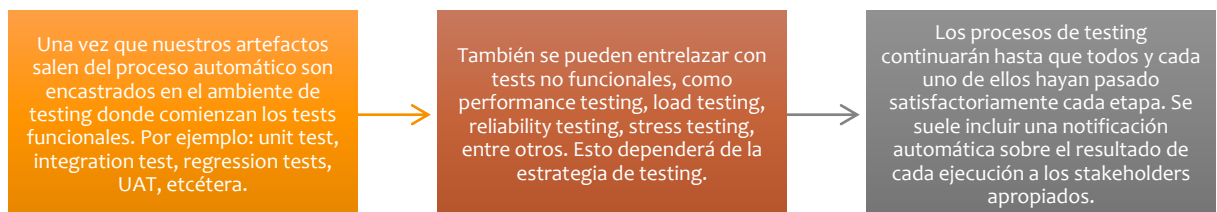
Pasamos lo que se definió y diseñó a código fuente, en el lenguaje de programación determinado:



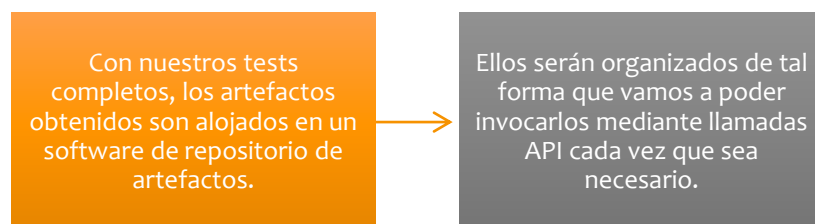
Procesamiento automático: testing inicial y compilación



Testeo



Almacenaje de artefactos



El principio de inmutabilidad

Inmutabilidad: "sin mutaciones", "sin cambios". En el sentido de DevOps, significa que una vez que creamos un artefacto —ya sea una imagen de contenedor o un paquete de código compilado— no debería ser necesario modificarlo. Y si se requieren cambios, se creará una nueva versión del objeto.

¿POR QUÉ NOS RESULTA ÚTIL QUE UN OBJETO O ARTEFACTO SEA INMUTABLE? Porque cuando lo copiamos —por ejemplo, desde un entorno de desarrollo a la producción— ya sabemos cómo se va a comportar. No solo el artefacto tiene que ser inmutable para que funcione bien en todos los entornos, o ambientes, sino también la infraestructura.

Resumen de la Semana

INTEGRACIÓN CONTINUA: Es una práctica común en el desarrollo de software. Los desarrolladores fusionan regularmente cambios de código en un repositorio central para ejecutar pruebas y compilaciones automatizadas. Facilita los ciclos de retroalimentación entre equipos de operaciones y desarrollo para que se puedan implementar iteraciones de actualizaciones más rápidamente en las aplicaciones en producción.

PIPELINE: Conjunto de prácticas que implementan los equipos de desarrollo (Dev) y operaciones (Ops) para crear, probar e implementar software de forma más rápida y sencilla. El flujo de trabajo contiene fases que impulsan el desarrollo continuo, integración, pruebas y posterior retroalimentación para comenzar de nuevo.

AUTOMATIZACIÓN: Utilización de tecnología que reduce el uso de asistencia humana sobre los procesos haciéndolos más confiables, rápidos, seguros y escalables.

TESTING: Proceso organizativo dentro del desarrollo de software en el que se verifica la corrección, calidad y rendimiento del software crítico para el negocio.

BUILD: Proceso que convierte archivos con código en un producto de software en su forma final o consumible.

TRIGGERS: Es ese “algo” que activa la ejecución de otro proceso. Un desencadenante para lograr que algo se ponga en movimiento.

ARTEFACTOS: Es uno de los muchos tipos de subproductos tangibles que se producen durante el desarrollo de software.

Clase 16: Pipelines – Realease y continuos delivery

CD: Despliegue continuo

Una vez terminada la elaboración del producto, es decir la culminación del proceso de CI (integración continua) donde ya tenemos la aplicación compilada, comienza la distribución para su consumo. Pero antes debe pasar por una serie de etapas que comprueban su calidad y aseguran su correcta entrega, estas etapas están dentro de lo que se conoce como CD o despliegue continuo.

CD hace referencia al *segmento del pipeline final* que se encarga de llevar un producto ya construido a un ambiente productivo, este proceso es continuo porque al estar en un pipeline automatizado nos permite realizar esta tarea de forma continua siempre que sea necesario. El proceso entero se compone de las siguientes etapas:

ETAPAS DE TESTING: Donde se realizan pruebas funcionales, de integración, de seguridad, de performance, de carga que comprueban el correcto funcionamiento y la calidad del producto.

ETAPA DE RELEASE: En ella se lleva a cabo la creación, la calificación del paquete y la subida y guardado en una bodega de paquetes, lo que llamamos repositorio de artefactos, donde quedará disponible para su despliegue en el entorno que se requiera.

En este paquete además de nuestra aplicación y su metadata, cómo puede ser una versión específica, también se suelen incluir archivos de configuración, programa, instalación y documentación.

ETAPA DE DEPLOYMENT: Acá se mueve el producto, se realiza el despliegue, su instalación y se pone en funcionamiento y queda listo para que pueda ser usado por los usuarios.

ETAPA DE OPERACIÓN: En esta se opera, se utiliza el producto. Aquí se debe preparar el entorno o lugar de despliegue, es decir, configurar servidores entre otras cosas.

Las etapas, como en todo el pipeline, van una detrás de la otra de manera que si alguna falla o no cumple con su cometido se aborta al proceso hasta que se corrija aquello que ocasionó el problema.

El CD nos permite entregar y garantizar un producto de calidad en el menor tiempo posible y cumpliendo con todos los requisitos de las personas usuarias.

Continuous Delivery vs Continuous Deployment

La CI/CD incorpora la automatización continua y el control permanente en todo el ciclo de vida de las aplicaciones, desde las etapas de integración y prueba hasta las de distribución e implementación. Este conjunto de prácticas es una solución para los problemas que puede generar la integración del código nuevo a los equipos de desarrollo y de operaciones.

Vamos ahora a detenernos en un momento en particular de este proceso. El Despliegue Continuo (CD) puede referirse a dos prácticas similares pero con impactos diversos:

- Continuous Delivery.
- Continuous Deployment.

