Platform Engineering

Índice

- 14.12 Comunidad y Ecosistema de Backstage

1. Introducción: De DevOps a Platform Engineering

DevOps (15 años de evolución)

DevOps es la filosofía y práctica que busca automatizar el ciclo completo desde el desarrollo hasta la operación, buscando agilidad, eficiencia y entregas frecuentes y seguras.

Objetivo principal: Automatizar todas las etapas del proceso entre Desarrollo (**DEV**) y Operaciones (**OPS**).

Etapa automatizada	Enfoque	Resultado esperado
Build	Desarrollo ágil	Artefacto listo para desplegar
Test	Integración continua (CI)	Informe de pruebas automáticas
Release	Entrega continua	Artefacto entregable directamente al cliente
Deploy	Despliegue continuo (CD)	Aplicación funcionando en producción

2. Automatización: El núcleo de DevOps

Automatizar significa:

> Crear una máquina, o modificar el comportamiento de una existente mediante un programa, que realice tareas que previamente hacía un humano.

Esto implica construir software específico que automatice tareas manuales, repetitivas y propensas a errores.

Ejemplos de herramientas por fase:

Fase	Herramientas comunes
Build	Maven, Gradle, NPM, Yarn, MSBuild

Fase	Herramientas comunes
Test	JUnit, NUnit, Selenium, Cucumber, ReadyAPI, SonarQube
Release	Helm, Spinnaker, Nexus, Artifactory
Deploy	Kubernetes, Docker, Terraform, Ansible, Puppet, Chef

Segundo nivel de automatización: CI/CD

Son plataformas que orquestan estas automatizaciones individuales:

- Jenkins
- GitLab CI
- Azure DevOps
- GitHub Actions
- CircleCI
- Travis CI
- Bamboo
- TeamCity

Estas herramientas permiten encadenar procesos (pipelines) que ejecutan scripts automáticamente al detectar cambios en el código o en la infraestructura.

3. Problemas actuales derivados del enfoque tradicional de DevOps

Escenario actual:

Históricamente, cada proyecto crea sus propios scripts para automatizar tareas, asumiendo necesidades únicas. Además, muchos scripts son desarrollados por personas con limitados conocimientos en desarrollo profesional de software, ignorando conceptos clave como:

- Modularidad/Encapsulación
- Reutilización
- Mantenibilidad
- Escalabilidad

Esto genera varios problemas críticos:

Problema principal	Consecuencias
Falta de estandarización	Automatizaciones ad-hoc; mantenimiento costoso
Falta de documentación	Pérdida de conocimiento; dependencia de personas concretas
Incremento de tiempos	Automatizaciones artesanales generan retrasos en proyectos

Esta situación lleva a un caos difícil de sostener y mantener a largo plazo:

- Miles de pipelines diferentes.
- Dificultad para realizar cambios rápidos o implementar mejoras generales.
- Riesgo operativo elevado por errores humanos.

4. Evolución de las prácticas en paralelo a la automatización: Git, Lenguajes Declarativos, IaC y Contenedores

En paralelo a la evolución natural de DevOps y la automatización, hemos visto cómo han ido emergiendo y consolidándose algunas prácticas esenciales que han aportado robustez, claridad y estandarización en la industria del desarrollo de software.

Estas prácticas son:

- **Gestión del flujo de trabajo con Git**
- **Uso intensivo de lenguajes declarativos**
- **Infraestructura como Código (IaC)**
- **Contenedores**

4.1 Gestión del flujo de trabajo con Git

Un flujo de trabajo claramente definido es fundamental para mantener orden y control en el desarrollo, especialmente cuando manejamos automatizaciones complejas.

