Backstage

Introducción

¿Qué es Backstage?

Backstage es una plataforma de desarrollo de portales internos de desarrolladores (IDP, Internal Developer Platform) creada inicialmente por Spotify y donada a la CNCF (Cloud Native Computing Foundation).

Su objetivo es unificar herramientas, documentación, servicios y catálogos en una única interfaz web para facilitar el trabajo de los desarrolladores. Proporciona una experiencia de autoservicio completa que permite iniciar, mantener y operar software sin necesidad de conocer todos los sistemas subyacentes.

Backstage se basa en:

- Un frontend extensible en React
- Un backend en Node.js
- Un modelo de plugins altamente personalizable
- Soporte integrado para catalogar componentes, documentación (TechDocs), plantillas de scaffolding, autenticación, y más.

Objetivo del proyecto

Esta instancia de Backstage tiene como objetivo:

- Proporcionar una plataforma unificada para la gestión y despliegue de microservicios.
- Permitir a los equipos de desarrollo trabajar en modo autoservicio, sin depender directamente de operaciones o plataforma.
- Estandarizar el proceso de creación, documentación y despliegue de nuevos proyectos.
- Integrar herramientas existentes como GitHub, Jenkins, Terraform, Kubernetes, SonarQube, etc., desde una única interfaz.
- Facilitar la gestión del ciclo de vida completo del software (desde el código fuente hasta el entorno de producción).

Público objetivo

- Equipos de desarrollo que deseen un entorno integrado para iniciar y mantener sus servicios.
- Ingenieros de plataforma (Platform Engineers) que busquen estandarizar los procesos de creación, despliegue y operación de software.
- DevOps e ingenieros de automatización que necesiten coordinar herramientas a través de un frontend común.
- Líderes técnicos que deseen mejorar la trazabilidad, calidad, reutilización y gobernanza de sus sistemas.

Instalación y configuración inicial

Requisitos previos

Antes de comenzar, asegúrate de tener instalados los siguientes componentes en tu entorno de desarrollo:

- Node.js (versión 18 LTS o superior)
- Yarn (preferiblemente versión 1.22+)
- Git
- Docker (si se planea ejecutar en contenedor)
- npx (incluido con Node.js)

Opcional pero recomendado:

- PostgreSQL (si se quiere conectar a base de datos real)
- Un token de GitHub (para integraciones)

Crear un nuevo proyecto Backstage

Se puede crear un nuevo proyecto desde cero usando el comando oficial:

```
npx @backstage/create-app@latest
```

Durante el proceso, se solicitará:

- Nombre del proyecto (por ejemplo, mi-backstage)

El generador crea un proyecto base ya funcional, con todas las dependencias configuradas y una estructura modular basada en paquetes.

Estructura inicial del proyecto

```
mi-backstage/
├── app-config.yaml                 # Configuración principal de la app
├── app-config.local.yaml           # Configuración específica para entorno local
├── packages/                       # Contiene app (frontend) y backend (API)
├── plugins/                        # Plugins personalizados o adicionales
├── examples/                       # Ejemplos de entidades y plantillas
├── catalog-info.yaml               # Información del propio Backstage como compor
├── yarn.lock, package.json         # Dependencias y scripts
└── README.md                       # Instrucciones del proyecto
```

- El frontend se encuentra en packages/app
- El backend se encuentra en packages/backend

Configuración local mínima (app-config.local.yaml)

Este archivo sobrescribe la configuración general para entornos de desarrollo.

```yaml
app:
  title: Backstage
  baseUrl: http://localhost:3000

backend:
  baseUrl: http://localhost:7007
  listen:
    port: 7007

integrations:
  github:
    - host: github.com
      token: ${GITHUB_TOKEN}

proxy:
  '/jenkins':
    target: https://jenkins.miempresa.com
    headers:
      Authorization: Basic ${JENKINS_BASIC_AUTH}
```

Se recomienda usar un archivo .env para gestionar las variables sensibles como tokens de GitHub o Jenkins.

Inicializar y ejecutar

Desde la raíz del proyecto generado:

```
yarn install --immutable
yarn dev
```

Esto arrancará el frontend y backend en paralelo, dejando disponible la interfaz en:

http://localhost:3000

Arranque en desarrollo

Este bloque detalla cómo ejecutar el entorno de desarrollo de Backstage de forma local, tanto en modo monolito como en modo contenedor, y cómo trabajar con sus distintos

componentes.

Ejecución local (modo desarrollo)

El entorno de desarrollo ejecuta por defecto tanto el frontend (packages/app) como el backend (packages/backend) con recarga automática.

Paso 1: Instalar dependencias

Desde la raíz del proyecto:

```
yarn install --immutable
```

Paso 2: Compilar el backend y el frontend

```
yarn tsc
```

Paso 3: Ejecutar en modo desarrollo

```
yarn dev
```

Esto ejecutará:

- El frontend en http://localhost:3000
- El backend en http://localhost:7007

Ambos procesos estarán enlazados y se recargarán automáticamente al modificar el código fuente.

Ejecución local con Docker

Se puede construir y ejecutar Backstage como una imagen de contenedor Docker.