¡Una diferencia clave!

Los procesos de continuous delivery (o entrega continua) y continuous deployment (despliegue continuo) siguen las mismas etapas, ¡pero con una pequeña gran diferencia!

La entrega continua se focaliza en la automatización de los pasos para que nuestro software esté disponible para ser aplicado en los ambientes productivos en cualquier momento, ¡pero no hace la implementación automática en producción!

El despliegue va un paso más allá. En esta práctica, todo es automático en los ambientes, ¡también en producción! Buscamos que no exista intervención humana en ninguno de los procesos de trabajo.

Esta es la diferencia más importante con respecto a la entrega continua.

Para lograrlo, el pipeline de producción tiene una serie de pasos que deben ejecutarse en orden y forma satisfactoria ¡y de manera automática! Si algunos de esos pasos no finalizan de forma esperada, el proceso de despliegue no se llevará a cabo.



Examinemos ambos procesos

ANTES Y DESPUES: Dependiendo de las practicas adoptadas por la organización, el despliegue puede percibirse como una etapa posterior a la entrega porque estamos impactando nuestro ambiente productivo de forma directa luego de hacer el delivery de nuestro producto.

ATENCION: Si bien ambos términos “continuos deployment” y “continuos delivery” manejan todo su ciclo de procesos de forma similar (Test y Release), la etapa final es distinta. En continuos deployment el paso al ambiente productivo es automático

MADUREZ Y CONFIANZA: Para que estos procesos corran de manera satisfactoria en nuestra organización necesitamos de un alto grado de madurez y confianza técnica de nuestro equipo de trabajo.

AUTOMATIZACION: Es el punto vital de estas prácticas porque sin ella no existirían desencadenantes o “triggers” que impacten sobre los procesos siguientes. La automatización también nos facilita los procesos lógicos de toma de decisiones o diseñar bucles para tareas repetitivas.

BENEFICIOS:

- Tiempos de entrega más rápidos
- Mayor visibilidad al liberarse código con más frecuencia
- Mejor calidad de software: hacemos cambios pequeños y podemos identificar fácilmente los problemas mediante múltiples testeos que podemos configurar antes que el producto sea lanzado en nuestro ambiente productivo



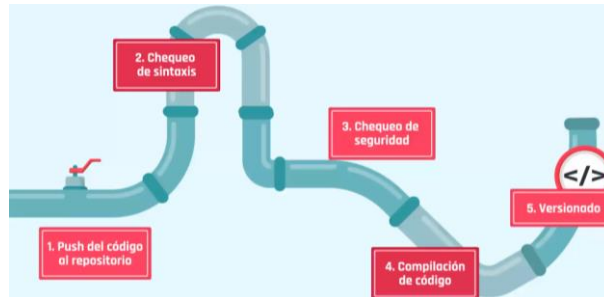
Ilustración 1: ¿Entrega o despliegue?

Clase 17: Pipelines – End to End

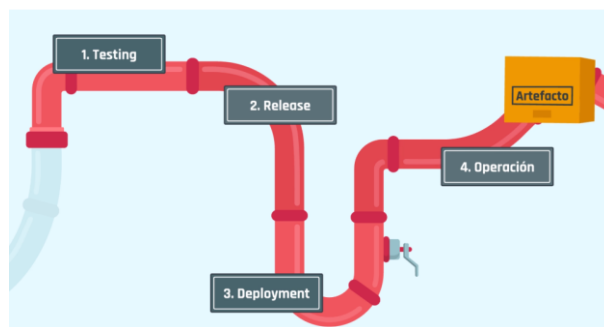
Pipelines: CI/CD

Imaginemos una fábrica que produce un bien físico. Necesita materia prima, ciertas máquinas, personal que las controle y procesos que definen, de forma lógica, la secuencia de pasos que se deben ejecutar para lograr el producto que finalmente saldrá de la fábrica y llegará a algún mercado para su consumo, una vez terminada la elaboración del producto comienza su distribución. Pero antes debe pasar por una serie de etapas que comprueben su calidad y se aseguren de la correcta entrega a los usuarios que van utilizar dicho producto. Todo este proceso automatizado de principio a fin, dentro de la elaboración de un producto, es la unificación de dos conjuntos de pipelines que garantizan su calidad y tiempo de entrega desde la creación del código hasta su operación: CI y CD.

Iniciemos con *CI (integración continua)* que no es más que la práctica de fusionar o mergear copias de trabajos de los proyectos de cada desarrollador en un proyecto principal. La idea es realizar integraciones de código constantemente y quizás, varias veces al día dependiendo del enfoque estratégico de la compañía. Por cada integración pueden existir uno o más testeos para detectar posibles errores lo más rápido posible.



Luego seguimos con CD (Entrega Continua) que es el eslabón procedural que le sigue a CI. Este es un enfoque o práctica en ingeniería software en el que los equipos producen software en ciclos cortos. Esto va a granizar que el software se pueda lanzar o entregar de manera confiable en cualquier momento y de forma automatizada. Su objetivo es crear, probar y lanzar software con mayor velocidad y frecuencia.



Si bien el proceso CI/CD “finaliza”, al estar operativo el producto, en realidad el ciclo vuelve a comenzar. El uso de CI/CD nos aporta los siguientes beneficios:

- Los cambios de códigos son más pequeños y más simples, más atómicos y tienen menos consecuencias no deseadas.
- Detectar errores se hace más simple y rápido.
- El tiempo medio de resolución (MTTR) es más corto justamente debido a cambios de código más pequeños
- Los cambios más pequeños permiten pruebas positivas y negativas más precisas
- El tiempo transcurrido para detectar y corregir los escapes de producción es más corto
- Tasa de liberación mucho más rápida
- El producto mejora a través de la rápida introducción de funciones y una respuesta a los cambios de funciones más veloz.
- Los ciclos de lanzamiento y producción son más cortos.

- La participación y los comentarios del usuario final durante el desarrollo y el testeo conducen a mejorar la usabilidad del producto final (retroalimentación continua)

Hoy en día es común ver que las áreas de operaciones y desarrollo trabajan en conjunto en el desarrollo de un producto de software, lo que se conoce formalmente como prácticas de DevOps. En este sentido, CI/CD son los pilares fundamentales de estas prácticas que trabajan sobre los principios de integración continua, entrega continua y desarrollo a través de varias pruebas automatizadas que garantizan un producto de calidad y su correcta entrega.



