## **Introduction to Internal Developer Portals (IDPs)**

### **What is an Internal Developer Portal (IDP)?**

An **Internal Developer Portal (IDP)** is a centralized system that brings together all the tools, documentation, services, APIs, and infrastructure components developers need to work efficiently within an organization. It's essentially a **one-stop shop** for developers.

#### **Think of it as:**

An internal “app store” or dashboard where developers can discover, create, manage, and document software components, services, pipelines, environments, and more.

### **Why Do Organizations Need an IDP?**

Modern development involves:

* Microservices
* CI/CD pipelines
* Cloud infrastructure
* Multiple tools (e.g., Jenkins, GitHub, Kubernetes, Datadog)

This leads to **fragmentation** and **cognitive overload**.

An IDP:

* **Unifies the developer experience**
* **Reduces onboarding time**
* **Improves discoverability of services and tools**
* **Encourages best practices**
* **Improves productivity and collaboration**

**Key Features of a Good IDP**

| **Feature** | **Purpose** |
| --- | --- |
| **Software Catalog** | Discover and manage all services and components. |
| **Templates and Scaffolding** | Quickly generate boilerplate code/projects. |
| **Documentation** | Centralized, searchable technical docs. |
| **CI/CD Views** | See build status, deployment, and logs. |
| **Ownership Tracking** | Know who owns what—crucial for DevOps. |
| **Plugin Extensibility** | Integrate with any internal/external tool via plugins. |

### **📈 Business Benefits**

| **Benefit** | **Explanation** |
| --- | --- |
| **Faster Time to Market** | Reduces setup friction and automates service creation. |
| **Consistency** | Templates and best practices are followed across teams. |
| **Developer Happiness** | Developers spend less time hunting for information. |
| **Reduced Ops Overhead** | Self-service reduces reliance on platform/infra teams. |

### **Real-World Analogy**

Imagine you join a new company as a developer. Without an IDP:

* You need to ask where the repos are.
* Figure out how to request a new microservice.
* Guess who owns which service.
* Dig through 10 tools to understand deployments.

With an IDP:

* Everything is in one place.
* You can scaffold a new service in minutes.
* You instantly see the status, owner, and pipeline of any service.

### **Examples of IDPs**

* **Backstage** (by Spotify)
* **Port**
* **Cortex**
* **OpsLevel**
* **Humanitec**

Backstage is **open-source** and currently the **most popular** choice for building custom IDPs.

## **What is Backstage and How It Helps in Modern Development**

### **What is Backstage?**

**Backstage** is an **open-source Internal Developer Portal framework** created by **Spotify**. It provides a unified interface for developers to **manage all aspects of their software lifecycle**—from creation and deployment to documentation and monitoring.

It is highly **extensible**, **plugin-based**, and **customizable**, allowing organizations to integrate all their developer tools into one central hub.

### **Core Idea Behind Backstage**

“Bring order to your microservices chaos.”

As organizations scale, they accumulate hundreds or thousands of services, APIs, and infrastructure pieces. Backstage offers a **developer-friendly abstraction layer** that ties all of these together.

### **Key Features of Backstage**

| **Feature** | **Description** |
| --- | --- |
| **Software Catalog** | Central registry for services, libraries, and systems, complete with ownership and metadata. |
| **Software Templates** | Scaffolding system to generate new services using standardized templates (e.g., React app, Node service, Python API). |
| **TechDocs** | Built-in documentation system using Markdown + MkDocs for internal documentation. |
| **Plugins** | Integrate CI/CD tools, monitoring, cost analysis, Kubernetes, secrets management, and more. |
| **Scalability** | Used by companies with thousands of services (e.g., Spotify, Netflix, Expedia). |
| **Customization** | Fully customizable UI and backend, tailored to your infrastructure. |

**Example Use Cases in Modern Development**

| **Use Case** | **Backstage Solution** |
| --- | --- |
| Creating a new microservice | Use a **software template** to scaffold it with CI/CD, logging, monitoring, and more. |
| Finding service owners | Use the **catalog** to find ownership metadata, Slack handles, and code repos. |
| Reading internal documentation | Use **TechDocs** for integrated Markdown-based docs. |
| Monitoring build/deployment status | Integrate Jenkins, GitHub Actions, ArgoCD, etc., via **plugins**. |
| Working with Kubernetes | Use the **Kubernetes plugin** to view pod/deployment status directly inside Backstage. |

**Tools It Replaces or Unifies**

Backstage doesn’t replace tools—it **integrates them**.

| **Traditional Tool** | **Backstage Integration** |
| --- | --- |
| GitHub/GitLab | Pull requests, repositories, and CI/CD integrations |
| Jenkins/ArgoCD | Build, deploy, and pipeline visualization |
| Kubernetes/Docker | Pod status, service health, cluster views |
| PagerDuty/Sentry/Datadog | Monitoring, alerting, and logging via plugins |
| Confluence/Docs Sites | Built-in documentation with TechDocs |

### **How It Helps Developers and Teams**

| **Role** | **Benefits** |
| --- | --- |
| **Developers** | Less context switching, faster ramp-up, consistent experience |
| **SREs/Platform Engineers** | Easier onboarding, reduced support requests, centralized visibility |
| **Engineering Managers** | Metrics on service health, ownership, documentation coverage |
| **DevOps** | CI/CD transparency, reduced configuration duplication |

### **Who’s Using Backstage?**

* Spotify (creator)
* Netflix
* American Airlines
* Expedia
* Zalando
* Epic Games
* Shopify

All these companies use Backstage to tame complex microservices and enable developer self-service at scale.

## **How Backstage Fits Into Your Development and Delivery Workflows**

### **Overview**

Modern software development involves a wide range of tools and processes:

* Source control (e.g., GitHub, GitLab)
* CI/CD pipelines (e.g., Jenkins, GitHub Actions, ArgoCD)
* Infrastructure (e.g., Kubernetes, Terraform)
* Monitoring and observability (e.g., Prometheus, Grafana)
* Documentation (e.g., Confluence, Markdown-based tools)
* Service discovery and ownership tracking

Backstage acts as a **central developer interface** that connects all of these components into a unified developer experience.

### **Key Workflow Stages and Backstage's Role**

#### **1. Service Creation and Standardization**

* **Without Backstage:** Developers manually copy existing services or templates, often inconsistently.
* **With Backstage:** Developers use standardized software templates to generate a new service with pre-wired CI/CD, monitoring, testing, and documentation.

**Benefit:** Promotes best practices and reduces onboarding time.

#### **2. Development and Code Management**

* Backstage integrates with Git repositories (e.g., GitHub, GitLab).
* Developers can browse service code, open pull requests, and view repository metadata from within Backstage.

**Benefit:** Centralized view of all projects and codebases.

#### **3. CI/CD Integration**

* Backstage connects to CI/CD systems such as Jenkins, CircleCI, GitHub Actions, and ArgoCD.
* Developers can track build status, test results, and deployment history from within the portal.

**Benefit:** Reduces the need to jump between tools and provides real-time visibility.

#### **4. Deployment and Operations**

* Integration with Kubernetes or cloud platforms lets developers see pod health, service status, logs, and metrics directly in Backstage.
* Also integrates with monitoring tools like Prometheus, Sentry, or Datadog.

**Benefit:** Simplifies debugging and monitoring without requiring direct Kubernetes access.

#### **5. Documentation Access**

* Backstage supports internal documentation using TechDocs, which converts Markdown into searchable, browsable docs.
* Documentation is tied to each service in the catalog.

**Benefit:** Encourages better documentation and makes it easily discoverable.

#### **6. Ownership and Discovery**

* The Software Catalog stores metadata about services: owner, system, domain, team, links, documentation, status, etc.
* Developers can quickly find who owns a service, open Slack or email links, or raise issues.

**Benefit:** Eliminates tribal knowledge and speeds up support and incident resolution.

### **Typical Developer Workflow with Backstage**

1. Log into Backstage.
2. Use a software template to scaffold a new Node.js service.
3. Automatically generate a GitHub repository with a CI/CD pipeline.
4. Push code, track build status from the same interface.
5. View deployment status in Kubernetes via the plugin.
6. Read or update TechDocs from the catalog.
7. See alerts and metrics in the integrated monitoring dashboard.