Paso 1: Compilar el proyecto y generar la imagen

```
yarn build:backend
yarn workspace backend build-image
```

Esto compila y genera una imagen Docker llamada backstage, usando packages/backend/Dockerfile.

Paso 2: Ejecutar la imagen

```
docker run -p 7007:7007 backstage
```

Montar configuración personalizada (opcional)

```
docker run -p 7007:7007 \
  -v $(pwd)/app-config.yaml:/app/app-config.yaml \
  -v $(pwd)/app-config.local.yaml:/app/app-config.local.yaml \
  backstage
```

Uso de Docker Compose (con PostgreSQL)

Para simular un entorno real de ejecución con base de datos:

Archivo docker-compose.yaml

```
version: '3'
services:
  backstage:
    image: backstage
    ports:
      - "7007:7007"
    environment:
      POSTGRES_HOST: db
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: example
    depends_on:
      - db

  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: example
```

Ejecutar servicios

```
docker-compose up
```

Esto levantará Backstage en http://localhost:7007, con PostgreSQL como base de datos.

Scripts útiles

Acción	Comando
Ejecutar entorno de desarrollo	yarn dev
Compilar proyecto completo	yarn tsc && yarn build:backend
Crear imagen Docker	yarn workspace backend build-image
Ejecutar contenedor manualmente	docker run -p 7007:7007 backstage
Ejecutar con configuración personalizada	docker run -v ... backstage
Ejecutar todo con Docker Compose	docker-compose up

Despliegue con Docker

Backstage incluye herramientas integradas para empaquetar la aplicación como una imagen de contenedor. Esto permite un despliegue uniforme, reproducible y fácilmente integrable con sistemas como Kubernetes, ECS, Nomad, etc.

Construcción de la imagen de contenedor

Desde la raíz del proyecto Backstage:

Paso 1: Instalar dependencias y compilar

```
yarn install --immutable
yarn tsc
yarn build:backend
```

Paso 2: Construcción de la imagen

Usa el script oficial:

```
yarn workspace backend build-image
```

Esto:

- Construye el frontend y el backend
- Los empaqueta juntos usando backstage-cli backend:build-image
- Usa el Dockerfile ubicado en packages/backend/Dockerfile
- Crea una imagen Docker etiquetada por defecto como backstage

Contenido del Dockerfile por defecto

Ubicado en packages/backend/Dockerfile:

```
FROM node:18-bullseye-slim

WORKDIR /app

COPY . .

RUN yarn install --frozen-lockfile && yarn tsc && yarn build

CMD ["node", "packages/backend", "--config", "app-config.yaml"]
```

Este archivo puede personalizarse según las necesidades del entorno (por ejemplo, añadiendo certificados, herramientas, secrets, etc.).

Ejecución de la imagen

Modo básico

```
docker run -p 7007:7007 backstage
```

Backstage estará disponible en http://localhost:7007.

Con configuración personalizada

Si necesitas montar tus propios archivos de configuración:

```
docker run -p 7007:7007 \
  -v $(pwd)/app-config.yaml:/app/app-config.yaml \
  -v $(pwd)/app-config.local.yaml:/app/app-config.local.yaml \
  backstage
```

Puedes montar más archivos (catalog-info.yaml, techdocs.yaml, etc.) si es necesario.

Uso de Docker Compose con PostgreSQL

Una opción práctica para entornos locales o entornos mínimos de staging.

Archivo docker-compose.yaml

```
version: '3'
services:
  backstage:
    image: backstage
    ports:
      - "7007:7007"
    environment:
      POSTGRES_HOST: db
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: example
    depends_on:
      - db

  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: example
```

Ejecutar todo

```
docker-compose up
```

Accede a Backstage en http://localhost:7007.

Buenas prácticas para producción

- Montar un volumen para almacenar la caché de TechDocs (si se usa local).
- Montar secrets y env externos mediante volúmenes o .env gestionado.
- Gestionar logs correctamente (puede añadirse un logLevel en app-config.yaml).
- Usar certificados TLS si se expone directamente.
- Idealmente, ejecutar la imagen en un orquestador como Kubernetes (bloques posteriores).

Catálogo de Software en Backstage

El catálogo es el corazón de Backstage. Permite organizar, visualizar, buscar y gestionar todos los elementos de la plataforma: microservicios, APIs, librerías, usuarios, equipos, pipelines, documentación, etc.

Entidades principales

Cada recurso del catálogo se define como una entidad YAML con estructura estandarizada. Los tipos más comunes son:

Tipo (kind)	Descripción
Component	Servicio, librería, job, tarea o aplicación. Unidad desplegable.
API	Una interfaz ofrecida por un componente (REST, gRPC, GraphQL, etc.)
System	Agrupación lógica de componentes que colaboran entre sí.
Resource	Elemento físico o lógico compartido (BBDD, cola, cluster, etc.)
User	Persona identificable (desarrollador, ops, etc.)
Group	Equipo, squad, tribu o departamento (organización interna)
Location	Agrupa otras entidades en un repositorio o URL externa
Template	Plantilla del Scaffolder para crear nuevos proyectos