Ilustración 2: Ciclo de Desarrollo usando pipelines en CI/CD

Combinamos Deployments

En términos generales la palabra deploy es utilizada para describir que algo fue colocado en su posición, cuando un sistema es habilitado para su uso ya sea en un ambiente de desarrollo, en uno para realizar pruebas o, en producción. Pensemos en el caso de haber creado un sitio web en la computadora y lo dejemos estático, sin publicar. En el momento en que incorporamos la página a un servidor de web Hosting ese proceso será considerado como deploy.

Ahora bien, no debemos confundir el término implantar con implementar porque son cosas muy distintas. Cuando decimos **implantar** hacemos referencia a iniciar algo, mientras que **implementar** es el acto de poner algo en práctica. Si lo aplicamos al mundo de la programación cuándo se inicia el desarrollo de un sistema este está siendo implantado, en el momento en que el proyecto comienza a ser usado por los usuarios podemos decir que ha sido implementado.

¿CUÁLES SON LAS FORMAS DE REALIZAR DEPLOY? Hoy existen básicamente 3 formas realizarlo:

1. **MANUAL:** Por ejemplo, el protocolo de transferencia de archivos conocido como FTP. Se trata de un tipo de conexión que permite que dos computadoras con acceso a internet intercambien archivos. Al realizarlo de forma manual se necesita una persona ejecutando esta transferencia.
2. **PARCIALMENTE AUTOMATIZADO:** un ejemplo puede ser el Git push que se realiza en el repositorio git el cual opera un pequeño Trigger y actualiza el servidor de web Hosting. Aunque necesita algunos comandos, el proceso ocurre de manera automática. su ventaja es el control de la versión y el estado de cada deploy.
3. **COMPLETAMENTE AUTOMATIZADA:** Es la tendencia más moderna en el desarrollo web. Esta no solamente copia los cambios de forma automática en el servidor, sino que también está íntimamente conectada con el concepto integración continua. La herramienta de deploy,

en este caso, realiza todas las pruebas necesarias para que no existan problemas a la hora de juntar las integraciones de la producción. Una de las herramientas más utilizadas para el deploy automatizado es Jenkins.

Entre sus beneficios destacan el alto nivel de productividad, la seguridad que nos brinda y la calidad en el desarrollo de software.

Existen también tres estrategias muy simples que puede ser implementadas y que permiten realizar deploy en el día a día:

1. **ROLLING:** Consiste en subir los servicios con la nueva versión del código, pudiendo coexistir o no con la versión antigua.
2. **BLUE-GREEN:** Se caracteriza por tener dos ambientes idénticos conocidos como *mirror* que tienen un load balancer (balanceador de carga) permitiendo así, redireccionar el tráfico para el ambiente deseado. El beneficio de esta estrategia es que nos permite subir una nueva versión de la aplicación que está en producción mientras que la versión actual, Blue, solo recibe las solicitudes. De esta forma tan pronto como sean realizadas las pruebas en la nueva versión, Green, es posible realizar otras solicitudes que apunten hacia esta.
3. **CANARY:** Se trata de la estrategia más compleja, consiste en colocar la nueva versión en producción para una pequeña parte de los usuarios. Es posible, por ejemplo, liberar una función solamente para el público de sexo femenino menor de 30 años permitiéndonos probar la versión antes de liberar el acceso total.

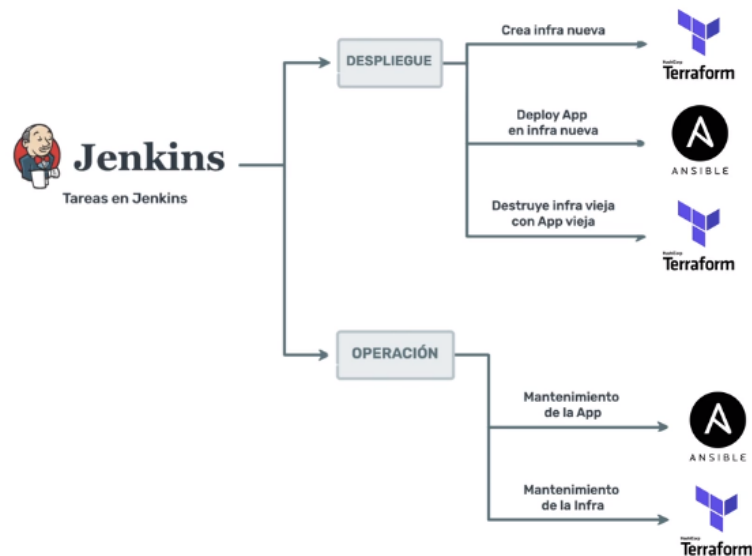
¿CÓMO INTEGRAMOS IAC HACER DENTRO DEL CICLO DE DEVOPS DE NUESTRA APLICACIÓN? Los equipos de infraestructura utilizan pipelines para IAC por las mismas razones que los equipos de desarrollo usan pipelines para su código de aplicación, ya que garantizan que cada cambio se haya aplicado a cada entorno y que las pruebas automatizadas hayan pasado.

Con este enfoque las nuevas instancias del entorno pueden activarse bajo pedido lo cual tiene varios beneficios:

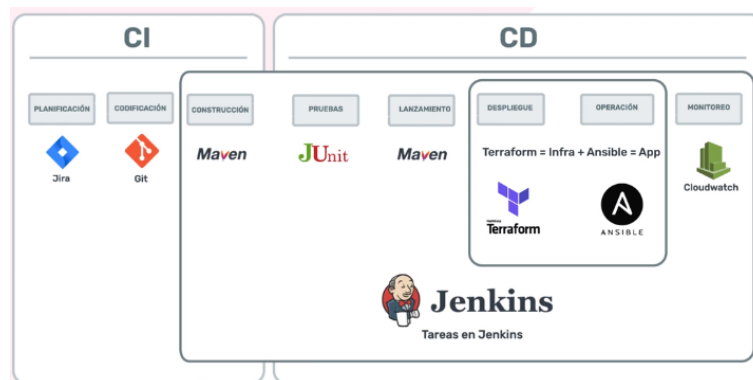
1. Los desarrolladores pueden crear sus propias instancias de sandbox para que puedan implementar y probar aplicaciones basadas en infraestructura en la nube o, trabajar en los cambios de las definiciones de entorno sin entrar en conflicto con los otros miembros del equipo.
2. Los cambios y las implementaciones se pueden manejar con enfoque Blue-Green. Crea una nueva instancia, la prueba, luego se intercambia el tráfico y se elimina la estancia anterior
3. Quiénes prueban, revisan y otros miembros del equipo pueden crear entornos según sea necesario eliminándolos cuando no se utiliza.