Esquema del Workflow de trabajo habitual:

```
hotfix1
    ^
main/master   ← Protegida (No commit directo, solo merges desde otras ramas). Proc
    ^
release       ← Release Candidate (v2.0.1-RC1). Pruebas de sistema (entorno de pru
    ^
develop       ← Protegida (No commit directo, solo merges desde feature tras CI e
    ^
feature1      ← Desarrollo unitario e integración
feature2/...  ← Ramas específicas por desarrollador o tarea
```

Automatizaciones asociadas al workflow de Git:

Pipeline CI (feature, develop)

- **Pruebas unitarias**
- **Pruebas de integración**
- Producto: **Informe de pruebas**

Pipeline CI avanzado (release)

- **Pruebas de sistema**
- **Pruebas de rendimiento y carga**
- **Análisis de calidad del código** (SonarQube)

Pipeline CD (Delivery)

- Toma del artefacto desde la rama **main/master**
- Publicación del artefacto en registros específicos según tipo:
  - **Imágenes de contenedor**: Docker Registry, Quay, Artifactory, GitLab Container Registry
  - **Webapps Java**: Nexus, Artifactory
  - **Librerías JS**: Nexus, Artifactory, NPM Registry
  - **Librerías Java**: Nexus, Artifactory, Maven Repository
  - **Apps móviles (Android, iOS)**: Nexus, Artifactory, Google Play, Apple Store, Microsoft Store
  - **.NET Apps**: Nexus, Artifactory, Microsoft Store

Pipeline CD (Deployment)

- Despliegue automatizado en producción:
  - Por ejemplo, despliegue de un microservicio interno en Kubernetes usando una imagen del registro.
  - Registro automático en servicios de descubrimiento (por ejemplo, APIgee).

4.2 Lenguajes Declarativos: cambio de paradigma

Tradicionalmente, las automatizaciones se han escrito en lenguajes imperativos como Bash (.sh), PowerShell (.ps1) o Python (.py).

Paradigma Imperativo (instrucciones detalladas):

```
$ mkdir ventana  # Orden directa, centrada en la acción
```

En contraste, hoy se prefiere el paradigma declarativo, que se centra en definir el estado final deseado en vez del procedimiento para alcanzarlo:

Paradigma Declarativo (estado final definido):

> "Debajo de la ventana tiene que haber una silla."

Ventajas del Paradigma Declarativo:

- Menos código escrito, más claro y sencillo.
- Mayor legibilidad y facilidad de mantenimiento.
- **Idempotencia intrínseca**: la ejecución repetida siempre lleva al mismo estado final.

Ejemplos de herramientas declarativas:

- **Terraform**: Describe infraestructuras sin verbos explícitos.
- **Kubernetes**: Manifiestos YAML que definen estado deseado (sin verbos).
- **Docker Compose**: Define estado deseado de contenedores en YAML.
- **Spring Framework**: Anotaciones declarativas sin verbos explícitos.
- **Angular**: Componentes y bindings declarativos.

Ejemplo de acción declarativa con Terraform:

```
"aws_instance"">resource "aws_instance" "web" {
  ami           = "ami-abc123"
  instance_type = "t2.micro"
}
```

Comandos asociados (verbos externos):

```
terraform apply
terraform destroy
```

4.3 Idempotencia y su importancia

Idempotencia implica que, independientemente del estado inicial, al ejecutar un script múltiples veces, siempre obtenemos el mismo estado final deseado.

Esto facilita enormemente automatizaciones y reduce errores operacionales.

Ejemplo práctico (Ansible):

- **Imperativo**: “Generar certificados SSL hoy.”
- **Declarativo (idempotente)**: “Garantizar existencia de certificados SSL válidos en todo momento.”

4.4 Aclaración importante sobre Ansible y la idempotencia

Ansible utiliza principalmente un lenguaje **imperativo**, pero muchos módulos ofrecen un enfoque declarativo con la propiedad `state`.

Ejemplo mixto (imperativo-declarativo) con Ansible:

```yaml
- name: Ejemplo mixto
  hosts: all
  tasks:
    - name: Comando imperativo
      shell: echo "Imperativo puro"  # Imperativo, no idempotente

    - name: Crear directorio (declarativo, idempotente)
      file:
        path: /tmp/carpeta
        state: directory
```

En Ansible, la idempotencia real depende del módulo. El desarrollador debe asegurarla explícitamente a nivel de Playbook.