Ejemplo de archivo catalog-info.yaml

Este archivo suele colocarse en el raíz de cada repositorio de código y declara la existencia del componente en el catálogo.

```yaml
apiVersion: backstage.io/v1alpha1
kind: Component
metadata:
  name: servicio-backend
  description: Servicio REST de ejemplo
  tags:
    - backend
    - java
  annotations:
    github.com/project-slug: mi-org/servicio-backend
    backstage.io/techdocs-ref: dir:.
spec:
  type: service
  lifecycle: production
  owner: equipo-backend
  system: sistema-principal
```

Atributos principales de una entidad

- apiVersion: Versión del esquema. Siempre empieza por backstage.io/
- kind: Tipo de entidad (Component, User, etc.)
- metadata:
  - name: Nombre único del recurso
  - description: Descripción
  - annotations: Clave para la integración (TechDocs, GitHub, Jenkins, etc.)
- spec:
  - type: Tipo funcional (service, library, website, job, etc.)
  - lifecycle: experimental, production, deprecated
  - owner: Grupo o usuario responsable (Group o User)
  - dependsOn, subcomponentOf, etc.

Registro de entidades

Las entidades pueden ser registradas en el catálogo:

Registro manual

Desde la interfaz:

- Ir a “Catalog” > “Register Existing Component”
- Introducir la URL al archivo YAML (puede ser local o remoto)
  - Ejemplo: https://github.com/mi-org/mi-repo/blob/main/catalog-info.yaml

Registro automático

Configurando una ubicación (Location) en app-config.yaml:

```
catalog:
  locations:
    - type: url
      target: https://github.com/mi-org/catalog/raw/main/all.yaml
      rules:
        - allow: [Component, API, Group]
```

O usando exploración de repositorios completos:

```
catalog:
  providers:
    github:
      organization:
        - url: https://github.com/mi-org
```

Anotaciones útiles

Anotación	Función
backstage.io/techdocs-ref	Vincula a la documentación técnica (dir:.)
github.com/project-slug	Vincula con el repositorio GitHub (org/repo)
jenkins.io/job-full-name	Vincula con un Job de Jenkins

Anotación	Función
backstage.io/kubernetes-label-selector	Mapea el componente con un deployment en K8s
backstage.io/managed-by-location	Muestra que la entidad ha sido creada por una plantilla
backstage.io/managed-by-origin-location	Igual que anterior, más útil para auditoría

Visualización y navegación

Una vez registrada, cada entidad tiene una página propia en el catálogo:

- Información general: owner, sistema, dependencias
- Documentación integrada (TechDocs)
- Links al repositorio, Jenkins, API, monitoreo, etc.
- Widgets personalizados según los plugins habilitados

Organización avanzada

- Agrupar servicios en System
- Asociar servicios a API
- Relacionar Resource como base de datos o colas
- Vincular User con Group
- Usar anotaciones para enlazar recursos externos

TechDocs — Documentación Técnica en Backstage

TechDocs es el sistema integrado de Backstage para generar, almacenar y mostrar documentación técnica directamente desde el catálogo de componentes.

¿Qué es TechDocs?

Es un generador de documentación basado en MkDocs (estático y basado en Markdown), pensado para que cada equipo mantenga la documentación técnica junto a su código, en su mismo repositorio.

Ventajas:

- Documentación siempre actualizada y versionada junto al código.
- Se genera automáticamente desde Markdown.
- Se visualiza directamente en la interfaz de Backstage.
- Soporta extensiones como plantillas, navegación lateral, búsquedas, etc.

Estructura esperada en el repositorio

```
mi-servicio/
├── catalog-info.yaml
├── mkdocs.yml              <- Configuración de la documentación
├── .docs/                  <- Contenido técnico
│   └── index.md            <- Página principal
│   └── arquitectura.md
│   └── uso.md
```

También puede usarse docs/ en lugar de .docs/.

Ejemplo de mkdocs.yml

```yaml
site_name: Documentación de Servicio
nav:
  - Introducción: index.md
  - Arquitectura: arquitectura.md
  - Uso: uso.md

plugins:
  - techdocs-core
```

Enlace entre el componente y su documentación

Debe incluirse esta anotación en el catalog-info.yaml del componente:

```yaml
metadata:
  annotations:
    backstage.io/techdocs-ref: dir:.
```

Esto le dice a Backstage que debe buscar mkdocs.yml y el contenido en la raíz del repo (o en la subcarpeta especificada).

Modos de funcionamiento

TechDocs puede funcionar en dos modos:

Modo	Descripción
Generación local	Genera HTML al vuelo cada vez que alguien accede (más simple).
Generación precompilada (recommended)	Se genera la documentación como HTML y se publica en un bucket (S3, GCS, Azure Blob).

Configuración en app-config.yaml:

Modo precompilado (por defecto en producción):

```yaml
techdocs:
  publisher:
    type: 'local' # También: 'awsS3', 'googleGcs', 'azureBlobStorage'
  generator:
    runIn: 'local'  # 'docker' si quieres aislar la generación
```

Compilación y prueba local

Instala la CLI:

```
npm install -g @techdocs/cli
```

Generar HTML:

```
techdocs build
```

Visualizar en local:

```
techdocs serve
```

Compilación en CI

En una pipeline (ej: GitHub Actions):

```
- name: Build TechDocs
  run: |
    npx @techdocs/cli build
    npx @techdocs/cli publish --publisher-type local
```

Cambia local por awsS3, etc., según la configuración del backend de TechDocs.