De esta manera tenemos integrado la gestión de la infraestructura como código dentro de nuestro ciclo DevOps donde Jenkins administra las tareas que, para este caso, serán el deploy y la operación como etapas del ciclo.

Jenkins interactúa con ambas herramientas para que cada una realice la tarea necesaria. Dichas herramientas devuelven una respuesta si la tarea finalizó correctamente o no y según lo que tenga definido en el pipeline continua o lo finaliza.



Es importante conocer el ciclo de vida de una aplicación desde su planificación hasta su despliegue y operación teniendo presente cuáles herramientas se utilizan para cada una de las etapas.



¿Cómo hacer un pipeline de CI/CD exitoso?

Construir un pipeline completo de un extremo a otro no es tarea sencilla, ya que involucra multitud de procesos y herramientas. Por eso es bueno que podamos generar y tener siempre presente estas buenas prácticas para el trabajo en DevOps.

FALLAR RÁPIDO: Provocar que el pipeline falle ni bien se detecta un problema en la etapa, generando un feedback a los desarrolladores para que resuelvan el problema sin perder tiempo.

TENER ETAPAS BIEN DEFINIDAS: Cada etapa del pipeline debe representar de manera clara un proceso de CD/CI para que sea mas fácil medir y analizar su funcionamiento y mejorar o detectar problemas.

HACER PRUEBAS FUERTES: Dedicar tiempo para que las pruebas automatizadas sean exhaustivas, ya que serán responsables de la calidad de nuestra entrega. ¡No queremos que un producto fallido llegue a producción!

ACTIVAR EL ROLLBACK: En caso de que llegue un producto fallido a producción, mediante el rollback podemos volver atrás los cambios y desplegar una versión anterior y sin fallas.

BAJAR LA DEPENDENCIA: Eliminar del pipeline cualquier factor externo que no podamos controlar. Por ejemplo, si al hacer un test tenemos que hacer una consulta a una base de datos, es conveniente

generar una base de datos efímera y no depender de una ajena. La mejor práctica, aunque difícil de conseguir, es levantar todo el entorno de prueba en cada corrida de pipeline y luego destruirlo.

EJECUTAR FÁCILMENTE: Un pipeline no puede ser complejo de lanzar. El desarrollador tiene que poder ejecutarlo sin esfuerzo. Por ejemplo, es posible automatizar el lanzamiento al aprobar un nuevo código en un gestor de versiones.

CONTAR CON CREDENCIALES SEGURAS: Gestionar usuarios y contraseñas en depósitos que permitan resguardar estos datos sensibles. Ofuscar logs en donde puedan aparecer.

Clase 18: Cierre de la semana 6

Resumen de la semana

CD: Existen dos términos clave cuando hablamos de CD y nos sirven para revisar cómo se lanzan releases de software a un entorno de producción: la entrega continua (continuous delivery) y el despliegue continuo (continuous deployment).

ENTREGA CONTINUA: se produce software en ciclos cortos, lo que garantiza que se pueda lanzar de manera confiable en cualquier momento. Su objetivo es crear, probar y lanzar software con mayor velocidad y frecuencia.

DESPLIEGUE CONTINUO: El software se entrega con frecuencia en producción a través de implementaciones de pipelines totalmente automatizadas de inicio a fin.

EL PROCESO DE RELEASE: Este proceso consta de una serie de acciones que nos llevan a tener disponible un paquete completo. En esta instancia es importante enfocarse en las múltiples versiones ⁵y en las etapas que lo componen.

ARQUITECTURA RECOMENDADA SOBRE PIPELINES: Con el objetivo de no “reinventar” la rueda, hemos visto algunas prácticas fuertemente testeadas en el mundo de DevOps que nos ayudan a arquitecturar un pipeline de forma eficiente:

- Fallar rápido.
- Tener etapas bien definidas.
- Hacer pruebas fuertes.
- Activar el Rollback.
- Bajar la dependencia
- Ejecutar fácilmente.
- Contar con credenciales seguras

PIPELINES END-TO-END: Hablamos del conjunto completo de procesos: CI+CD. Los pipelines abarcan flujos de trabajo que van coordinando las diversas etapas. Todo esto se define en archivos de configuración de proyecto e involucra triggers.

Clase 19: Monitoreo – Introducción

Módulo 4: Monitoreo

¿Qué significa monitorear?

Monitoreo es el proceso que permite recopilar métricas sobre las operaciones en aplicaciones e infraestructura para garantizar que todo funcione correctamente. El término monitoreo puede

⁵ <https://sites.google.com/site/practicadesarrollosoft/temario/scm/versionado-y-staging-de-componentes>

estar asociado a diversas aristas el mundo IT como la infraestructura, las aplicaciones y el negocio. En todos los casos el objetivo que se persigue es siempre el mismo:

- Seguimiento de parámetros críticos
- Tener observabilidad sobre lo que ocurre nuestro ambiente o plataforma
- Registro, almacenamiento y exportación de datos para su posterior análisis y/o procesamiento
- Prevenir problemas y, si ya estamos en problemas, comprender cómo podemos solucionarlos.

La diferencia está en que cuando hablamos de monitoreo de infraestructura generalmente apuntamos a servidores, redes, bases de datos, etcétera. Cuando lo hacemos sobre aplicaciones hablamos de todo un software que nos brinda un servicio y cuando hablamos de negocios es sobre la utilización para la toma de decisiones.

Para que el monitoreo se puede hacer efectivo necesitamos contar con diferentes tipos de herramientas tecnológicas que nos permiten realizarlos:

1. **HERRAMIENTAS DE OBSERVACIÓN:** Nos ayudan a supervisar la efectividad operacional del Software, Hardware y las distintas aplicaciones que se están utilizando.
2. **HERRAMIENTAS ANALÍTICAS:** Una vez que se obtienen datos sobre el desempeño de los activos digitales, las aplicaciones nos ayudan a analizar la información para encontrar problemas y revisar cómo y porque están ocurriendo.
3. **HERRAMIENTAS DE ENGAGEMENT:** Nos facilitan tomar acciones concisas basadas en las observaciones y los análisis que se han realizado con las herramientas anteriores.