### **Alignment with Platform Engineering**

Backstage helps platform teams provide a curated developer experience. They can:

* Define software templates that include standard tooling.
* Offer plugins as building blocks (e.g., CI/CD, monitoring).
* Reduce reliance on support tickets by offering self-service.

This allows developers to focus on building features rather than navigating tooling.

## **Overview of the Backstage Architecture and Key Components**

### **High-Level Architecture**

**Backstage is composed of two main layers:**

1. **Frontend (React-based web application)**
2. **Backend (Node.js server with a plugin system)**

**These layers communicate through APIs and serve as the foundation for a modular, extensible developer portal.**

### **Core Components of Backstage**

#### **1. App (Frontend)**

* **Written in React.**
* **Provides the UI that developers interact with.**
* **Consists of plugins, routes, and themes.**
* **The app is fully customizable—you can add your company branding, rearrange layout, or add custom pages.**

#### **2. Plugins**

* **Backstage is built entirely on a plugin-based architecture.**
* **Both frontend and backend can include plugins.**
* **Examples:**
  + **Catalog Plugin – Displays and manages services.**
  + **TechDocs Plugin – Renders internal documentation.**
  + **Kubernetes Plugin – Displays service health, deployments.**
  + **CI/CD Plugins – Jenkins, GitHub Actions, ArgoCD.**
* **You can build custom plugins for internal tools.**

#### **3. Software Catalog**

* **Central to Backstage.**
* **Stores metadata about all software components: services, websites, libraries, APIs.**
* **Catalog data is defined using YAML files (catalog-info.yaml) stored in code repositories.**
* **Supports concepts like ownership, domain, system, and lifecycle stage.**

#### **4. TechDocs**

* **Documentation system integrated directly into Backstage.**
* **Based on MkDocs, using Markdown stored in the same repository as code.**
* **Automatically builds and renders internal docs as part of the Software Catalog.**

#### **5. Backend (Node.js Express Server)**

* **Hosts backend plugins and handles service integrations.**
* **Offers APIs for reading catalog metadata, integrating with GitHub, Kubernetes, CI/CD tools, etc.**
* **Can run as a standalone service or be deployed with the frontend in one container.**

#### **6. Authentication & Authorization**

* **Backstage supports integration with enterprise auth providers (Google, GitHub, Azure AD, Okta).**
* **You can enforce RBAC (role-based access control) for secure access to services or actions.**

### **Backstage Data Flow**

1. **Developer accesses the frontend (React app).**
2. **React components render plugin UIs (Catalog, Docs, etc.).**
3. **Plugins make requests to the backend APIs.**
4. **Backend plugins fetch data from external systems (e.g., GitHub, Kubernetes, Jenkins).**
5. **Data is processed and returned to the frontend via plugin APIs.**

### **Deployment Architecture Options**

* **Monolith: Run frontend and backend in one container (simple for dev/testing).**
* **Split: Frontend and backend run as separate services (recommended for production).**
* **Cloud-native: Run Backstage in Kubernetes, with external databases and storage (e.g., PostgreSQL, S3 for docs).**

### **Persistent Components**

* **Database: Used for storing catalog entities and plugin metadata (commonly PostgreSQL).**
* **Object Storage: Required for storing generated TechDocs content (e.g., S3, GCS).**
* **Authentication Provider: Used for sign-in and access control.**

### **Summary**

| **Component** | **Purpose** |
| --- | --- |
| **Frontend** | **React-based UI for all users** |
| **Plugins** | **Modular feature units that add capabilities** |
| **Backend** | **Node.js server that integrates with external systems** |
| **Catalog** | **Central registry for all software components** |
| **TechDocs** | **Integrated documentation system** |
| **Auth** | **Enterprise login and role management** |
| **Storage** | **Stores persistent data and generated docs** |

## **Differences Between Frontend and Backend Components in Backstage**

**Understanding the separation between the frontend and backend in Backstage is critical for customization, plugin development, and system design. Backstage follows a modular monorepo architecture where both frontend and backend live in the same codebase but are logically and operationally distinct.**

### **1. Frontend in Backstage**

#### **Description:**

* **Built using React.**
* **Provides the user interface that developers interact with.**
* **Handles rendering of pages, navigation, themes, UI plugins, and visualizations.**

#### **Responsibilities:**

* **Display catalog entities and metadata.**
* **Render documentation (TechDocs).**
* **Integrate UI components from plugins (e.g., CI/CD dashboards, Kubernetes views).**
* **Handle routing and navigation between plugin pages.**
* **Perform client-side authentication flow (e.g., redirect to sign-in).**

#### **Key Technologies:**

* **React**
* **TypeScript**
* **Material-UI**
* **Frontend plugins (e.g., catalog, TechDocs, scaffolder)**

#### **Example Frontend Plugins:**

* **@backstage/plugin-catalog**
* **@backstage/plugin-techdocs**
* **@backstage/plugin-kubernetes**
* **@backstage/plugin-github-actions**

#### **Developer Tasks:**

* **Customize layout or navigation.**
* **Add new plugins.**
* **Theme branding (colors, logos, etc.).**
* **Implement custom UI components or routes.**

### **2. Backend in Backstage**

#### **Description:**

* **Built using Node.js and Express.**
* **Provides the server-side logic and integration layer.**
* **Handles data fetching, storage, and external service communication.**

#### **Responsibilities:**

* **Interact with third-party systems (e.g., GitHub, Kubernetes, Jenkins).**
* **Expose APIs for frontend to consume.**
* **Manage catalog entities and metadata.**
* **Build and serve TechDocs documentation.**
* **Handle authentication and authorization.**
* **Schedule background tasks (e.g., entity refresh, doc builds).**

#### **Key Technologies:**

* **Node.js**
* **TypeScript**
* **Express.js**
* **Backend plugins and routers**

#### **Example Backend Plugins:**

* **@backstage/plugin-catalog-backend**
* **@backstage/plugin-techdocs-backend**
* **@backstage/plugin-auth-backend**
* **@backstage/plugin-kubernetes-backend**

#### **Developer Tasks:**

* **Implement custom integrations (e.g., a proprietary CI system).**
* **Add backend APIs or proxy routes.**
* **Connect to databases or cloud services.**
* **Extend or override catalog processing.**

### **Summary Table**

| **Feature** | **Frontend** | **Backend** |
| --- | --- | --- |
| **Language** | **TypeScript + React** | **TypeScript + Node.js** |
| **Framework** | **React, Material-UI** | **Express.js** |
| **Purpose** | **UI rendering and user interaction** | **Business logic, APIs, integrations** |
| **Runs In** | **Browser** | **Server** |
| **Extensibility** | **UI Plugins, Routes, Components** | **Backend Plugins, Routers** |
| **Common Tasks** | **Add pages, customize themes** | **Connect external tools, process catalog data** |
| **Example Plugin** | **plugin-catalog (UI)** | **plugin-catalog-backend (API logic)** |

### **Real-World Analogy**

**Think of Backstage like a modern web application:**

* **The frontend is the dashboard your developers see.**
* **The backend is the engine room that powers it, pulls in data, authenticates users, and manages external integrations.**

**Both are part of the same ecosystem but serve different roles.**

## **Introduction to TechDocs**

**TechDocs is Backstage's built-in documentation feature that enables organizations to manage and display internal documentation alongside their software catalog. It is designed to make documentation a first-class citizen in the developer experience and helps ensure documentation is discoverable, consistent, and versioned alongside the code.**

### **What is TechDocs?**

**TechDocs is an integrated documentation site generator within Backstage, built on top of MkDocs, a static site generator that uses Markdown files. It enables software teams to:**

* **Write documentation using Markdown.**
* **Store it in the same repo as their codebase.**
* **Render and serve the documentation inside Backstage as part of the software component view.**

### **Why TechDocs?**

**Traditionally, engineering teams struggle to maintain consistent, discoverable documentation. TechDocs addresses these problems by:**

| **Problem** | **How TechDocs Helps** |
| --- | --- |
| **Documentation is scattered across platforms** | **Centralizes docs in Backstage** |
| **Documentation is outdated** | **Encourages co-locating docs with source code** |
| **Inconsistent formats and tools** | **Standardizes format using MkDocs and Markdown** |
| **Difficult to find relevant docs** | **Integrates docs directly into the software catalog** |