Requisitos del backend

El backend de Backstage debe tener habilitado el plugin de TechDocs. Ya viene instalado por defecto:

En packages/backend/src/plugins/techdocs.ts:

```
import { createRouter } from '@backstage/plugin-techdocs-backend';

export default async function createPlugin(env: PluginEnvironment) {
  return await createRouter({
    logger: env.logger,
    config: env.config,
    discovery: env.discovery,
    cache: env.cache,
    database: env.database,
  });
}
```

Visualización en Backstage

- Cada componente que tenga backstage.io/techdocs-ref mostrará la pestaña **Docs** automáticamente.
- Incluye navegación lateral, buscador y diseño responsivo.
- El renderizado es seguro: no se permite JS en los .md.

Scaffolder — Generación de proyectos desde plantillas

El Scaffolder de Backstage permite a los desarrolladores crear nuevos proyectos en modalidad autoservicio a partir de plantillas parametrizables. Estas plantillas pueden incluir cualquier lógica de creación: generación de código, push a Git, CI/CD, registros en el catálogo, ejecución de procesos, etc.

¿Qué es una plantilla (Template)?

Una plantilla es un conjunto de archivos y metadatos que definen:

- Qué parámetros debe introducir el usuario (nombre, lenguaje, tipo de servicio, etc.)
- Qué pasos se ejecutarán automáticamente (generar archivos, crear repositorio, lanzar Jenkins, registrar en catálogo, etc.)
- Qué salidas o enlaces mostrar al finalizar el proceso

Estructura de una plantilla

```
templates/
└── mi-plantilla/
    ├── template.yaml          <- Metadatos e instrucciones (es el "orquestador")
    └── content/               <- Archivos y carpetas que se copiarán al nuevo repo
        ├── README.md
        ├── main.py
        └── mkdocs.yml
```

Ejemplo de template.yaml

```yaml
apiVersion: scaffolder.backstage.io/v1beta3
kind: Template
metadata:
  name: python-service
  title: Servicio Python
  description: Plantilla para crear un microservicio en Python
spec:
  owner: platform-team
  type: service

  parameters:
    - title: Información del proyecto
      required: [name, owner]
      properties:
        name:
          type: string
          description: Nombre del componente
        owner:
          type: string
          description: Equipo responsable

  steps:
    - id: fetch
      name: Obtener contenido base
      action: fetch:template
      input:
        url: ./content
        values:
          name: ${{ parameters.name }}
          owner: ${{ parameters.owner }}

    - id: publish
      name: Crear repositorio en GitHub
      action: publish:github
      input:
        repoUrl: github.com?owner=mi-org&repo=${{ parameters.name }}
        defaultBranch: main
        description: Repositorio generado automáticamente
        protectDefaultBranch: true

    - id: register
      name: Registrar en el catálogo
```

```
      action: catalog:register
      input:
        repoContentsUrl: ${{ steps['publish'].output.repoContentsUrl }}
        catalogInfoPath: /catalog-info.yaml

  output:
    links:
      - title: Repositorio en GitHub
        url: ${{ steps['publish'].output.remoteUrl }}
      - title: Ver en el catálogo
        url: ${{ steps['register'].output.entityRef | entityRefToUrl }}
```

Archivos típicos dentro de content/

```
content/
├── README.md                ← Puede usar los valores de la plantilla
├── catalog-info.yaml        ← Archivo que registra la entidad generada
├── mkdocs.yml               ← Configuración de TechDocs
├── .docs/index.md           ← Documentación técnica inicial
└── main.py                  ← Código base del microservicio
```

Puedes usar variables en los archivos con `${{ values.name }}`, `${{ values.owner }}`, etc.

Acciones disponibles en Scaffolder

Acciones más comunes:

Acción	Descripción
`fetch:template`	Copiar archivos desde un path local, remoto o repositorio.
`publish:github`	Crear y subir código a un nuevo repo de GitHub.
`catalog:register`	Registrar una entidad en el catálogo.
`http:backstage:request`	Hacer llamadas HTTP desde el backend de Backstage (útil para Jenkins, etc).
`file:template`	Escribir o sobrescribir archivos en el destino.

Acción	Descripción
`execute`	Ejecutar comandos en un entorno controlado (requiere configuración especial).