4.5 Infraestructura como Código (IaC)

IaC implica tratar la infraestructura como código:

Características clave de IaC:

- Infraestructura definida en ficheros de código.
- Gestión de versiones de infraestructura.
- Integración en pipelines de CI/CD igual que cualquier otro software.

Ejemplo habitual:

- Scripts de Terraform gestionados en GitLab.
- Automatización mediante GitLab CI.
- Ejecución de múltiples Playbooks Ansible detrás, garantizando estado correcto.

Ejemplo típico:

```
GITLAB CI → Terraform (infra mínima)
          → ANSIBLE (provisionado detallado)

JENKINS / GITLAB CI → Terraform + Ansible AWX → workflow de playbooks
```

4.6 Uso de contenedores frente a métodos tradicionales de despliegue

Históricamente, la instalación de software presentaba problemas:

Método tradicional (problemas):

- Colisión de dependencias entre aplicaciones.
- Fallo de una aplicación compromete todo el sistema.

Máquinas virtuales (problemas):

- Pérdida de recursos.
- Baja eficiencia de recursos y rendimiento.
- Complejidad operativa.

Solución: Contenedores

Contenedor: entorno aislado dentro del kernel Linux del host para ejecutar procesos.

- Eficientes y ligeros.
- Aislamiento de procesos y dependencias.

Arquitectura de contenedores:

```
App1       |   App2 + App3
---------------------------
Contenedor1|  Contenedor2
---------------------------
Docker, Podman, Cri-o, ContainerD
---------------------------
Sistema operativo Linux (kernel)
---------------------------
Hardware físico
```

Contenedor vs. Máquina virtual:

- **Contenedor**: creado a partir de imágenes comprimidas con software preinstalado (`.tar`).
- **Máquina Virtual**: creada desde imágenes ISO o imágenes virtuales (`.ova` , `.vmdk` , `.vdi`).

Imágenes de contenedor:

- Almacenadas en registros:
  - Docker Hub
  - Microsoft Container Registry
  - Quay.io
  - Oracle Container Registry
- Facilitan distribución y despliegue uniforme y reproducible.

5. Introducción a Platform Engineering

Platform Engineering surge como respuesta a los desafíos encontrados tras años aplicando prácticas DevOps tradicionales. Busca resolver la complejidad, falta de estandarización y dificultades de mantenimiento generadas por la proliferación de automatizaciones ad-hoc, proporcionando plataformas internas que estandarizan y facilitan la vida a desarrolladores y equipos operativos.

Definición clara:

> **Platform Engineering** consiste en diseñar, construir y mantener plataformas internas (Internal Developer Platforms, IDP) que simplifican y aceleran los flujos de trabajo del desarrollo de software, infraestructura y operaciones, proporcionando interfaces claras y procesos estandarizados.

6. Problemas concretos que soluciona Platform Engineering

El enfoque tradicional de DevOps conduce a los siguientes problemas críticos:

Problema	Consecuencia práctica
Automatización ad-hoc	Automatizaciones difíciles de mantener y poco reutilizables.
Falta de documentación	Pérdida del conocimiento y dependencias excesivas.
Ineficiencia operativa	Incremento de tiempos, dificultades para escalar y alto coste.

Platform Engineering aborda estos problemas con una estrategia clara basada en plataformas internas estandarizadas.

7. Qué es una Internal Developer Platform (IDP)

Una **Internal Developer Platform (IDP)** es un conjunto de herramientas, servicios, plantillas y procesos automatizados, diseñados específicamente para satisfacer las necesidades internas de los equipos de desarrollo.

Características principales de una IDP:

- **Autoservicio**: Los desarrolladores acceden directamente a recursos sin depender constantemente de equipos externos.
- **Estandarización**: Herramientas, pipelines, y prácticas homogéneas.
- **Automatización integral**: Desde la creación del proyecto hasta su despliegue y monitorización.
- **Observabilidad integrada**: Facilita seguimiento y monitorización continua.
- **Seguridad incorporada**: Autenticación, autorización y cumplimiento normativo automatizados.