¿CÓMO FUNCIONAN ESTÁS HERRAMIENTAS? En la mayoría de los casos la herramienta va a dividirse en dos partes: por un lado, un *agente* que se instala en el recurso que vamos a monitorear y por el otro lado, una *consola de monitoreo y/o administración*. El agente cumplirá la función de escuchar los eventos que queramos para enviarlos, luego, a la consola de monitoreo y poder visualizar la captura efectuada ya sea por medio de gráficos, histograma, áreas, etcétera.

¿CÓMO CONTROLAMOS NUESTROS SISTEMAS? Mediante las métricas vamos a conseguir monitorear, gestionar, optimizar y generar informes de todos nuestros sistemas. Para esto tenemos que definir nuestra necesidad. Por ejemplo, que para una carga de 100 usuarios concurrentes los recursos hardware de un servidor web no superen el 60% de su capacidad de uso, luego nos aseguramos de que nuestro sistema funcione bien bajo los parámetros establecidos, en caso de que no funcione podemos ajustarnos agregando más recursos.

Por último, debemos establecer las alertas que serán enviadas en caso de que se rompan ciertos umbrales.

¿Vamos por buen camino? ¿Por qué? ¿Cómo y en que deberíamos modificar nuestra intervención para obtener los resultados esperados? Las respuestas a estas preguntas se responden monitoreando.

Monitoreo es un proceso sistemático que se encarga de recolectar, analizar y utilizar información con el objetivo de mantener de manera óptima los recursos computacionales y el software de acuerdo a los requerimientos del negocio.

¡El monitoreo es un proceso clave para garantizar que todo funcione de la manera que se espera y necesitamos! Nos da información para saber si estamos avanzando en el sentido esperado y nos alerta acerca de cambios o correcciones que deberíamos hacer. En definitiva, es un proceso indispensable al implementar un proyecto de sistemas. Cuanto más grande sea el proyecto, ¡más importancia tendrá!

¿Qué buscamos monitorear?

¿Para qué monitoreamos? ¿Qué beneficios nos brinda esta práctica? ¿Podemos planificar y diseñar el mejor monitoreo para nuestro proyecto?

Como venimos descubriendo, el monitoreo se trata de generar información para poder controlar y verificar que nuestros procesos marchan de acuerdo a lo planificado.

Desarmando esta idea, podemos identificar cuatro objetivos o propósitos que el monitoreo viene a cubrir:

- Evitar y/o prevenir los problemas que puedan surgir ¡Y poder anticiparnos!
- Entender qué está sucediendo en tiempo real en nuestros recursos y mantenerlos siempre alineados a nuestras necesidades.
- Hacer análisis porque nos permite almacenar eventos para una revisión posterior.
- Observar y rendir cuentas comunicando al personal interviniente.

Como podemos ver, el monitoreo nos guía en la toma de decisiones de gestión, nos permite generar reportes de desempeño y crear indicadores de uso y rendimiento.

Beneficios del monitoreo

OBSERVAR INFRAESTRUCTURA Y APLICACIONES: Las aplicaciones modernas están montadas y se ejecutan mediante todo tipo de arquitecturas (monolíticas, distribuidas, de microservicios, entre otras) que generan grandes cantidades de datos en forma de métricas, registros y eventos. Mediante las herramientas de monitoreo podemos recopilar, visualizar y correlacionar en un único punto los datos de todos los recursos, aplicaciones y servicios que funcionan en servidores en la nube y datacenters.

RECOPILAR MÉTRICAS PARA SU POSTERIOR ANÁLISIS: Monitorizar recursos y aplicaciones puede efectuarse en tiempo real y ser consumido mediante una consola centralizada que formatee y presente esos datos de forma legible. También puede hacerse mediante el uso asincrónico, es decir, recopilación y almacenamiento para su uso posterior.

MANIPULAR DATOS PROCESABLES: Las herramientas actuales de monitoreo nos permiten exportar los datos crudos, también llamados en inglés raw data, para poder ser graficados, almacenados y/o procesados por otros motores de datos con el fin de obtener la visualización que necesitemos.

ALCANZAR RESULTADOS ESPECÍFICOS: Cada proyecto se construye bajo la influencia de diversas interacciones. El monitoreo es clave en la gestión de cada proyecto y nos ayuda a alcanzar los objetivos tomando en consideración la evolución del contexto, las estrategias, las hipótesis, las suposiciones, etc.

MEJORAR EL RENDIMIENTO OPERATIVO: Al poder establecer alarmas y automatizar acciones en función de umbrales predefinidos, podemos saber con exactitud cuándo un recurso o serie de recursos de

mi plataforma operativa se están desalineando para poder tomar acciones correctivas y volverlos a la normalidad.

VISIBILIZAR LAS OPERACIONES DIARIAS: Tener una vista operativa unificada nos permite optimizar el rendimiento y el uso de recursos. Al disponer de datos detallados, y en tiempo real, operadores, gerentes de proyecto y stakeholders podrán actuar y tomar decisiones de acuerdo a la información que reciben.

Monitoreo de infraestructura y de aplicaciones

Antes de comenzar con un proceso de monitoreo es clave que nos preguntemos qué es lo que queremos ver, analizar, comprender, medir, o acerca de qué nos interesa estar alertados. Pensemos... ¡No es lo mismo medir recursos hardware que servicios de software, procesos, o la comunicación de una API!

A la hora de definir el monitoreo de nuestro ambiente, existen dos grandes separaciones que tenemos que considerar:

- Por un lado, la **infraestructura**. Es usual que necesitemos monitorear recursos como sistemas de almacenamiento, redes, bases de datos, servidores, etc.
- Por otro lado, vamos a encontrar **aplicaciones**. Allí nos va a interesar monitorear elementos como el número de peticiones o requests, la cantidad de fallos producidos por un servicio web, eventos de llamadas API, etc.