Parámetros de entrada

Puedes declarar parámetros personalizados para pedir al usuario:

```yaml
parameters:
  - title: Información del microservicio
    required: [name, owner, language]
    properties:
      name:
        type: string
        title: Nombre del servicio
      owner:
        type: string
        title: Responsable del componente
      language:
        type: string
        enum: [python, nodejs, java]
        title: Lenguaje
```

Salidas (output)

La sección output permite mostrar enlaces o mensajes una vez finalizada la plantilla:

```yaml
output:
  links:
    - title: Ver repositorio en GitHub
      url: ${{ steps['publish'].output.remoteUrl }}
    - title: Ver componente en catálogo
      url: ${{ steps['register'].output.entityRef | entityRefToUrl }}
```

Registro de plantillas en el catálogo

Para que la plantilla esté disponible en la interfaz de Scaffolder, debe registrarse con un catalog-info.yaml:

```yaml
apiVersion: backstage.io/v1alpha1
kind: Template
metadata:
  name: python-service-template
  title: Servicio Python
  description: Crea un microservicio Python desde cero
  tags:
    - python
    - backend
spec:
  path: ./templates/python-service/template.yaml
  type: service
  owner: platform-team
```

Este archivo debe estar registrado o referenciado desde el catálogo principal (app-config.yaml → catalog.locations).

Integración con GitHub

La integración con GitHub permite a Backstage interactuar con repositorios de código, ya sea para crear repositorios automáticamente desde el Scaffolder, para leer archivos del catálogo (catalog-info.yaml), o para acceder a recursos versionados como documentación (TechDocs). Este bloque cubre todos los aspectos clave para conectar correctamente Backstage con GitHub.

Creación de repositorios desde plantillas

Una de las funcionalidades más potentes del Scaffolder de Backstage es la capacidad de crear automáticamente un nuevo repositorio en GitHub a partir de una plantilla (template.yaml).

Ejemplo de paso en template.yaml:

```yaml
- id: publish
  name: Publicar repositorio en GitHub
  action: publish:github
  input:
    repoUrl: github.com?owner=mi-org&repo=${{ parameters.repoName }}
    defaultBranch: main
    description: Repositorio generado automáticamente desde Backstage
```

Configuración de integración (integrations.github)

La integración se realiza en el archivo app-config.yaml, en el bloque integrations. Aquí se especifican uno o varios host de GitHub (por ejemplo, github.com o un GitHub Enterprise on-premises) y el token de acceso asociado:

```yaml
integrations:
  github:
    - host: github.com
      token: ${GITHUB_TOKEN}
```

Se recomienda no escribir el token directamente, sino almacenarlo en una variable de entorno (.env) y referenciarla como `${GITHUB_TOKEN}` .

Gestión de tokens de acceso

Tipos de token posibles:

- **Token de acceso personal (PAT)**: útil para pruebas o entornos internos. Se crea desde GitHub settings.
- **GitHub App**: opción recomendada para producción. Más segura y gestionable, con scopes limitados por repositorio y eventos.

Scopes mínimos recomendados:

- `repo` (crear y administrar repositorios)
- `workflow` (crear acciones de CI/CD)
- `admin:repo_hook` (gestionar webhooks)

- `read:org` (leer miembros de la organización)

Anotaciones para vinculación automática

Para que Backstage reconozca entidades asociadas a GitHub (repos, usuarios, documentación, etc.), se utilizan **anotaciones** específicas en los manifiestos catalog-info.yaml.

Ejemplos comunes:

Anotación	Descripción
`backstage.io/source-location`	Ubicación del `catalog-info.yaml` en GitHub
`backstage.io/managed-by-location`	Fuente de origen de la entidad (útil para scaffolding)
`backstage.io/techdocs-ref`	Ubicación de la documentación técnica (`mkdocs`)
`github.com/project-slug`	Slug de GitHub (`owner/repo`) asociado al componente

Ejemplo completo:

```yaml
apiVersion: backstage.io/v1alpha1
kind: Component
metadata:
  name: backend-api
  annotations:
    github.com/project-slug: mi-org/backend-api
    backstage.io/techdocs-ref: dir:.
spec:
  type: service
  lifecycle: production
  owner: equipo-backend
```

Autenticación con GitHub

Backstage permite integrar sistemas de autenticación de terceros para controlar el acceso al portal, mostrar información del usuario autenticado y vincularlo con entidades del catálogo. Una de las integraciones más habituales es con GitHub como proveedor de identidad mediante OAuth 2.0.

Creación de una OAuth App en GitHub

1. Ir a GitHub Developer Settings.
2. En la sección **OAuth Apps**, hacer clic en **New OAuth App**.
3. Rellenar los datos del formulario:

Campo	Valor sugerido
Application name	Backstage
Homepage URL	`http://localhost:3000` (o tu URL en producción)
Authorization callback URL	`http://localhost:7007/api/auth/github/handler/frame`

4. Al registrar la aplicación, se obtienen:

- `Client ID`
- `Client Secret`

Guarda ambos valores para su uso en la configuración de Backstage.