8. Componentes fundamentales de una Internal Developer Platform

Una IDP típicamente incluye los siguientes componentes clave:

Componente	Ejemplos prácticos o herramientas
Catálogo de servicios	Repositorio centralizado de servicios, APIs y recursos preaprobados (Backstage, Humanitec, Port).
Templates y scaffolding	Automatización en creación de proyectos (Backstage Scaffolder, Cookiecutter).
Infraestructura como código	Terraform, Pulumi, Crossplane.

Componente	Ejemplos prácticos o herramientas
Despliegue continuo (CD)	Kubernetes, ArgoCD, Flux, GitOps.
Gestión de secretos y seguridad	Hashicorp Vault, AWS Secrets Manager, Azure Key Vault.
Monitoreo y observabilidad	Prometheus, Grafana, Datadog, ELK Stack.

9. Ejemplos destacados de plataformas internas (IDP)

- **Backstage** (Spotify): Popular por su flexibilidad y extensibilidad, facilita la creación de catálogos, templates y documentación automatizada.
- **Humanitec**: Plataforma enfocada en entornos dinámicos de desarrollo y despliegue continuo con fuerte soporte en Kubernetes.
- **Port**: Orientada a la creación ágil y sencilla de portales internos para desarrolladores.
- **Internal.io**: Facilita la creación de interfaces internas robustas y altamente personalizables.

10. Ventajas prácticas de implementar Platform Engineering en la organización

Implementar Platform Engineering aporta ventajas concretas como:

- **Reducción significativa en tiempos de entrega**: Automatización ágil y plantillas reutilizables.
- **Menores costes operativos**: Reducción de complejidad y estandarización disminuye costes.
- **Incremento de la calidad**: Estándares claros y buenas prácticas integradas automáticamente.
- **Autonomía y productividad de los equipos**: Acceso autoservicio a recursos.
- **Mayor visibilidad y documentación**: Procesos claramente definidos y automatizados.

11. Cómo adoptar Platform Engineering en tu empresa (Propuesta de implantación)

Estrategia general para adopción:

La adopción efectiva de Platform Engineering en una organización requiere un enfoque estructurado en varias fases:

Fase	Acciones clave
Evaluación inicial	Auditoría de automatizaciones actuales; identificación de puntos débiles.
Selección de tecnología	Escoger una plataforma interna adecuada (por ejemplo, Backstage).
Estandarización	Definir estándares claros para desarrollo, pipelines, plantillas y procesos.
Implementación de IDP	Construcción incremental de componentes internos: catálogo, plantillas, automatizaciones.
Automatización de extremo a extremo	Automatización completa desde desarrollo hasta operaciones en producción.
Documentación exhaustiva	Documentación sistemática de procesos, recursos y flujos operativos.
Formación interna	Formación continua a equipos sobre uso eficiente de IDP.

12. Ejemplo práctico del uso integrado de herramientas dentro de una IDP

Un flujo típico puede ser:

1. Desarrollador usa **Backstage Scaffolder** para iniciar un proyecto nuevo:
    - Genera automáticamente repositorio de Git con estructura básica.
    - Inicializa pipeline CI/CD en GitLab o Jenkins.

2. Terraform automáticamente aprovisiona infraestructura (VMs, redes, bases de datos).
3. GitLab CI o Jenkins ejecuta playbooks Ansible para configuración detallada y despliegue del software.
4. El despliegue usa imágenes de contenedores desde registros centralizados (Docker Hub, Artifactory).
5. Kubernetes gestiona automáticamente el escalado, actualizaciones, y alta disponibilidad.
6. Toda la información del despliegue y del servicio queda automáticamente registrada en el catálogo interno, con enlaces a monitoreo (Prometheus/Grafana), documentación y logs.

13. Conclusión y perspectivas futuras

Platform Engineering representa la evolución natural y lógica de DevOps tradicional, resolviendo sus limitaciones y llevando la automatización y estandarización a niveles superiores de eficiencia, productividad y calidad.