### **Core Features of TechDocs**

1. **Markdown-Based**
   * **Authors use simple, readable .md files.**
   * **Supports standard Markdown syntax with extensions.**
2. **MkDocs Compatibility**
   * **Follows MkDocs structure (mkdocs.yml as the config file).**
   * **You can use any MkDocs-compatible plugin or theme.**
3. **Documentation Lives with Code**
   * **Recommended practice is to include a docs/ folder and mkdocs.yml in the same repo as your service or library.**
4. **Automatic Rendering in Backstage**
   * **Once configured, TechDocs renders the Markdown content into HTML.**
   * **The documentation is displayed under the “Docs” tab for each catalog entity.**
5. **Versioned and Git-backed**
   * **Documentation is tied to the same Git repository and versioning as the code.**
   * **This ensures that the docs reflect the actual state of the code at a given point.**
6. **Search and Navigation**
   * **Built-in navigation pane, headings, and search features.**
   * **Backstage-wide search can include TechDocs content.**

### **TechDocs Architecture Overview**

**The TechDocs system in Backstage involves:**

* **Docs Source: Markdown files and mkdocs.yml in your repository.**
* **Build Phase:**
  + **TechDocs can either build the documentation:**
    - ***Locally* at runtime (techdocs.builder: 'local'), or**
    - ***Externally*, using CI pipelines and upload output to storage (techdocs.builder: 'external').**
* **Storage:**
  + **The generated HTML output is stored in an object storage bucket (e.g., S3, GCS, Azure Blob).**
* **TechDocs Backend Plugin:**
  + **Serves the static content from storage to the frontend.**
* **Frontend Plugin:**
  + **Renders the content inside the Backstage UI.**

### **Benefits of Using TechDocs**

* **Unified experience: Docs are part of the developer portal, not scattered links.**
* **Low overhead: Write Markdown, push to Git—Backstage handles the rest.**
* **Contextual: Docs are tied directly to the software entities in the catalog.**
* **Scalable: Works across hundreds of services, each with its own docs.**
* **Customizable: Extend MkDocs configuration, styling, and plugins.**

### **Typical Directory Structure for TechDocs**

**In a service repository:**

**my-service/**

**├── docs/**

**│ ├── index.md**

**│ └── architecture.md**

**├── mkdocs.yml**

**├── src/**

**└── catalog-info.yaml**

* **docs/: contains Markdown files.**
* **mkdocs.yml: configuration file for TechDocs.**
* **catalog-info.yaml: entity metadata that links the docs to Backstage.**

### **Summary**

| **Feature** | **Description** |
| --- | --- |
| **Based On** | **MkDocs and Markdown** |
| **Integration** | **Renders in Backstage UI** |
| **Source** | **Code repositories (docs live with code)** |
| **Build Options** | **Local or external (CI-based)** |
| **Storage** | **Object storage (S3, GCS, Azure, etc.)** |
| **Benefits** | **Centralized, consistent, contextual documentation** |

## **Advanced TechDocs Features and Publishing Pipelines**

**Once you’re familiar with the basics of TechDocs in Backstage—Markdown files rendered with MkDocs—the next step is to understand the advanced features and publishing pipeline options that make TechDocs scalable and production-ready for enterprise environments.**

### **1. TechDocs Build Strategies**

**There are two main build strategies in TechDocs:**

| **Mode** | **Description** | **Use Case** |
| --- | --- | --- |
| **local** | **Docs are built on demand by the Backstage backend when a user views them.** | **Good for testing and local development.** |
| **external** | **Docs are prebuilt (e.g., via CI/CD) and uploaded to cloud storage.** | **Best for production environments to reduce latency and backend load.** |

**You can configure this via:**

**techdocs:**

**builder: 'local' # or 'external'**

### **2. TechDocs Publishing Pipeline (External Publishing)**

**In external mode, the TechDocs publishing pipeline works like this:**

1. **Docs Authoring**
   * **Developers write .md files and mkdocs.yml in the code repository.**
2. **CI/CD Build Step**
   * **A CI workflow (e.g., GitHub Actions, GitLab CI, Jenkins) runs:**
     + **Installs MkDocs and TechDocs CLI.**
     + **Builds static site output (site/ folder).**
     + **Uploads the content to cloud storage (S3, GCS, Azure Blob).**
3. **Storage and Serving**
   * **The techdocs-backend plugin serves content from the storage location.**
   * **No need to rebuild docs at runtime.**

### **3. TechDocs CLI**

**Backstage provides the @techdocs/cli tool to automate build and publish tasks.**

#### **Key commands:**

**techdocs build # Builds docs locally into HTML (site/)**

**techdocs publish # Uploads the generated site to configured storage**

**techdocs serve # Starts a local HTTP server for previewing docs**

**These commands are typically used in CI pipelines.**

### **4. Cloud Storage Backends**

**TechDocs supports multiple object storage backends for storing and serving documentation:**

| **Provider** | **Configuration Key** |
| --- | --- |
| **AWS S3** | **awsS3** |
| **Google Cloud Storage (GCS)** | **googleGcs** |
| **Azure Blob Storage** | **azureBlobStorage** |
| **Local Filesystem (Dev Only)** | **local** |

**Example configuration for S3:**

**techdocs:**

**publisher:**

**type: 'awsS3'**

**awsS3:**

**bucketName: 'my-techdocs-bucket'**

**credentials:**

**accessKeyId: ${AWS\_ACCESS\_KEY\_ID}**

**secretAccessKey: ${AWS\_SECRET\_ACCESS\_KEY}**

### **5. Documentation Metadata in Catalog**

**To link TechDocs with a catalog entity, the catalog-info.yaml must include:**

**metadata:**

**name: my-service**

**annotations:**

**backstage.io/techdocs-ref: dir:.**

**Other formats supported: url:, github:, gitlab:, etc., depending on source location.**

### **6. Custom MkDocs Configuration**

**TechDocs supports custom configuration of MkDocs via mkdocs.yml. You can:**

* **Add plugins (e.g., search, diagrams)**
* **Define navigation menus**
* **Apply themes or use a corporate theme**
* **Use Markdown extensions**

**Example mkdocs.yml:**

**site\_name: My Service Docs**

**nav:**

**- Home: index.md**

**- Architecture: architecture.md**

**plugins:**

**- techdocs-core**

**- search**

**theme:**

**name: material**

### **7. Search Integration**

**TechDocs supports integration into Backstage’s unified search interface.**

* **Requires Backstage’s search backend to be configured.**
* **TechDocs content is indexed during build time.**
* **Users can search for terms across catalog metadata, docs, and more.**

**This requires:**

* **@backstage/plugin-search-backend-module-techdocs**
* **@backstage/plugin-search-backend**
* **@backstage/plugin-search**

### **8. Security Considerations**

* **Ensure bucket-level access controls for object storage (S3/GCS).**
* **Use signed URLs or backend proxying for secure delivery.**
* **Sanitize input during Markdown rendering to avoid XSS.**

### **9. Multi-Repo and Monorepo Support**

**TechDocs supports both:**

* **Multi-repo (each component has its own docs/ folder)**
* **Monorepo (centralized docs per team/service in a single repo)**

**Annotation flexibility (dir:, url:, etc.) supports diverse repo setups.**

### **10. Benefits of Advanced Setup**

| **Feature** | **Benefit** |
| --- | --- |
| **External builds** | **Faster page loads and better performance** |
| **Cloud publishing** | **Scales across thousands of components** |
| **CI-based pipeline** | **Consistent, repeatable builds** |
| **Search integration** | **Faster doc discovery** |
| **Custom themes/plugins** | **Align with branding and usability standards** |

### **Summary**

**Advanced TechDocs features make documentation:**

* **Production-ready**
* **Scalable**
* **Discoverable**
* **Secure**

**By adopting an external publishing pipeline with CI/CD and cloud storage, you ensure that TechDocs meets enterprise needs without compromising performance or developer experience.**

## **Deep Dive into React Architecture in Backstage**

**Backstage is built with a modern web development stack, and at its core, the frontend is powered by React. Understanding the React architecture of Backstage is crucial for extending the platform, building custom plugins, and tailoring the UI to your organization’s needs.**

### **1. High-Level Overview**

* **Frontend Framework: React (with TypeScript)**
* **UI Component Library: Material-UI (MUI)**
* **Routing: React Router**
* **State Management: Mostly React hooks and context APIs**
* **Plugin-Based Architecture: Each feature is modularized as a plugin**
* **Theming & Layout: Customizable themes and layout system via APIs**