Configuración del proveedor de autenticación

En el archivo `app-config.yaml` (o `app-config.local.yaml` para entorno local), se debe declarar el proveedor `github` en el bloque `auth.providers`:

```
auth:
  environment: development
  providers:
    github:
      development:
        clientId: ${GITHUB_CLIENT_ID}
        clientSecret: ${GITHUB_CLIENT_SECRET}
```

Usa un archivo `.env` para almacenar los secretos sensibles:

```
GITHUB_CLIENT_ID=xxxxxxxxxxxx
GITHUB_CLIENT_SECRET=yyyyyyyyyyyy
```

Resolución del usuario (signInResolver)

Para vincular un usuario autenticado con una entidad tipo `User` del catálogo de Backstage, se usa el parámetro `signIn.resolvers` . Hay varias estrategias:

a) `usernameMatchingUserEntityName`

Relaciona el nombre de usuario de GitHub con el `metadata.name` de una entidad `User` en el catálogo:

```
signIn:
  resolvers:
    - resolver: usernameMatchingUserEntityName
```

```
# catalog/users.yaml
apiVersion: backstage.io/v1alpha1
kind: User
metadata:
  name: juanlopez
spec:
  profile:
    displayName: Juan López
```

b) `emailMatchingUserEntityAnnotation`

Relaciona el email de GitHub con la anotación `backstage.io/email` del usuario:

```
signIn:
  resolvers:
    - resolver: emailMatchingUserEntityAnnotation


# catalog/users.yaml
apiVersion: backstage.io/v1alpha1
kind: User
metadata:
  name: juan
  annotations:
    backstage.io/email: juan@example.com
spec:
  profile:
    displayName: Juan López
```

Restricción por organización (opcional)

Puedes limitar el acceso únicamente a los usuarios que pertenezcan a una organización específica en GitHub:

```
auth:
  providers:
    github:
      development:
        clientId: ${GITHUB_CLIENT_ID}
        clientSecret: ${GITHUB_CLIENT_SECRET}
        organization: mi-organizacion
```

Verificación y prueba

1. Asegúrate de tener las variables de entorno cargadas y la app configurada.
2. Ejecuta Backstage (`yarn dev` o con Docker).
3. Abre `http://localhost:3000` y haz clic en **Sign In with GitHub**.
4. Al autenticarte, Backstage debería mostrar el perfil del usuario logueado y vincularlo al catálogo si está configurado el resolver.

Integración con Jenkins

Backstage permite la ejecución de procesos externos como **pipelines de Jenkins** directamente desde plantillas del Scaffolder. Esta integración se realiza mediante el backend de Backstage actuando como **proxy HTTP seguro**, autenticado y capaz de interactuar con la API REST de Jenkins.

Configuración del proxy a Jenkins

Para que Backstage pueda comunicarse con Jenkins, se configura un **proxy en `app-config.yaml`** que redirige las peticiones desde el backend al servidor Jenkins:

```
proxy:
  '/jenkins':
    target: https://jenkins.miempresa.com
    headers:
      Authorization: Basic ${JENKINS_BASIC_AUTH}
    changeOrigin: true
    secure: true
```

`JENKINS_BASIC_AUTH` debe ser una cadena base64 con el formato `usuario:token`.

```
echo -n 'admin:API_TOKEN' | base64
```

Ejemplo de `.env`:

```
JENKINS_BASIC_AUTH=YWRtaW46YXBpdG9rZW5fZXhlbXBsZQ==
```

Y en `app-config.yaml`:

```
scaffolder:
  secrets:
    jenkinsBasicAuth: ${JENKINS_BASIC_AUTH}
```

Autenticación mediante API Token

Para autenticar las peticiones desde Backstage, se recomienda usar un **API Token** creado desde la cuenta de Jenkins:

1. Ir a **Manage Jenkins > Manage Users > tu usuario**.
2. Crear un **API Token** y copiarlo.
3. Codificar `usuario:token` en base64.
4. Usarlo como se explicó en el apartado anterior.

Este método es compatible tanto con Jenkins standalone como con instancias integradas con LDAP o SSO.

Obtención del CSRF Crumb (anti-CSRF token)

Algunas instalaciones de Jenkins requieren un token CSRF (`Jenkins-Crumb`) para peticiones POST. Se puede obtener mediante una llamada GET previa:

```
- id: get-jenkins-crumb
  name: Obtener Jenkins Crumb
  action: http:backstage:request
  input:
    method: GET
    path: /proxy/jenkins/crumbIssuer/api/json
    headers:
      Authorization: Basic ${{ secrets.jenkinsBasicAuth }}
```

El resultado se guarda en `${{ steps['get-jenkins-crumb'].output.body.crumb }}` para usarlo más adelante.

Disparo de jobs con parámetros desde plantillas

Una vez obtenido el `crumb` , se puede lanzar un job remoto con parámetros:

```
- id: trigger-jenkins
  name: Lanzar Job en Jenkins
  action: http:backstage:request
  input:
    method: POST
    path: /proxy/jenkins/job/mi-job/buildWithParameters
    query:
      COMPONENT_NAME: ${{ parameters.name }}
    headers:
      Authorization: Basic ${{ secrets.jenkinsBasicAuth }}
      Jenkins-Crumb: ${{ steps['get-jenkins-crumb'].output.body.crumb }}
```

Verificación y buenas prácticas

Para que esto funcione correctamente:

- El **Job en Jenkins** debe aceptar parámetros (`buildWithParameters`).
- El **usuario del token API** debe tener permisos para ejecutar el job.
- Se debe habilitar la opción **"Trigger builds remotely"** en la configuración del Job (si se usa `token`).
- Se recomienda incluir mensajes en el `output.text` de la plantilla para facilitar la trazabilidad:

```
output:
  text:
    - "El Job Jenkins 'mi-job' ha sido lanzado con COMPONENT_NAME=${{ parameters.r
```

Plugins personalizados

Backstage es una plataforma extensible diseñada para ser adaptada a las necesidades de cada organización. Una de sus principales fortalezas es la posibilidad de crear **plugins personalizados**, que permiten incorporar funcionalidades internas o integraciones con herramientas específicas.