Al adoptar una plataforma interna, la empresa logra:

- Menor complejidad operativa.
- Mayor escalabilidad y sostenibilidad a largo plazo.
- Un entorno más eficiente para los desarrolladores, impulsando innovación y entregas más rápidas y seguras.

En resumen, Platform Engineering no es simplemente una mejora tecnológica, sino un cambio significativo y estratégico en cómo las empresas modernas gestionan el desarrollo y las operaciones del software.

14. Backstage: Plataforma Interna para Desarrolladores (IDP)

Backstage es una plataforma interna de desarrollo (**Internal Developer Platform - IDP**) creada originalmente por Spotify, diseñada para facilitar, centralizar y estandarizar el trabajo diario de los equipos de desarrollo, operaciones e infraestructura.

Su objetivo principal es ofrecer a los desarrolladores un entorno de trabajo autoservicio que les permita crear, gestionar y mantener proyectos de forma sencilla, consistente y rápida,

eliminando la complejidad técnica que implica interactuar con múltiples herramientas dispersas.

14.1 Arquitectura Técnica de Backstage

Backstage es una aplicación web basada en tecnologías JavaScript modernas, tanto para el frontend como para el backend.

Componente	Tecnología empleada	Descripción
Frontend	React	Interfaz visual intuitiva, modular y personalizable.
Backend	Node.js, Express	Proporciona servicios backend, APIs y conectividad con herramientas externas.
Bases de datos	PostgreSQL, SQLite	Almacenamiento persistente de información (catálogo, usuarios, configuración).
Extensibilidad	Plugins	Arquitectura basada en plugins para integrar cualquier herramienta o sistema externo.

La extensibilidad mediante plugins permite que Backstage se adapte fácilmente a casi cualquier necesidad organizativa.

14.2 ¿Qué ofrece Backstage como IDP?

Backstage centraliza y automatiza todas las actividades necesarias para lanzar, mantener y evolucionar software dentro de una organización:

Principales funcionalidades de Backstage:

- **Catálogo de Software**
- **Generación de Proyectos (Scaffolder)**
- **Automatización CI/CD integrada**
- **Documentación Técnica integrada**
- **Monitoreo y Observabilidad integrada**
- **Gestión de Permisos y Seguridad**
- **Portal unificado de herramientas externas**

14.3 Catálogo de Software en Backstage

El **Catálogo** es el núcleo de Backstage. Ofrece un inventario organizado y actualizado de todos los servicios, APIs, componentes, y recursos técnicos de la empresa.

- Vista centralizada de todos los servicios y componentes técnicos.
- Documentación técnica y operativa integrada automáticamente.
- Visualización de dependencias entre componentes y servicios.

Beneficios del catálogo:

- Mayor transparencia y reutilización.
- Reducción de trabajo duplicado.
- Fácil mantenimiento y auditoría de componentes existentes.

14.4 Scaffolder: Creación automática de Proyectos

El **Scaffolder** permite crear nuevos proyectos de forma automatizada, mediante plantillas predefinidas.

Ejemplo de plantilla para microservicio Spring/Java:

Al crear un nuevo microservicio mediante Scaffolder, Backstage automáticamente:

- Genera un repositorio Git (GitLab, GitHub, Bitbucket).
- Configura la estructura básica del proyecto.
- Establece automáticamente pipelines CI/CD básicos (Jenkins, GitLab CI).
- Provisiona infraestructura (Terraform) necesaria para pruebas y despliegue.
- Configura entornos específicos (namespaces de Kubernetes o recursos en VMware).
- Genera documentación inicial básica (README, guías técnicas).

Ejemplos comunes de plantillas que podrías tener:

- Microservicios Java Spring Boot.
- Aplicaciones frontend en Angular, React, o Vue.
- Librerías JavaScript o Java compartidas.
- Componentes de infraestructura (Terraform, Ansible).

Esto permite estandarizar significativamente el proceso de creación de nuevos proyectos y acelerar la productividad inicial del equipo.