### **2. Core Concepts in Backstage React Architecture**

#### **a. Plugins**

* **Everything is a plugin.**
* **Each feature—catalog, docs, search, scaffolder—is implemented as a separate React plugin.**
* **Plugins are developed independently and plugged into the main app.**

**Example:**

**import { CatalogPage } from '@backstage/plugin-catalog';**

**const routes = (**

**<Route path="/catalog" element={<CatalogPage />} />**

**);**

#### **b. App Framework**

**Backstage provides a set of APIs and components to bootstrap a frontend app:**

* **createApp() initializes the app and registers plugins.**
* **App root is rendered using AppRouter.**

**App Entry Point:**

**const app = createApp({**

**apis,**

**plugins: [catalogPlugin, techdocsPlugin],**

**components: { ... },**

**});**

### **3. Plugin Anatomy**

**A Backstage plugin is a package that exports:**

| **Part** | **Purpose** |
| --- | --- |
| **plugin.ts** | **Entry point for the plugin** |
| **routes.ts** | **Defines routes/components exposed by the plugin** |
| **components/** | **UI components specific to the plugin** |
| **api.ts** | **API contracts if needed** |
| **index.ts** | **Export interfaces for consumption** |

**Example structure:**

**plugins/**

**└── catalog/**

**├── src/**

**│ ├── plugin.ts**

**│ ├── routes.ts**

**│ ├── components/**

**│ └── api.ts**

**└── index.ts**

### **4. Routing System**

**Backstage uses React Router v6 for navigation.**

* **Each plugin contributes routes to the global router.**
* **Main routes are declared in App.tsx or equivalent.**

**Example:**

**<Route path="/catalog" element={<CatalogPage />} />**

**<Route path="/docs/:namespace/:kind/:name/\*" element={<TechDocsReaderPage />} />**

### **5. Theme System**

* **Uses Material-UI theming with Backstage custom extensions.**
* **You can override default styles, component looks, and brand identity.**

**Theme Customization:**

**const myTheme = createTheme({**

**palette: {**

**primary: {**

**main: '#0047AB',**

**},**

**},**

**defaultPageTheme: 'home',**

**});**

**The theme is passed to createApp() or wrapped with <ThemeProvider>.**

### **6. Layout and UI Components**

* **Backstage offers reusable layout components:**
  + **<Header />, <Sidebar />, <Content />, <Page />**
* **These components follow consistent spacing, responsive design, and theming.**

**Example Page Layout:**

**export const MyPage = () => (**

**<Page themeId="tool">**

**<Header title="My Plugin" subtitle="Custom page" />**

**<Content>**

**<MyComponent />**

**</Content>**

**</Page>**

**);**

### **7. APIs and Context**

**Backstage plugins share functionality via API factories and context providers.**

* **APIs are registered globally and injected into components using React hooks.**
* **Contextual APIs: Identity, Scaffolder, Catalog, Config, etc.**

**Example:**

**const identity = useApi(identityApiRef);**

**console.log(identity.getUserId());**

### **8. Composability and Extension Points**

* **Components expose extension points (hooks, props) for overriding default behavior.**
* **Plugins can consume or contribute functionality to others.**
* **Custom tabs, actions, widgets can be injected dynamically.**

### **9. Bundling and Code Splitting**

* **Uses Webpack (under the hood via @backstage/cli) for building.**
* **Code-splitting is supported to optimize large apps.**
* **Each plugin is a separate chunk.**

### **10. Dev and Hot Reloading**

* **yarn dev runs the app with hot module reload using @backstage/cli.**
* **Plugins are developed live without full rebuilds.**

### **Summary of Architecture Components**

| **Component** | **Role** |
| --- | --- |
| **React** | **UI Library** |
| **Material-UI** | **Component styling and theming** |
| **React Router** | **Routing and navigation** |
| **Plugins** | **Modular features** |
| **APIs** | **Shared services and context** |
| **Theme** | **Brand and visual identity** |
| **Layouts** | **Consistent structure for pages** |
| **Webpack** | **Module bundler with code splitting** |

### **Conclusion**

**Backstage’s React architecture offers a modular, scalable, and customizable framework for building developer portals. By leveraging modern React patterns and a plugin-based design, it empowers teams to extend, customize, and build UIs that align with their workflows and organizational needs.**

## **SSO Integration with OAuth, Okta, and SAML in Backstage**

**Single Sign-On (SSO) integration is a key enterprise feature in Backstage that enables developers to authenticate once and gain access to all authorized services seamlessly. Backstage supports SSO via OAuth 2.0, SAML, and identity providers like Okta, GitHub, Google, Azure AD, and more.**

### **1. Why SSO in Backstage?**

* **Unified access control: Centralized user identity across all Backstage plugins and components.**
* **Security: Reduces password fatigue and risk of credential compromise.**
* **Auditability: Tie user actions in Backstage to corporate identity.**
* **Seamless experience: No need to log in separately to each service or tool integrated into Backstage.**

### **2. Backstage Authentication Overview**

**Backstage uses a plugin-based authentication system, implemented in its backend. The main pieces are:**

* **auth-backend plugin: Provides a framework for integrating authentication providers.**
* **sign-in page in frontend: Presents login options to users.**
* **session handling: Uses JWT or cookie-based sessions.**
* **identity API: Standardizes user identity across plugins.**

### **3. Common Authentication Flows**

| **Flow** | **Supported Protocol** | **Providers** |
| --- | --- | --- |
| **Authorization Code Flow** | **OAuth 2.0 / OIDC** | **GitHub, Google, Okta, Azure AD** |
| **SAML 2.0** | **SAML** | **Okta, OneLogin, ADFS, etc.** |

### **4. OAuth 2.0 Integration**

**Backstage supports OAuth2-based identity providers like GitHub, Google, and Azure AD via the @backstage/plugin-auth-backend and passport strategies.**

#### **Key Steps:**

1. **Configure the provider in app-config.yaml:**

**auth:**

**environment: development**

**providers:**

**google:**

**development:**

**clientId: ${GOOGLE\_CLIENT\_ID}**

**clientSecret: ${GOOGLE\_CLIENT\_SECRET}**

1. **Register the provider in packages/backend/src/plugins/auth.ts:**

**import { google } from '@backstage/plugin-auth-backend';**

**router.use('/google', await google.createRouter({**

**clientId,**

**clientSecret,**

**callbackUrl,**

**}));**

1. **Use the provider in the frontend sign-in page:**

**<SignInPage**

**providers={[**

**'google',**

**'github',**

**]}**

**/>**

1. **Result: After login, Backstage returns a Backstage identity token that plugins can use.**

### **5. Okta Integration (OAuth / OIDC or SAML)**

**Okta can be used in two ways:**

* **As an OIDC provider (simpler with OAuth).**
* **As a SAML 2.0 identity provider (more complex, but sometimes required).**

#### **a. Okta as OAuth/OIDC:**

1. **Create an Okta app with OIDC.**
2. **Configure in app-config.yaml:**

**auth:**

**providers:**

**okta:**

**development:**

**clientId: ${OKTA\_CLIENT\_ID}**

**clientSecret: ${OKTA\_CLIENT\_SECRET}**

**audience: https://your-okta-domain.okta.com**

#### **b. Okta as SAML Provider:**

* **SAML requires additional configuration such as SAML metadata, certs, and assertion mappings.**
* **Requires a custom integration or third-party SAML support in the backend.**

### **6. SAML Integration**

**While Backstage doesn't have native SAML support out-of-the-box like it does for OAuth providers, SAML can be integrated through custom Passport strategies or reverse proxies like AuthProxy.**

#### **Approach:**

* **Use passport-saml strategy in the backend.**
* **Define your SAML metadata, assertion consumer service URL, and certificate.**
* **Extract user identity from the SAML response and create a Backstage token.**

#### **Alternative:**

* **Place Backstage behind an SSO proxy (e.g.,** [**oauth2-proxy**](https://oauth2-proxy.github.io/oauth2-proxy/)**).**
* **The proxy handles SAML, and injects headers (e.g., X-Forwarded-User).**
* **Backstage reads these headers to establish identity.**