Estructura del directorio de plugins

Por convención, los plugins personalizados se ubican la carpeta raíz `plugins/` del proyecto:

```
mi-backstage/
├── plugins/
│   ├── mi-plugin/
│   │   ├── package.json
│   │   ├── src/
│   │   └── README.md
```

Cada subcarpeta representa un plugin independiente con su propio `package.json` .

Creación de un nuevo plugin

Backstage incluye una utilidad CLI para crear un esqueleto de plugin:

```
yarn backstage-cli create-plugin
```

El asistente pedirá el nombre del plugin. Por ejemplo: `mi-plugin`

Esto genera:

- Código base para frontend (React)
- Estructura modular compatible con `packages/app`
- Ejemplo de componente y ruta para el plugin

Registro del plugin en la app

Después de crear el plugin, es necesario **registrarlo** dentro de `packages/app/src/App.tsx` para integrarlo en la navegación y enrutado del frontend:

```
import { MiPluginPage } from '@internal/mi-plugin';

<Route path="/mi-plugin" element={<MiPluginPage />} />
```

Y en la navegación (`Root.tsx`):

```
<SidebarItem icon={ExtensionIcon} to="/mi-plugin" text="Mi Plugin" />
```

Casos de uso típicos

Algunos ejemplos habituales de plugins internos:

Caso de uso	Descripción
Monitorización de costes	Panel que se conecta a AWS Billing, Azure Cost, etc.
Métricas de calidad de software	Integra herramientas como SonarQube, Codacy o Checkmarx
Seguimiento de cumplimiento	Muestra niveles de cumplimiento técnico o de seguridad
Integración con sistemas internos	Consultas a bases de datos, dashboards, reporting

Ejemplo de plugin mínimo

Archivo `src/plugin.ts` de un plugin:

```
import { createPlugin, createRouteRef } from '@backstage/core-plugin-api';

export const miPluginRouteRef = createRouteRef({
  id: 'mi-plugin',
});

export const miPlugin = createPlugin({
  id: 'mi-plugin',
  routes: {
    root: miPluginRouteRef,
  },
});
```

Archivo `src/components/MiPluginPage.tsx` :

```
import React from 'react';

export const MiPluginPage = () => {
  return <div>Hola desde mi plugin personalizado</div>;
};
```

Publicación y reutilización

Los plugins personalizados pueden:

- Compartirse internamente en un **monorepo**
- Publicarse en un **registro privado de NPM**
- Distribuirse como paquetes reutilizables entre proyectos

Para eso, se recomienda estandarizar su creación, pruebas y documentación.

Variables y secretos

El sistema de plantillas de Backstage (Scaffolder) permite el uso de expresiones dinámicas que acceden a valores definidos por el usuario, secretos seguros o salidas de pasos previos. Esta flexibilidad permite construir flujos complejos de generación de proyectos, despliegues, o integración con herramientas externas (Jenkins, Terraform, etc.).

Tipos de variables

Las variables se utilizan dentro de los archivos `template.yaml` (de una plantilla) para referenciar valores dinámicos:

Variable	Descripción
`${{ parameters.* }}`	Parámetros introducidos por el usuario al lanzar la plantilla
`${{ secrets.* }}`	Secretos definidos en el backend, como tokens, claves, contraseñas

Variable	Descripción
`${{ steps.*.output.* }}`	Resultados (outputs) de pasos anteriores dentro del mismo flujo
`${{ config.* }}`	Valores de configuración extraídos de los archivos `app-config.yaml`

Ejemplo de uso de `${{ parameters.* }}`

```yaml
parameters:
  - title: Información básica
    required: [nombre]
    properties:
      nombre:
        type: string
        title: Nombre del componente

steps:
  - id: print
    name: Imprimir nombre
    action: debug:log
    input:
      message: "Se ha introducido el nombre: ${{ parameters.nombre }}"
```

Uso de `${{ secrets.* }}`

Los secretos se definen en el backend y no deben estar en texto plano. Se inyectan en tiempo de ejecución desde `app-config.yaml` :

```yaml
scaffolder:
  secrets:
    githubToken: ${GITHUB_TOKEN}
    jenkinsBasicAuth: ${JENKINS_BASIC_AUTH}
```

Y luego en la plantilla:

```
headers:
  Authorization: Basic ${{ secrets.jenkinsBasicAuth }}
```

Buenas prácticas: nunca codificar secretos directamente en `template.yaml`.

Uso de `${{ steps.*.output.* }}`

Permite encadenar pasos y reutilizar datos producidos por uno en pasos posteriores.

```
steps:
  - id: generar
    name: Generar UUID
    action: uuid
  - id: log
    name: Mostrar UUID
    action: debug:log
    input:
      message: "UUID generado: ${{ steps.generar.output.uuid }}"
```

Uso de `${{ config.* }}`

Permite acceder a la configuración de Backstage (como URLs, nombres, o variables del entorno):

```
input:
  url: ${{ config.integrations.github[0].host }}
```

Validación y depuración

Para verificar que las variables se interpolan correctamente, se puede:

- Usar `debug:log` en pasos intermedios.
- Verificar el resultado del scaffolder en Backstage (UI).
- Consultar los logs del backend si hay errores de ejecución.

