Backstage 101

**1. High-Level Architecture Overview**

Backstage is a platform for building developer portals, and its architecture is divided into two main parts:

* **Frontend**: The user interface (UI) built with React.
* **Backend**: The server-side logic built with Node.js and Express.

These two components communicate via APIs and work together to provide a unified experience. Plugins extend the functionality of Backstage, and TechDocs provide documentation capabilities. A database (e.g., PostgreSQL) is used for persistent storage.

Reference - <https://backstage.io/docs/overview/architecture-overview/>

**2. Frontend Workflow**

The frontend is the user-facing part of Backstage, built using React. Here's how it works:

1. **User Interaction**:
   * Users interact with the Backstage UI (e.g., browsing catalog entities, viewing TechDocs, or managing infrastructure).
   * The UI is composed of reusable React components.
2. **API Calls**:
   * The frontend makes HTTP requests to the backend APIs to fetch or update data (e.g., catalog entities, TechDocs, or plugin-specific data).
   * These requests are handled by the backend.
3. **Plugin Integration**:
   * Plugins are integrated into the frontend as React components.
   * Each plugin can have its own UI and API endpoints.
4. **TechDocs Rendering**:
   * TechDocs are rendered in the frontend using Markdown or MkDocs.
   * The frontend fetches documentation files from the backend or an external storage (e.g., AWS S3, Google Cloud Storage).

**3. Backend Workflow**

The backend is responsible for handling business logic, serving APIs, and managing data. Here's how it works:

1. **API Endpoints**:
   * The backend exposes RESTful APIs for the frontend to interact with.
   * These APIs handle requests related to the catalog, TechDocs, plugins, and other services.
2. **Catalog Management**:
   * The backend manages the software catalog, which stores metadata about services, components, and resources.
   * The catalog is stored in a database (e.g., PostgreSQL) or fetched from external sources (e.g., GitHub, GitLab).
3. **Plugin Backend**:
   * Plugins can extend the backend by adding new API endpoints or integrating with external services.
   * Each plugin can have its own backend logic.
4. **TechDocs Generation**:
   * The backend generates TechDocs by pulling documentation files from source code repositories.
   * It uses MkDocs or Markdown to render the documentation and stores it in a storage system (e.g., S3, GCS).
5. **Database Interaction**:
   * The backend interacts with a database to store and retrieve data (e.g., catalog entities, user preferences).
   * PostgreSQL is commonly used for this purpose.

**4. How Frontend and Backend Work Together**

1. **API Communication**:
   * The frontend sends HTTP requests to the backend APIs to fetch or update data.
   * The backend processes these requests, interacts with the database or external services, and returns the response to the frontend.
2. **Plugin Integration**:
   * Plugins can have both frontend and backend components.
   * The frontend plugin UI communicates with the backend plugin API to fetch or update data.
3. **TechDocs Workflow**:
   * The frontend requests documentation from the backend.
   * The backend fetches or generates the documentation and serves it to the frontend.
4. **Catalog Management**:
   * The frontend displays catalog entities (e.g., services, components) fetched from the backend.
   * The backend manages the catalog by storing metadata in the database or fetching it from external sources.

**5. Plugin Architecture**

Plugins are the core extensibility mechanism in Backstage. Here's how they work:

1. **Frontend Plugin**:
   * A React component that provides a UI for the plugin.
   * Communicates with the backend plugin API to fetch or update data.
2. **Backend Plugin**:
   * Adds new API endpoints or integrates with external services.
   * Handles business logic and data processing.
3. **Plugin Integration**:
   * Plugins are registered in the Backstage app.
   * They can extend the catalog, add new pages, or integrate with external tools (e.g., CI/CD systems).

**6. TechDocs Workflow**

TechDocs is a documentation system in Backstage. Here's how it works:

1. **Documentation Source**:
   * Documentation files (Markdown, MkDocs) are stored in the source code repository.
2. **Documentation Generation**:
   * The backend pulls the documentation files and generates HTML using MkDocs.
   * The generated documentation is stored in a storage system (e.g., S3, GCS).
3. **Documentation Rendering**:
   * The frontend fetches the generated documentation from the backend or storage system and renders it in the UI.

**7. Database Workflow**

The database is used for persistent storage in Backstage. Here's how it works:

1. **Catalog Storage**:
   * Metadata about catalog entities (e.g., services, components) is stored in the database.
2. **User Preferences**:
   * User-specific settings and preferences are stored in the database.
3. **Plugin Data**:
   * Plugins can use the database to store their own data.

**8. End-to-End Workflow**

1. **User Interaction**:
   * A user interacts with the Backstage UI (e.g., browsing the catalog or viewing TechDocs).
2. **API Request**:
   * The frontend sends an API request to the backend (e.g., to fetch catalog entities or documentation).
3. **Backend Processing**:
   * The backend processes the request, interacts with the database or external services, and returns the response.
4. **Frontend Rendering**:
   * The frontend renders the response in the UI (e.g., displaying catalog entities or documentation).
5. **Plugin Integration**:
   * Plugins extend the functionality of Backstage by adding new UI components and backend logic.
6. **TechDocs Workflow**:
   * The backend generates and serves documentation, which is rendered in the frontend.