### **7. Backstage Identity Resolution**

**Once authenticated, Backstage maps users to internal identities using the identity API.**

**Example:**

**const identity = await identityClient.getBackstageIdentity({ token });**

**console.log(identity.userEntityRef); // e.g., "user:default/jdoe"**

**This enables:**

* **Role-based access controls**
* **Catalog ownership attribution**
* **Audit trails**

### **8. Multi-Provider Support**

**Backstage supports multiple auth providers simultaneously. For example:**

**auth:**

**providers:**

**github: ...**

**google: ...**

**okta: ...**

**The SignInPage can present multiple buttons:**

**<SignInPage providers={['github', 'google', 'okta']} />**

**Each button initiates a different OAuth flow.**

### **9. Common Identity Scenarios Supported**

| **Use Case** | **Approach** |
| --- | --- |
| **GitHub Enterprise login** | **GitHub OAuth** |
| **Internal AD users** | **Okta with OIDC or SAML** |
| **Google Workspace users** | **Google OAuth** |
| **Multiple providers** | **Combined via SignInPage config** |
| **Centralized user mapping** | **Identity resolvers in backend** |

### **10. Security Considerations**

* **Always use HTTPS for auth flows and callbacks.**
* **Store secrets in secure vaults, not directly in app-config.yaml.**
* **Limit scopes and access tokens to minimum required permissions.**
* **Regularly rotate OAuth client secrets and certificates.**

### **Summary**

**Backstage’s flexible authentication framework allows seamless SSO integration using:**

* **OAuth 2.0/OIDC: Google, GitHub, Azure AD, Okta**
* **SAML 2.0: Via custom or proxy-based solutions**
* **Custom Providers: Supported via Passport.js strategy adapters**

**With this setup, organizations can enforce security, streamline access, and align with enterprise identity governance practices.**

## 

## **RBAC and Permission Policies Using the Backstage Permission Framework**

**Role-Based Access Control (RBAC) and fine-grained permissions are essential in any internal developer portal to ensure only authorized users can access or modify specific resources. Backstage provides a flexible permission framework that enables you to implement RBAC, attribute-based access control, and custom permission policies across plugins.**

### **1. Why RBAC in Backstage?**

* **Security: Prevent unauthorized users from accessing sensitive data or actions.**
* **Governance: Enforce rules around who can publish docs, scaffold components, or modify metadata.**
* **Auditability: Know who did what, when, and why.**
* **Scalability: Manage permissions across large organizations without hardcoding them into each plugin.**

### **2. Backstage Permission Framework Overview**

**Backstage introduced the Permission Framework to centralize access control logic across plugins and services.**

**It consists of three major components:**

| **Component** | **Description** |
| --- | --- |
| **Policy** | **A centralized place to define permission rules** |
| **Permission API** | **Allows plugins to ask if a user can perform an action** |
| **Resource Reference** | **Represents the thing (entity, resource) being acted on** |

### **3. Key Concepts in the Permission Framework**

#### **a. Permissions**

**A Permission defines an action a user wants to perform, like reading or updating a resource.**

**Example:**

**const catalogEntityReadPermission: Permission = {**

**name: 'catalog.entity.read',**

**attributes: { ... }**

**};**

**Permissions can be:**

* **Resource-based (e.g., read a catalog entity)**
* **Global (e.g., access the scaffolder plugin)**

#### **b. Policies**

**A policy is the core logic that decides:**

* **If a given user can perform a given action.**
* **On a given resource.**

**Policies can use attributes like:**

* **User identity (user:default/jdoe)**
* **Resource type (Component, API)**
* **Conditions (is owner, is admin)**

### **4. Anatomy of a Permission Decision**

1. **A plugin (e.g., TechDocs, Scaffolder) requests a permission decision:**

**const decision = await permissionApi.authorize({**

**permission: somePermission,**

**resourceRef: 'component:default/my-service',**

**});**

1. **Backstage calls the policy you’ve defined to evaluate that request.**
2. **The policy returns one of:**

* **ALLOW**
* **DENY**
* **CONDITIONAL**

### **5. Writing Custom Policies**

**Policies are implemented in the backend in TypeScript.**

**Example Policy (basic RBAC):**

**const myPolicy: PermissionPolicy = async (request) => {**

**const { permission, resourceRef, identity } = request;**

**if (!identity) return { result: AuthorizeResult.DENY };**

**if (identity.userEntityRef === 'user:default/admin') {**

**return { result: AuthorizeResult.ALLOW };**

**}**

**if (permission.name === 'catalog.entity.read') {**

**return { result: AuthorizeResult.ALLOW };**

**}**

**return { result: AuthorizeResult.DENY };**

**};**

**Install it:**

**createRouter({ policy: myPolicy, ... });**

### **6. Example Use Case Scenarios**

| **Use Case** | **Implementation** |
| --- | --- |
| **Only allow admins to register new components** | **Custom policy checks if user is admin** |
| **Users can only read docs of entities they own** | **Use entity ownership metadata in permission** |
| **Only SREs can update Scaffolder templates** | **Restrict scaffolder.template.update by group** |
| **Read-only access for interns** | **Deny all create, update permissions** |

### **7. Combining RBAC with Attributes**

**While RBAC is role-based, the permission framework supports attribute-based checks too.**

**Example:**

* **Allow catalog.entity.read only if user is an owner:**

**if (isOwner(resourceRef, identity.userEntityRef)) {**

**return { result: AuthorizeResult.ALLOW };**

**}**

**This creates an attribute-based RBAC model.**

### **8. Default Permissions in Plugins**

**Some Backstage plugins already have built-in permissions, such as:**

| **Plugin** | **Permission** |
| --- | --- |
| **Catalog** | **catalog.entity.read, catalog.entity.delete** |
| **TechDocs** | **techdocs.read** |
| **Scaffolder** | **scaffolder.action.execute, scaffolder.template.publish** |

**You can override the default policy to control these permissions.**

### **9. Conditional Permissions**

**A permission may be conditionally allowed, such as:**

* **If user is owner**
* **If entity is in a certain namespace**

**In this case, the plugin must evaluate the condition at runtime.**

**return {**

**result: AuthorizeResult.CONDITIONAL,**

**conditions: {**

**pluginId: 'catalog',**

**resourceType: 'entity',**

**rule: 'IS\_ENTITY\_OWNER',**

**params: { ... }**

**}**

**};**

### **10. Managing User Roles**

**Roles (like admin, developer, viewer) are not built-in. You can:**

* **Use groups from identity providers (e.g., Okta, GitHub teams)**
* **Use Group entities in the catalog**
* **Map roles using custom logic in your policy**

### **11. Real-World Example**

**Suppose a developer is trying to scaffold a new service.**

* **Plugin sends: permission = scaffolder.template.create**
* **Backend receives request and checks:**
  + **Is the user part of group devs?**
  + **Is the template type allowed for this group?**
* **Policy returns: ALLOW or DENY**

### **12. Summary**

| **Feature** | **Description** |
| --- | --- |
| **Permission API** | **Central interface for evaluating access** |
| **Custom Policy** | **Where you define your RBAC logic** |
| **Plugins** | **Can enforce permissions before taking actions** |
| **Flexible Models** | **RBAC, ABAC, or hybrid** |
| **Role Integration** | **Via identity provider or catalog groups** |

### **Conclusion**

**Backstage’s permission framework provides a centralized, extensible, and powerful system for implementing RBAC and fine-grained access controls. Whether you’re managing developer privileges, protecting sensitive docs, or limiting scaffold access, the framework allows you to define and enforce policies tailored to your organizational structure.**

## **Debugging Common Issues with Backstage**

**Backstage, being a modular and plugin-driven platform, can run into a variety of issues during setup, customization, or regular use. Understanding common failure points and having structured debugging strategies are essential for maintaining a healthy developer portal.**

### **1. Common Categories of Issues**

| **Category** | **Description** |
| --- | --- |
| **Configuration Errors** | **Invalid or missing values in the app-config.yaml file or environment variables** |
| **Plugin Failures** | **Issues with plugin integration, API contracts, or misconfiguration** |
| **Build Issues** | **Errors during the frontend or backend build process** |
| **Runtime Errors** | **Errors that occur while running the dev server or in production** |
| **Network/API Issues** | **Backend failing to connect to external services (e.g., GitHub, Kubernetes, LDAP)** |
| **Permission Errors** | **Unauthorized access due to improper RBAC policies** |