Herramientas para monitorear

El mercado nos ofrece una gran variedad de productos para considerar. Como venimos conversando, nuestra decisión va a depender de qué queremos monitorear, el costo, la complejidad y la curva de aprendizaje.

Todas las herramientas de monitoreo van a cumplir las mismas funciones sobre aplicaciones e infraestructura.

- Trazabilidad y observabilidad en tiempo real.
- Alertas.
- Almacenar para su posterior análisis.
- Métricas integradas.
- Reportes.

No obstante, cada herramienta ofrecerá prestaciones específicas dependiendo de su versión.

Datadog⁶

Plataforma de seguridad y monitoreo para aplicaciones en la nube. Reúne seguimientos, métricas y registros de un extremo a otro para que aplicaciones, infraestructura y servicios de terceros sean completamente observables. Está orientada a proveer servicios en la nube.

Puntos fuertes:

- Madura integración con distintos tipos de software mediante el uso de APIs.
- Gran flexibilidad y versatilidad para configurar de diversas formas los tableros de control.
- Granularidad extrema sobre los servicios a elegir para monitorear.

⁶ <https://www.datadoghq.com/product/>

Nagios⁷

Sistema de monitorización de redes ampliamente utilizado y de código abierto. Vigila los equipos (hardware) y servicios (software) que se especifiquen, alertando cuando el comportamiento no sea el deseado. Mayormente utilizado en ambientes on-premises, aunque dispone de servicios adicionales que apuntan a tener presencia en la nube.

Puntos fuertes:

- Descubrimiento automático de recursos sin uso de agentes.
- Sistema de ticketing.
- Monitoreo ambiental.

Prometheus⁸

Base de series de tiempo y un sistema de monitoreo y alertas. Las series de tiempo almacenan datos ordenados cronológicamente, midiendo variables a lo largo del tiempo. Las bases de datos enfocadas a series de tiempo son especialmente eficientes en almacenar y consultar estos datos. Dispone de funcionalidad para cloud y on-premises.

Puntos fuertes:

- Modelado de datos.
- Lenguaje PromQL.

Cloudwatch⁹

Servicio de monitorización y administración que suministra datos e información procesable para aplicaciones y recursos de infraestructura locales, híbridos y de AWS. Permite recopilar y obtener acceso a todos los datos de rendimiento y operaciones en formato de registros y métricas a partir de una sola plataforma. AWS Nativo, con lo cual se integra fácil y naturalmente con cualquier servicio de AWS.

Puntos fuertes:

- Correlación de registros con métricas.
- Métricas de flujo.
- Análisis de registros mediante inteligencia artificial.

La importancia de las métricas

A la hora de monitorear nuestro ambiente surge una pregunta fundamental: ¿Qué métricas debo considerar para saber el estado de mi infraestructura y de las aplicaciones que he montado sobre ella?

¿Qué es medir?

La medición es el proceso que se basa en comparar una unidad de medida preestablecida con el elemento cuya magnitud se desea medir y, de esta forma, comprender cuántas veces la unidad está contenida en esa magnitud.

¿QUÉ NOS PROPONEMOS? Definir una estructura coherente de métricas a monitorear para poder gestionar, optimizar y generar informes de todos nuestros servicios de forma regular.

⁷ <https://www.nagios.com/products/nagios-xi/#features>

⁸ <https://prometheus.io/docs/introduction/overview/#features>

⁹ <https://aws.amazon.com/es/cloudwatch/features/>

¿Qué son los indicadores?

Son el resultado de manipular las métricas para obtener información sobre el comportamiento basado en diferentes variables. Tanto las métricas como los indicadores nos pueden dar información para tomar decisiones de negocio relacionadas con las funcionalidades del producto o la plataforma donde se encuentra.

¿QUÉ NOS PROPONEMOS? Convertir datos crudos en indicadores nos permitirá conocer nuestro ambiente, controlarlo y medir los recursos eficientemente, poder anticiparnos al mercado y optimizar el esfuerzo en el desarrollo de productos.

Métricas y degradación

¿Qué es la degradación?

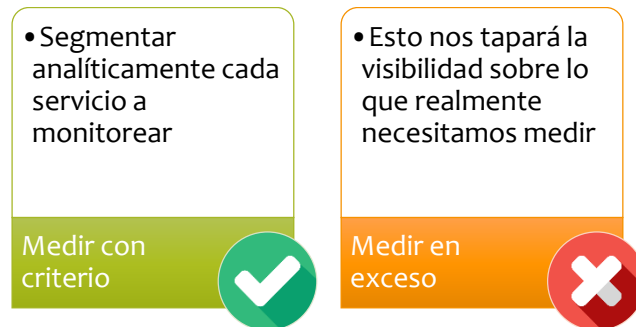
Es la desviación de las métricas predefinidas que se suelen manifestar mediante la baja performance de un servicio. El tiempo y el uso son factores que inciden directamente sobre la performance de nuestro ambiente.

Esta desviación en las métricas configuradas nos ayudará a identificar problemas en nuestros sistemas con suficiente tiempo antes de que se produzcan problemas mayores.

¿Qué hacer cuando nuestros sistemas tienen degradación?

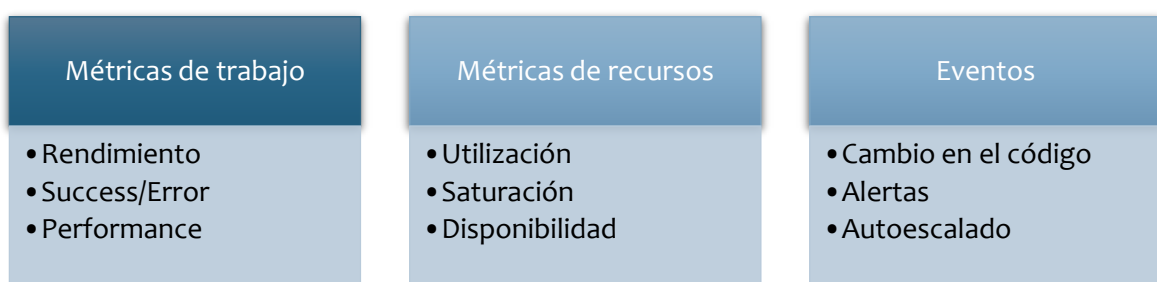
- Evaluar los recursos disponibles vs. lo consumido en función de los límites preestablecidos.
- Disponer de mapas de arquitectura que muestran el flujo de datos dentro del sistema.
- CDM (CPU, disco y memoria) son recursos esenciales a chequear en nuestras mediciones.
- Revisar logs e historial sobre cualquier cambio acontecido recientemente.