14.5 Integración profunda con herramientas CI/CD e Infraestructura

Backstage se integra con herramientas externas para automatizar procesos complejos como CI/CD y IaC. Algunas integraciones frecuentes:

- **CI/CD**: Jenkins, GitLab CI, GitHub Actions, Azure DevOps.
- **Infraestructura como Código (IaC)**: Terraform, Pulumi, Crossplane.
- **Orquestación de contenedores**: Kubernetes, Helm, ArgoCD.
- **Gestión de configuración y despliegue**: Ansible, Chef, Puppet.
- **Repositorios de artefactos**: Docker Registry, Artifactory, Nexus.

Esto evita que los desarrolladores tengan que interactuar directamente con múltiples herramientas y entornos de forma manual, simplificando enormemente su trabajo.

14.6 Gestión automatizada de Entornos y Recursos

Backstage automatiza la gestión de recursos y entornos para garantizar su uso eficiente y seguro:

- **Namespaces Kubernetes**:
  - Automáticamente creados por cada proyecto.
  - Limitaciones predefinidas (CPU, memoria, cuotas).
- **Máquinas Virtuales (VMware, AWS, Azure)**:
  - Automatización del aprovisionamiento (Terraform).
  - Control automático de recursos máximos permitidos (núcleos CPU, RAM).

Esto asegura la adherencia a estándares y políticas internas desde el primer día.

14.7 Documentación Técnica Integrada (TechDocs)

Backstage cuenta con una solución interna (**TechDocs**) que permite:

- Documentar automáticamente servicios, APIs y componentes.
- Mantener documentación técnica siempre actualizada junto al código fuente.
- Usar Markdown para documentación sencilla y fácil de mantener.

14.8 Observabilidad y Monitoreo en Backstage

Backstage integra visualización de monitoreo y métricas mediante herramientas comunes como:

- Prometheus
- Grafana
- Datadog
- ELK Stack

Esto permite que los desarrolladores accedan a logs, métricas y alertas directamente desde el catálogo del servicio, mejorando la visibilidad operativa.

14.9 Seguridad integrada y Autenticación (Auth)

Backstage incorpora mecanismos sólidos de seguridad:

- Integración con OAuth y sistemas SSO (Single Sign-On) como Google Workspace, Azure AD, Okta.
- Gestión centralizada de permisos basada en roles.
- Acceso seguro a recursos sensibles mediante integración con herramientas como Hashicorp Vault.

14.10 Ejemplo práctico de flujo completo en Backstage:

Escenario típico para nuevo microservicio Java:

1. El desarrollador accede al portal Backstage.
2. Selecciona plantilla "Microservicio Spring Boot".
3. Completa formulario con información básica del proyecto (nombre, propietario, recursos necesarios).
4. Backstage automáticamente:
    - Genera repositorio Git con esqueleto del proyecto.
    - Inicializa pipelines CI/CD (pruebas, calidad, despliegue).
    - Provisiona entorno Kubernetes (namespace) o VM específica.
    - Crea documentación técnica básica.
    - Actualiza automáticamente el catálogo interno.

5. Desarrollador accede inmediatamente al entorno configurado y comienza a desarrollar y probar.

14.11 Ventajas prácticas del uso de Backstage:

- **Reducción drástica del tiempo inicial para nuevos proyectos**.
- **Consistencia técnica y operativa** entre proyectos y equipos.
- **Reducción significativa de la complejidad técnica** que enfrentan los desarrolladores diariamente.
- **Menos errores operativos** debido a procesos automatizados y estandarizados.
- **Mayor visibilidad y transparencia técnica** a nivel organizacional.

14.12 Comunidad y Ecosistema de Backstage

Backstage es una herramienta open-source respaldada por una comunidad vibrante y en constante crecimiento. Algunas organizaciones notables que usan activamente Backstage incluyen Spotify, Netflix, American Airlines, Zalando, y Expedia.

Esto garantiza soporte continuo, abundantes recursos y una evolución constante basada en experiencias reales.