### **2. Common Debugging Scenarios**

#### **a. App Fails to Start (Dev Mode or Production)**

* **Check Console Output: Run yarn dev and look for startup errors.**
* **Common Causes:**
  + **Invalid app-config.yaml**
  + **Missing environment variables**
  + **Improper plugin registration**

#### **b. Plugin Not Working or Fails to Load**

* **Frontend Symptoms:**
  + **Blank screen or error overlay**
  + **Console logs with Module not found or Plugin not registered**
* **Backend Symptoms:**
  + **Plugin routes not found**
  + **API calls failing with 404/500**
* **Fixes:**
  + **Check packages/app/src/plugins.ts for proper plugin import**
  + **Verify backend route registration**

#### **c. Catalog Entities Not Visible**

* **Common Reasons:**
  + **Incorrect entity YAML**
  + **Location not registered**
  + **Catalog ingestion failure**
* **Tools:**
  + **Use the Entity validator in Backstage UI**
  + **Check the backend logs (packages/backend) for entity refresh errors**

#### **d. Authentication/SSO Fails**

* **Symptoms:**
  + **Login loop**
  + **JWT token errors**
* **Fixes:**
  + **Check auth provider configuration in app-config.yaml**
  + **Inspect browser cookies and local storage**

#### **e. Broken Links or 404 Errors**

* **Possible Causes:**
  + **Plugin route misconfiguration**
  + **Frontend not rebuilt after route change**
* **Fixes:**
  + **Run yarn build to ensure route definitions are correctly compiled**

### **3. Debugging Methodology**

#### **Step 1: Isolate the Problem**

* **Is the error from the frontend, backend, plugin, or third-party integration?**
* **Use browser dev tools, terminal logs, and network inspectors.**

#### **Step 2: Check Logs**

* **Frontend: Look in browser console or yarn start output**
* **Backend: Check output from packages/backend when using yarn start-backend**

#### **Step 3: Simplify the Environment**

* **Comment out custom plugins or features**
* **Revert to working app-config.yaml or git snapshot**

#### **Step 4: Use Debug Builds**

* **Add console.log or use node --inspect for backend debugging**
* **Use React DevTools and Redux DevTools for frontend state inspection**

### **4. Tools and Techniques**

| **Tool** | **Use** |
| --- | --- |
| **VSCode Debugger** | **Attach to node process running backend** |
| **React DevTools** | **Inspect component tree and props** |
| **Network Tab** | **Verify outgoing API calls and payloads** |
| **Postman/cURL** | **Test API endpoints independently** |
| **Backstage CLI (@backstage/cli)** | **Use backstage-cli versions:check to check version issues** |
| **Git Bisect** | **Identify faulty commits when issues are introduced gradually** |

### **5. Best Practices**

* **Keep logs verbose in development (LOG\_LEVEL=debug)**
* **Enable source maps for better stack traces**
* **Document custom plugins or config changes**
* **Maintain an error playbook for common internal errors**
* **Regularly validate plugins after upgrades**

### **6. Common Pitfalls and Fixes**

| **Pitfall** | **Fix** |
| --- | --- |
| **Plugin not rendering** | **Check plugin registration in frontend and backend** |
| **Auth loop** | **Verify callback URL and auth provider client ID/secret** |
| **Catalog not updating** | **Validate entity YAMLs and backend processing logs** |
| **Frontend build fails** | **Clear cache (yarn clean, yarn install)** |

### **Conclusion**

**Debugging in Backstage requires a good understanding of its plugin-based architecture, configuration system, and typical integration points. By categorizing issues and using structured tools and practices, developers can efficiently identify and fix common problems in both development and production environments.**

## **Reading Logs and Tracking Plugin Errors in Backstage**

**Proper log management and error tracking are crucial for maintaining the health of your Backstage instance. Since Backstage consists of both a frontend and backend (along with multiple plugins), logs help diagnose runtime issues, track plugin failures, and monitor overall system behavior.**

### **1. Types of Logs in Backstage**

| **Area** | **Log Source** | **Description** |
| --- | --- | --- |
| **Frontend Logs** | **Browser console (DevTools)** | **Errors from React components, plugin UI rendering, API calls** |
| **Backend Logs** | **Terminal output from yarn start-backend or production logs** | **Backend service behavior, plugin route errors, service connections** |
| **Build Logs** | **Output from yarn build or yarn dev** | **Errors during compilation or module resolution** |
| **Deployment Logs** | **Logs from Kubernetes, Docker, or VM environments** | **Runtime and startup logs in production** |
| **3rd-party Service Logs** | **GitHub, LDAP, SSO provider logs** | **Needed when debugging integration issues** |

### **2. Enabling and Controlling Logging**

**Backstage uses winston and pino for logging in the backend and CLI tools. You can control log verbosity using the LOG\_LEVEL environment variable.**

**Common log levels:**

* **error: Critical failures only**
* **warn: Warnings and recoverable issues**
* **info: Normal operation logs**
* **debug: Detailed trace logs (best for development)**

**Example:**

**LOG\_LEVEL=debug yarn start-backend**

### **3. Where to Look for Logs**

| **Context** | **Where to Look** |
| --- | --- |
| **Local Backend** | **Terminal output running yarn start-backend in packages/backend** |
| **Frontend (UI)** | **Chrome DevTools Console (check stack traces and failed requests)** |
| **API Failures** | **Network tab in DevTools (Inspect response codes and payloads)** |
| **Production Logs** | **System logs (e.g., kubectl logs, Docker logs, ELK/CloudWatch)** |

### **4. Common Logging Patterns in Backend**

**You can add custom logging in backend plugins:**

**logger.info('Starting integration with GitHub');**

**logger.error('Failed to connect to GitHub API', error);**

**Use structured logging to attach metadata:**

**logger.debug('Processing entity', { entityRef });**

### **5. Tracking Plugin Errors**

**Backstage plugins can fail due to:**

* **Misconfigured routes or endpoints**
* **Incompatible plugin versions**
* **Missing plugin APIs (especially if backend support is required)**

#### **To Track These Errors:**

* **Frontend: Check if the plugin renders a blank screen or shows a runtime exception.**
  + **Use console.error and component-level error boundaries.**
* **Backend: Look for messages like Plugin X failed to load route or stack traces indicating failed service calls.**

### **6. Monitoring with Log Aggregators**

**In production, Backstage logs can be sent to:**

* **ELK Stack (Elasticsearch + Logstash + Kibana)**
* **Promtail + Loki + Grafana**
* **AWS CloudWatch / GCP Logging**
* **Datadog / Splunk / New Relic**

**Set up structured logging for searchability and alerting (e.g., alerts on repeated plugin failures).**

### **7. Best Practices**

| **Practice** | **Benefit** |
| --- | --- |
| **Use consistent log levels** | **Helps distinguish severity of issues** |
| **Log contextual data** | **Easier to trace root cause using entity names, user IDs, etc.** |
| **Avoid excessive logging in production** | **Reduces clutter and log volume** |
| **Rotate and archive logs** | **Prevents disk space overuse** |
| **Monitor for patterns** | **Repeated failures may indicate deeper architectural issues** |

### **8. Real-World Debugging Example**

**Issue: The TechDocs plugin is not loading for a component.**

**Steps to Track:**

1. **Check browser console for frontend React errors or failed API calls.**
2. **Inspect the Network tab to see if /techdocs/static/docs/... is reachable.**
3. **View backend logs for static file resolution issues or markdown conversion errors.**
4. **Confirm that plugin logs in packages/backend include messages about TechDocs build status.**

### **Conclusion**

**Backstage log analysis is a key part of maintaining your portal. By effectively leveraging logs from both the frontend and backend, and integrating structured logging and log aggregators in production, you can detect, diagnose, and resolve plugin-related issues efficiently. This not only improves system reliability but also enhances the developer experience.**

## **Fixing Configuration or Connection Problems**

**Backstage is highly configurable and integrates with various external systems like source control, CI/CD, identity providers, and artifact stores. While this flexibility is a strength, misconfigurations are among the most common causes of operational issues.**

### **1. Configuration in Backstage**

**The primary configuration file is app-config.yaml, which is hierarchical and environment-sensitive. Configuration can also be split into multiple files like:**

* **app-config.yaml**
* **app-config.local.yaml**
* **app-config.production.yaml**