Buenas y malas prácticas



Tipos de métricas

Podemos reconocer tres tipos de categorías:



Métricas de trabajo

RENDIMIENTO: La cantidad de trabajo que realiza el sistema por unidad de tiempo. El rendimiento generalmente se registra como un número absoluto.

SUCCESS/ERROR: Las métricas que representan el porcentaje de trabajo que se ejecutó correctamente y las que han tenido fallo.

PERFORMANCE: Es la cuantificación de la eficiencia con la que un componente está haciendo su trabajo. Se suele comparar con medidas preestablecidas.

Métricas de recursos

UTILIZACIÓN: Es el porcentaje de tiempo que un recurso está ocupado o en uso.

SATURACIÓN: Es la medida de la cantidad de trabajo que el recurso aún no puede atender, a menudo en espera o queue.

DISPONIBILIDAD: Representa el porcentaje de tiempo que el recurso tiene para ofrecer a las solicitudes.

Eventos

CAMBIO EN EL CÓDIGO: Involucra todo tipo de cambio en el código: modificaciones, compilaciones y/o fallas.

ALERTAS: Eventos generados en función de algún parámetro preestablecido sobre algún recurso.

AUTOESCALADO: La adición y/o sustracción automática de recursos en función a la carga u otra configuración preestablecida.

Clase 20: Monitoreo – Monitoreando

Monitorear infraestructura

Dato de la realidad: las organizaciones dependen cada día más de herramientas que se apoyan en infraestructuras digitales. ¿Gestionar y monitorear estos recursos es cada vez más importante para el correcto desempeño de un proyecto o negocio!

La infraestructura está compuesta por una serie de elementos usualmente ligados al hardware. Pero, como venimos viendo, con la utilización de software de virtualización el límite entre hardware y aplicaciones o servicios es cada vez más difuso.

¿Qué podemos monitorear? Estos son algunos de los recursos en hardware físico o virtual:

- Redes: enrutadores, switches, firewalls, vpc, balanceadores, etc.
- Máquinas o servidores: CPU, memoria, almacenamiento, autoescalado de instancias.
- Servicios administrados: dependerá del tipo de servicio la medición a realizar.

Amazon CloudWatch

CloudWatch es un servicio de monitoreo y administración de AWS que nos permite centralizar todos los datos generados por infraestructuras, servicios y aplicaciones locales, en la nube, e híbridos.

Tiene la capacidad de generar alarmas programables, eventos que desencadenan acciones automáticas, y exponer estos datos para su análisis.

¿Cómo trabaja?

CloudWatch nos propone un flujo de trabajo con las siguientes partes:

1. **COLECTA:** Se recolectan los datos a visualizar o analizar. CloudWatch permite datos en formato de métricas, registros y eventos.
2. **ACCIÓN:** Luego de la colecta es posible establecer actuadores como alarmas o disparadores de acciones.
3. **MONITOREO:** Los datos recolectados se almacenan para ser mostrados en paneles visuales o dashboards.
4. **ANÁLISIS:** Por último, nos brinda una serie de herramientas para poder realizar análisis sobre los datos recolectados.

Obtención de datos

Existen tres tipos de datos

MÉTRICAS Existen tres tipos de datos: Valores de magnitudes medibles con una firma de tiempo e identificador.

REGISTROS: Trozos de texto que representan una información (mensajes de consola de una app).

EVENTOS: Notificaciones sobre cambios en la infraestructura, servicios externos o alarmas activadas.

Amazon cuenta con servicios diseñados para la obtención de datos y su envío a CloudWatch. ¡Solo hay que configurarlo! También ofrece la posibilidad de subir datos mediante su agente (aplicación que podemos instalar) o a través de su propia API. Dentro de este servicio vamos a acceder a distintas secciones: paneles, logs, métricas, eventos, etc., en las que podremos visualizar la información que necesitemos.

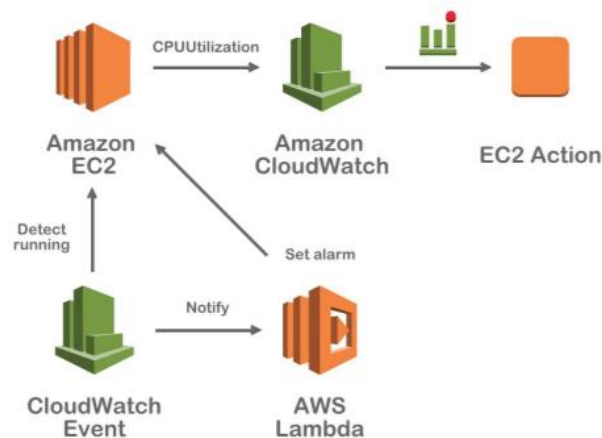
Monitoreo

CloudWatch -como toda buena herramienta de monitoreo- nos permite crear paneles a medida, gráficos e indicadores que visibilicen mejor el estado de los recursos. También tenemos disponibles paneles por defecto para la mayoría de los servicios existentes en AWS.

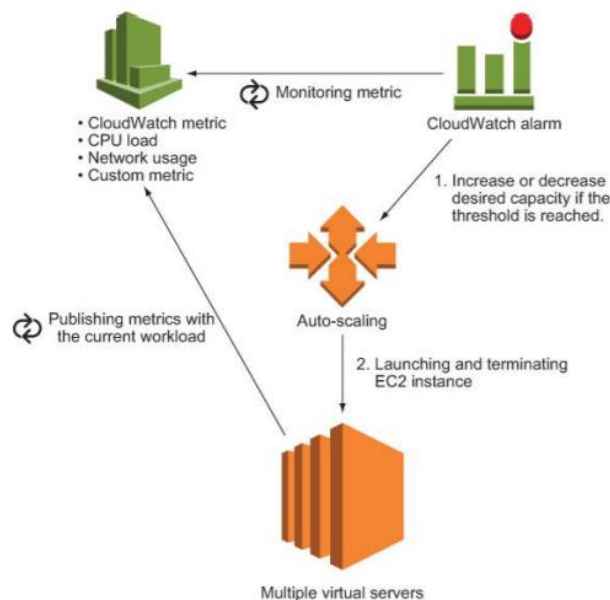
CloudWatch nos facilita crear alarmas simples -que se configuran con umbrales específicos de acuerdo a la necesidad- y alarmas compuestas, activadas por un conjunto de otras alarmas. Entre otras características posee detección de anomalías, utilizando aprendizaje automático para determinar comportamientos poco comunes.