Control de errores

- Si una variable no existe o está mal escrita, la ejecución del Scaffolder fallará.
- Si un secreto no está definido, se lanzará una excepción con mensaje explícito.

Despliegue a producción

Desplegar Backstage en un entorno de producción implica varios aspectos adicionales respecto al entorno de desarrollo: seguridad, escalabilidad, almacenamiento persistente, observabilidad y separación de entornos. A continuación se describen las buenas prácticas y configuraciones típicas.

Prácticas recomendadas

Recomendación	Descripción
Separación de entornos	Usar archivos de configuración diferentes por entorno: `app-config.production.yaml` , etc.
Persistencia de base de datos	Usar PostgreSQL con volúmenes persistentes (no SQLite).
Gestión de secretos	Usar un sistema seguro (Vault, K8s secrets, .env seguros) en lugar de archivos planos.
Revisión de configuración TLS	Ejecutar detrás de un reverse proxy (nginx, traefik) con TLS habilitado.
Escalado horizontal	Ejecutar el backend como servicio stateless y escalar frontend con un CDN si aplica.
Observabilidad	Incluir logs estructurados, métricas, trazas (opcional) y alertas sobre el estado del portal.

Separación de configuración por entorno

Backstage permite cargar múltiples archivos `app-config.*.yaml` que se combinan al iniciar:

```
APP_CONFIG_app-config.yaml,app-config.production.yaml yarn start
```

O en Docker:

```
docker run -v $(pwd)/app-config.yaml:/app/app-config.yaml \
           -v $(pwd)/app-config.production.yaml:/app/app-config.production.yaml \
           backstage
```

Variables sensibles

Se recomienda **no incluir valores secretos directamente en los YAML**. En su lugar:

- Usar variables de entorno (`process.env`)
- O sistemas externos como:
  - Kubernetes Secrets
  - HashiCorp Vault
  - AWS Secrets Manager

Ejemplo:

```
integrations:
  github:
    - host: github.com
      token: ${GITHUB_TOKEN}
```

Observabilidad y logs

Backstage no tiene sistema de logging estructurado por defecto, pero permite:

- Configurar salida de logs en formato JSON (para ingestion en Elastic o Loki)
- Añadir herramientas como Prometheus (vía exporters)
- Integración con herramientas como Sentry, Datadog, etc.

También puedes añadir middlewares de log en `packages/backend/src/index.ts` si se requiere personalización.

Despliegue con Docker / Kubernetes

Backstage se puede desplegar de forma robusta con:

- **Docker Compose** (ideal para entornos pequeños o pruebas)
- **Helm charts en Kubernetes** (producción, autoscaling, observabilidad)

Recurso oficial:
👉 https://github.com/backstage/charts

Configuraciones típicas en K8s:

- Ingress con TLS
- PostgreSQL como servicio separado
- SealedSecrets o External Secrets
- Namespace dedicado para Backstage
- Escalabilidad horizontal con HPA

Tests de despliegue

Se recomienda incluir:

- Smoke tests del portal
- Probes de salud en `/healthcheck`
- Validación de dependencias (BD, catálogo, plugins)
- Alarmas ante errores de autenticación o fallos en la ejecución del Scaffolder

Recursos y enlaces útiles

Este apartado recoge enlaces oficiales, herramientas complementarias, recursos comunitarios y ejemplos prácticos que facilitan el trabajo con Backstage y su ecosistema.

Documentación oficial

- Portal oficial de Backstage:
  https://backstage.io

- Documentación técnica:
  https://backstage.io/docs
- Guía de inicio rápido (Create App):
  https://backstage.io/docs/getting-started
- Catálogo de plugins oficiales y comunidad:
  https://backstage.io/plugins

Repositorios y recursos de ejemplo

- Plantilla oficial de creación de app:
  https://github.com/backstage/backstage
- Ejemplo de app contenedorizada (Docker):
  https://github.com/backstage/backstage/tree/master/contrib/docker
- Helm Chart de Backstage (Kubernetes):
  https://github.com/backstage/charts
- Plantillas de Scaffolder de ejemplo:
  https://github.com/backstage/software-templates

Comunidad y soporte

- Discord oficial (comunidad):
  https://discord.com/invite/MUpMjP2
- Foro de discusión (GitHub Discussions):
  https://github.com/backstage/backstage/discussions
- Reportar issues o bugs:
  https://github.com/backstage/backstage/issues
- Calendario de la comunidad (reuniones):
  https://calendar.google.com/calendar/embed?src=backstage.io

Recursos útiles para desarrollo

- TechDocs CLI:
  https://github.com/backstage/techdocs-cli
- Storybook para desarrollo de plugins:
  https://backstage.io/storybook

- Publicador de paquetes npm (si desarrollas plugins): https://www.npmjs.com/org/backstage