**You can also use environment variables to override sensitive values.**

### **2. Common Configuration Areas**

| **Area** | **Configuration Section** | **Typical Issues** |
| --- | --- | --- |
| **App Base URL** | **app.baseUrl** | **Incorrect domain or port, affecting SSO callbacks** |
| **Backend URL** | **backend.baseUrl, backend.listen** | **Wrong port binding or missing public URL** |
| **Auth Providers** | **auth.providers** | **Invalid client IDs, secrets, or callback URLs** |
| **Catalog Locations** | **catalog.locations** | **Invalid URLs, unreachable GitHub repos** |
| **Proxy Settings** | **proxy** | **Incorrect target URLs or CORS issues** |
| **Integrations** | **integrations.github/gitlab** | **Wrong tokens or misformatted hostnames** |
| **Database** | **database.connection** | **Wrong host, port, username/password for Postgres** |

### **3. Identifying Configuration Issues**

**Symptoms of configuration problems often include:**

* **Plugins not loading or showing 404 errors**
* **Failed API calls (visible in frontend DevTools)**
* **Authentication flows failing or redirecting incorrectly**
* **Catalog not showing entities or TechDocs not rendering**
* **Console errors like Missing config key, Cannot resolve URL, or connection timeouts**

**Tools to Use:**

* **Backend logs (yarn start-backend)**
* **Frontend console/network tab**
* **Postman/cURL for testing API endpoints**
* **YAML validators (e.g.,** [**yamlint**](https://yamllint.com/)**)**

### **4. Connection Problems**

**These often occur between:**

* **Backend ↔ External systems (e.g., GitHub, LDAP, Kubernetes)**
* **Backend ↔ Database**
* **Frontend ↔ Backend**

**Common Root Causes:**

* **DNS resolution failures**
* **Incorrect service ports or timeouts**
* **Proxy configuration issues**
* **Firewalls or IP restrictions**
* **Expired credentials (e.g., GitHub tokens, SSO certs)**

### **5. Debugging Steps**

#### **a. Validate Configuration**

* **Run: yarn backstage-cli config:check**
* **Manually inspect keys in app-config.yaml and .env**

#### **b. Use Diagnostic Logs**

* **Look for errors like:**
  + **Missing configuration for key**
  + **Unable to reach external service**
  + **Unauthorized or 401 errors**

#### **c. Use Test Tools**

* **Test APIs with curl/Postman**
* **Use ping, nslookup, or telnet to verify connectivity**
* **Try backend-only tools like psql or curl from inside container/VM**

#### **d. Use Fallback Values Temporarily**

* **Set defaults in configuration to verify that issues are external (e.g., switch to a known-working GitHub repo)**

### **6. Best Practices**

| **Practice** | **Benefit** |
| --- | --- |
| **Split configs by environment (local, prod)** | **Isolates risk and simplifies debugging** |
| **Use .env files or secrets manager for sensitive data** | **Keeps credentials secure and separate** |
| **Validate configs on deployment** | **Prevents runtime surprises** |
| **Document all keys used and expected values** | **Makes onboarding and troubleshooting easier** |
| **Use version control for config files** | **Enables rollback of misconfigurations** |

### **7. Real-World Example**

**Problem: GitHub plugin can’t fetch entities.**

**Investigation:**

* **Backend logs: Failed to fetch from GitHub - Unauthorized**
* **Config: Missing or expired GitHub token in app-config.yaml**
* **Fix: Regenerate token, update integrations.github.token, restart backend**

### **Conclusion**

**Most operational issues in Backstage stem from incorrect or missing configuration. By understanding how configuration is structured and validated, and by systematically testing connections, you can quickly isolate and fix problems. This helps keep your internal developer portal running smoothly and securely.**

## **Understanding How to Test Changes Before Production Rollout**

**Testing changes before deploying them to a production Backstage instance is essential to ensure reliability, prevent downtime, and avoid disruptions to developer workflows. Backstage provides mechanisms and recommended workflows for validating updates to core components, custom plugins, configuration, and integrations.**

### **1. Why Pre-Production Testing is Critical**

**Changes in Backstage may involve:**

* **Upgrading Backstage core or plugins**
* **Introducing new plugins or APIs**
* **Modifying app-config.yaml**
* **Customizing frontend components**
* **Updating authentication or authorization systems**

**Without testing:**

* **Unexpected regressions may break features**
* **Misconfigured auth or proxy settings can lock users out**
* **Plugin changes might fail silently or throw runtime errors**

### **2. Common Testing Environments**

| **Environment** | **Purpose** | **Notes** |
| --- | --- | --- |
| **Local Dev Environment** | **Fast feedback, plugin development** | **Use yarn dev, yarn start-backend** |
| **Staging/Test Environment** | **Mimics production setup** | **Should match production configs, plugins, auth systems** |
| **CI/CD Previews** | **On PRs, deploy preview environments** | **Use GitHub Actions or GitLab CI for ephemeral deployments** |

### **3. Strategies to Test Changes**

#### **a. Local Development Testing**

* **Use yarn dev to test the frontend (packages/app)**
* **Use yarn start-backend to run the backend**
* **Use mock data, sample catalog entities, and TechDocs locally**
* **Verify log output, error messages, and runtime behavior**

#### **b. Feature Branch Testing**

* **Create branches for each feature or upgrade**
* **Run test suites (yarn test, jest, linting)**
* **Deploy the feature branch to a test environment**
* **Let team members validate changes via a shared URL**

#### **c. End-to-End Testing**

* **Use tools like Cypress or Playwright to simulate user flows**
* **Test critical paths like:**
  + **Browsing the catalog**
  + **Viewing TechDocs**
  + **Authenticating via SSO**
  + **Using software templates**

#### **d. Configuration Validation**

**Use:  
  
 yarn backstage-cli config:check**

* **Ensures all config keys are recognized and resolvable**

#### **e. CI/CD Testing Pipelines**

* **Run:**
  + **Unit tests**
  + **Linting**
  + **Build verification**

**Example GitHub Action:  
  
 - name: Run tests**

**run: yarn test**

**- name: Lint code**

**run: yarn lint**

**- name: Build frontend**

**run: yarn tsc && yarn build**

### **4. Testing Plugin Updates and Compatibility**

**Backstage plugins are versioned independently, so:**

* **Check compatibility matrix if upgrading core**
* **Manually test plugin pages**
* **Review** [**Backstage changelogs**](https://backstage.io/docs/releases/changelog)

**For in-house plugins:**

* **Validate API schema changes**
* **Run isolated plugin builds**

### **5. Testing Authentication and Permission Changes**

**For changes in auth or permission policies:**

* **Use a test SSO provider setup (e.g., Okta dev org)**
* **Validate role-based access control for different user roles**
* **Ensure login redirects, logout flows, and token refreshes work**

### **6. Testing Deployment/Infrastructure Changes**

**If you're updating Docker images, Kubernetes manifests, or Helm charts:**

* **Use a test namespace**
* **Deploy the new version using:**
  + **Docker Compose (for local prod-like setup)**
  + **Helm (for Kubernetes environments)**
* **Monitor logs, readiness/liveness probes, and network policies**

### **7. Rollback Planning**

**Even with testing, failures can occur. Be ready to:**

* **Revert code via Git**
* **Rollback Helm charts**
* **Use blue/green or canary deployments in production**

### **8. Best Practices**

| **Practice** | **Benefit** |
| --- | --- |
| **Isolate environments (dev, test, prod)** | **Prevents accidental disruption** |
| **Automate tests in CI/CD** | **Reduces risk of regressions** |
| **Involve users in testing preview builds** | **Catches UX issues early** |
| **Document changes in a changelog or release notes** | **Helps operations and dev teams track updates** |
| **Use version control for config files** | **Enables rollback and auditability** |

### **Conclusion**

**A well-defined testing workflow ensures stability and confidence in deploying changes to Backstage. Whether you're updating plugins, changing configuration, or enhancing auth systems, thorough pre-production testing avoids costly downtime and preserves the trust of your developers.**

**.Upgrading Plugins and Core Versions**

**Keeping Backstage core and plugins up to date is important for accessing new features, security patches, performance improvements, and bug fixes. However, upgrades require careful planning to avoid breaking your internal developer portal.**