Acciones

Mediante CloudWatch podemos *disparar acciones automáticas* utilizando tanto las alarmas ya vistas como eventos. Los eventos pueden ser propios de AWS -como cambios en recursos, por ejemplo- se pueden programar para que se disparen cada un determinado periodo de tiempo, como un despertador.



AUTOESCALADO: Un ejemplo es el autoescalado de recursos: cuando se detecta un alto consumo de CPU, se dispara la acción de escalar horizontalmente (hacer crecer) la cantidad de máquinas virtuales para satisfacer la demanda. A su vez, si el consumo es mínimo, podemos hacer decrecer la cantidad de servidores.



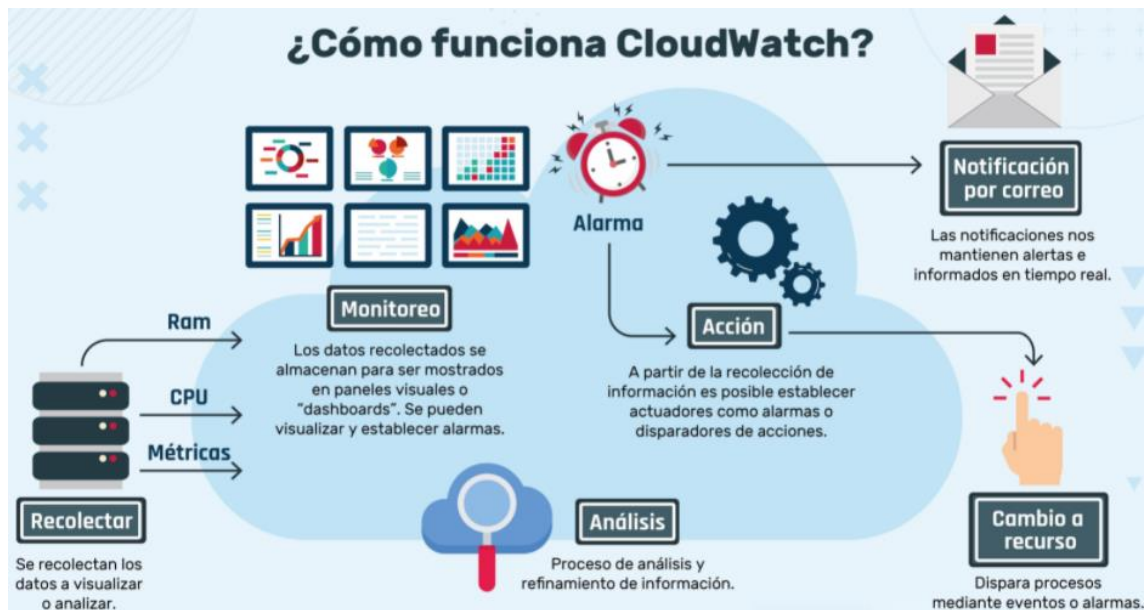
Análisis

CloudWatch ofrece operaciones sobre métricas, para obtener información en tiempo real y analizarla en un panel gráfico.

Mediante CloudWatch Logs Insight permite analizar los logs, realizar consultas con filtros y exportar a los paneles obteniendo una visibilidad operativa completa.

Para tener en cuenta:

- Almacena hasta 15 meses de métricas.
- Recolecta de datos en intervalos de hasta un segundo



Autoescalado y Elasticidad

El **autoescalado** es la posibilidad dimensionar nuestra infraestructura de manera responsiva ante la demanda de los clientes basándonos en la elasticidad de nuestro proveedor. Donde la **elasticidad** no es más que la capacidad de los servicios en la nube de entregar o eliminar recursos automáticamente, proporcionando la cantidad justa de activos para cada proyecto.

En una tienda física para disminuir el tiempo de espera entre una compra y otra, en un momento de gran demanda, lo único que podemos hacer es habilitar más cajas o puntos para pagar. Es algo que dependerá del espacio del lugar también y lo más importante, no podemos habilitar o deshabilitar estas cajas o puntos de pago de forma dinámica cada vez que la demanda sube o baja.

En una tienda online si podemos, siempre y cuando su infraestructura se base en un proveedor en la nube que sea elástico y pueda autoescalar nuestros recursos. Nuestra infraestructura base está realizada con un balanceador de carga y detrás dos servidores web, esa infraestructura es suficiente para sostener una demanda habitual. Ahora ¿qué pasa con esa infraestructura en momentos de mayor carga? Se generan demoras, o caídas perdiéndose ventas ¿no? El autoescalado propone un **crecimiento horizontal de los recursos** a medida que estos son más demandados. Entonces, de manera reactiva, se agregan más servidores al esquema de balanceo, siempre monitoreando la performance de nuestra tienda y actuando en consecuencia.



El autoescalado nos permite que este esquema de monitoreo-crecimiento se realiza de manera automática ya que AWS se encargará de monitorear determinados parámetros que le indiquemos y, en función de ellos, nuestra infraestructura crecerá o decrecerá según la necesidad. Manteniendo una infraestructura lo más ajustada posible evitando costos innecesarios.

Un producto o servicio autoescalable proporcionado por la elasticidad de nuestro proveedor de servicios en la nube es fundamental para evitar demoras y caídas por falta de recursos. Además, los

gastos de nuestra infraestructura crecerán solamente cuando la demanda pida, evitando que existan recursos ociosos y un aumento innecesario de los costos.

Clase 21: Cierre de la semana

Resumen de la semana

MONITOREO: El monitoreo es una tarea periódica que permite documentar y utilizar resultados, procesos y experiencias como base para dirigir la buena toma de decisiones de forma continua.

PIPELINES Y MONITOREO: El ciclo CI/CD no abarca solamente el planear, codificar, testear, deployar y liberar, sino también el monitoreo, práctica sin la cual nuestro ambiente no podría mantenerse en pie por los errores que puedan aparecer.