### **1. Understanding Backstage Core and Plugins**

* **Backstage Core: The framework and common infrastructure (frontend app, backend, APIs).**
* **Plugins: Individual features or integrations (e.g., TechDocs, GitHub, Kubernetes plugins).**
* **Both core and plugins have their own versioning, and plugins depend on core versions.**

### **2. When to Upgrade**

* **New feature releases that improve usability or integration.**
* **Security patches or critical bug fixes.**
* **Compatibility with dependencies or external services.**
* **Keeping up with community improvements.**

### **3. Upgrade Workflow**

#### **a. Review Release Notes**

* **Check Backstage** [**release notes**](https://backstage.io/docs/releases) **for core and plugin updates.**
* **Identify breaking changes or migration steps.**

#### **b. Check Compatibility**

* **Verify plugin versions are compatible with the targeted core version.**
* **Use package.json and the Backstage docs for compatibility matrices.**

#### **c. Prepare Your Codebase**

* **Pin your current version to prevent accidental upgrades.**
* **Backup configurations and customizations.**

#### **d. Update Dependencies**

**Run:  
  
 yarn backstage-cli versions:check**

* **to see outdated dependencies.**
* **Update core and plugin versions in package.json.**

**Run:  
  
 yarn install**

**yarn tsc**

**yarn build**

#### **e. Test Locally and in Staging**

* **Run all tests.**
* **Test your app and plugins thoroughly for regressions.**

#### **f. Deploy Gradually**

* **Use canary or blue/green deployments if possible.**
* **Monitor logs and user feedback closely.**

### **4. Handling Breaking Changes**

* **Read upgrade guides and migration notes carefully.**
* **Adjust your app-config.yaml or plugin code as needed.**
* **Some APIs or plugin interfaces may change.**
* **You might need to refactor custom plugins or integrations.**

### **5. Automating Upgrades**

* **Use tools like Renovate or Dependabot to track dependency updates.**
* **Automate test and build processes via CI/CD pipelines.**

### **6. Rollback Plan**

* **Keep your previous working versions pinned.**
* **Rollback using version control if the upgrade introduces issues.**
* **Maintain backups of configuration and databases.**

### **7. Real-World Example**

**Upgrading from Backstage version 1.0 to 1.3 might require:**

* **Updating all plugins to compatible versions.**
* **Adjusting new auth configuration settings.**
* **Testing OAuth provider integration as it may have API changes.**

### **Conclusion**

**Upgrading Backstage core and plugins is a routine yet critical task. Following best practices for version management, testing, and gradual rollout ensures your internal developer portal remains stable, secure, and up to date with new capabilities.**

## **Plugin Compatibility and Migration Best Practices**

**As Backstage evolves, plugins may need to be updated, migrated, or replaced to maintain compatibility with the core platform and ensure seamless developer experience. Managing plugin compatibility and migrations properly is crucial for long-term maintainability.**

### **1. Understanding Plugin Compatibility**

* **Plugins depend on specific versions of Backstage core and sometimes on other plugins.**
* **Incompatible plugins can cause runtime errors, build failures, or feature regressions.**
* **Version mismatches between core and plugins are a common source of issues.**

### **2. Checking Compatibility**

* **Always consult the official Backstage** [**release notes**](https://backstage.io/docs/releases) **and plugin documentation.**
* **Use yarn backstage-cli versions:check to get a report of versions used and their compatibility.**
* **Pay attention to major version changes, which often include breaking API changes.**

### **3. Migration Scenarios**

* **API changes: Core or plugins may change API signatures, requiring plugin code updates.**
* **Deprecations: Features or configurations may be deprecated and replaced.**
* **Plugin replacements: Some plugins may be deprecated and replaced with new alternatives.**
* **Configuration changes: Updates in app-config.yaml structure may require config migration.**

### **4. Migration Best Practices**

#### **a. Plan and Document**

* **Review release notes and migration guides.**
* **Document any custom changes required.**
* **Communicate planned changes with your developer teams.**

#### **b. Use Version Control**

* **Commit changes before migration.**
* **Create feature branches for migration work.**
* **Enable easy rollback if issues arise.**

#### **c. Incremental Updates**

* **Update plugins incrementally rather than all at once to isolate issues.**
* **Test each plugin update separately.**

#### **d. Automate Testing**

* **Leverage automated unit and integration tests to validate plugin functionality.**
* **Use end-to-end testing to verify plugin behavior in workflows.**

#### **e. Backup Configurations**

* **Backup configuration files before applying changes.**
* **Use configuration management to track and validate changes.**

#### **f. Monitor Post-Migration**

* **Monitor application logs for errors or warnings.**
* **Gather user feedback to detect subtle regressions.**

### **5. Handling Deprecated Plugins**

* **Identify deprecated plugins through release notes and official documentation.**
* **Evaluate new recommended alternatives.**
* **Plan migration timelines to avoid using unsupported plugins.**

### **6. Example Migration Workflow**

1. **Identify outdated or incompatible plugins.**
2. **Review migration guides for new plugin versions.**
3. **Update plugin dependencies in package.json.**
4. **Adjust plugin usage in code and configurations.**
5. **Test changes locally and in staging.**
6. **Deploy updated plugins to production.**

### **7. Summary**

**Properly managing plugin compatibility and migrations avoids unexpected downtime and ensures your Backstage instance evolves smoothly with the community and core platform. Planning, testing, and incremental changes are key strategies for successful plugin management.**

## **Debugging Frontend and Backend Performance Issues**

**Maintaining good performance in both the frontend and backend of Backstage is essential to ensure a responsive and reliable internal developer portal. Performance issues can degrade user experience and slow down workflows.**

### **1. Performance in Backstage**

* **Frontend performance impacts user experience, page load times, and interactivity.**
* **Backend performance affects API response times, plugin processing, and data retrieval.**
* **Both layers must be monitored and optimized.**

### **2. Common Causes of Performance Issues**

| **Layer** | **Causes** |
| --- | --- |
| **Frontend** | **Large bundle sizes, inefficient React renders, excessive network requests, slow API responses** |
| **Backend** | **Slow database queries, excessive plugin workload, blocking operations, network latency, resource constraints** |

### **3. Debugging Frontend Performance**

#### **a. Browser DevTools**

* **Use Chrome DevTools Performance tab to record page load and interaction profiles.**
* **Analyze:**
  + **Time to First Paint (TTFP)**
  + **Time to Interactive (TTI)**
  + **Long tasks blocking the main thread**
  + **Network waterfall for slow API calls or large assets**

#### **b. React Profiler**

* **Use React DevTools Profiler to identify slow or unnecessary renders.**
* **Optimize components causing frequent updates.**

#### **c. Bundle Analysis**

* **Use tools like webpack-bundle-analyzer to identify large dependencies.**
* **Optimize by lazy loading plugins, removing unused code.**

#### **d. Caching**

* **Enable client-side caching of API responses where appropriate.**
* **Use service workers or HTTP cache headers.**

### **4. Debugging Backend Performance**

#### **a. Logging and Metrics**

* **Review backend logs for slow requests or errors.**
* **Use metrics exporters (e.g., Prometheus) to monitor request latency and error rates.**

#### **b. Profiling**

* **Profile Node.js backend using tools like clinic.js or node --inspect.**
* **Identify blocking or CPU-intensive operations.**

#### **c. Database Performance**

* **Check query execution times.**
* **Use indexes and optimize queries.**
* **Monitor database resource utilization.**

#### **d. Plugin Load**

* **Some plugins can introduce significant overhead.**
* **Monitor plugin-specific metrics to isolate bottlenecks.**

### **5. Optimization Strategies**

| **Area** | **Optimization Tips** |
| --- | --- |
| **Frontend** | **Code splitting, memoization, lazy loading, reduce DOM size** |
| **Backend** | **Asynchronous programming, caching, optimizing database access, scaling horizontally** |
| **Network** | **Use CDN, compress responses, reduce payload sizes** |

### **6. Monitoring Tools**

* **Integrate monitoring and alerting for latency and error rates.**
* **Use Grafana dashboards to visualize performance trends.**
* **Use tracing tools (e.g., Jaeger, OpenTelemetry) to trace requests across frontend/backend.**

### **7. Summary**

**Systematic analysis of performance issues in both frontend and backend layers helps maintain a fast and efficient Backstage portal. Using profiling tools, monitoring, and best practices ensures your developers experience a smooth workflow.**