**9. External Integrations**

* **Components**:
  + **GitHub / GitLab**: Source code repositories for catalog entities and documentation.
  + **CI/CD Tools**: Jenkins, GitHub Actions, etc.
  + **Kubernetes**: For infrastructure management.
  + **Authentication Providers**: GitHub OAuth, Google Auth, etc.
* **Connections**:
  + The Catalog API fetches metadata from GitHub/GitLab.
  + CI/CD plugins interact with Jenkins/GitHub Actions.
  + Kubernetes plugins interact with Kubernetes clusters.

**Backstage Architecture Diagram**

**Example Diagram Layout**

+-------------------+ +-------------------+ +-------------------+

| Frontend Layer | | Backend Layer | | Database Layer |

|-------------------| |-------------------| |-------------------|

| React App |<----->| Express Server |<----->| PostgreSQL |

| Plugin UIs | | Catalog API | | |

| Material-UI | | TechDocs API | | |

| API Client | | Plugin APIs | | |

+-------------------+ +-------------------+ +-------------------+

↓ ↓ ↓

+-------------------+ +-------------------+ +-------------------+

| Storage Layer | | Plugin Layer | | Integration APIs |

|-------------------| |-------------------| |-------------------|

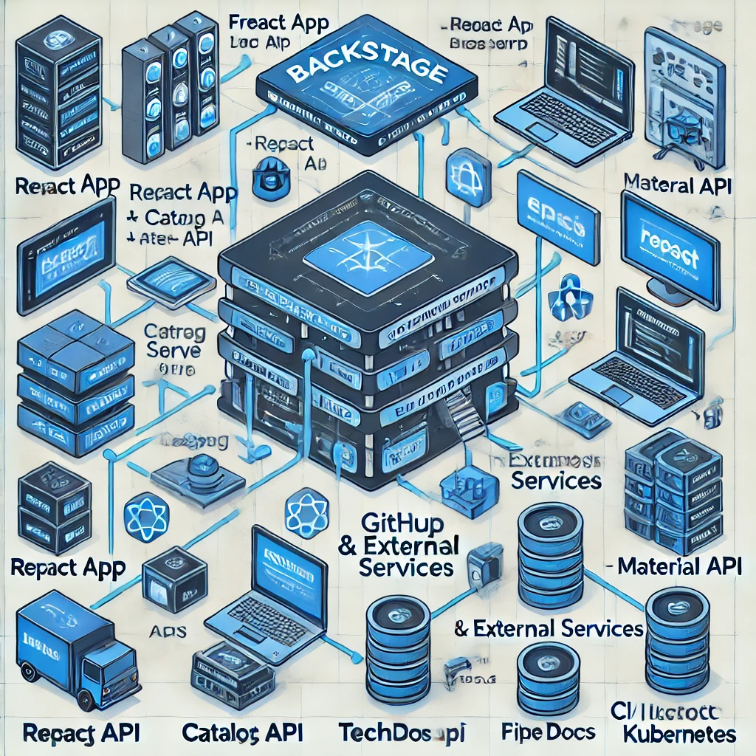
| AWS S3 / GCS | | Frontend Plugins |<----->| GitHub / GitLab |

| File System | | Backend Plugins | | CI/CD Tools |

| | | External

[Integration | | Kubernetes |

+-------------------+ +-------------------+ +-------------------+



**1. How Backstage is Different from a Typical React App**

Backstage is not just a React app; it’s a **platform** built on top of React and Node.js. Here’s how it differs from a typical React app:

**a. Platform vs. Application**

* **Typical React App**:
  + A standalone application with a single purpose (e.g., a dashboard, e-commerce site).
  + Usually has a single backend and frontend.
  + Limited extensibility; adding new features requires modifying the core codebase.
* **Backstage**:
  + A **platform** for building developer portals.
  + Designed to be **extensible** through plugins.
  + Provides a **unified interface** for managing software ecosystems (e.g., services, documentation, CI/CD).
  + Comes with pre-built features like a **software catalog**, **TechDocs**, and **scaffolding**.

**b. Built-in Features**

* **Software Catalog**:
  + Backstage includes a built-in catalog to manage metadata about services, components, and resources.
  + This is not something a typical React app would have out-of-the-box.
* **TechDocs**:
  + Backstage integrates documentation directly into the platform, allowing you to view and manage docs alongside your services.
* **Plugins**:
  + Backstage is designed to be extended with plugins, which can add new features or integrate with external tools (e.g., Jenkins, Kubernetes).

**c. Backend Integration**

* **Typical React App**:
  + The backend is usually custom-built for the app’s specific needs.
  + Communication between frontend and backend is limited to the app’s requirements.
* **Backstage**:
  + The backend is part of the platform and provides APIs for the frontend and plugins.
  + The backend also integrates with external systems (e.g., GitHub, Jenkins) to fetch data and perform actions.

**d. Developer Experience**

* **Typical React App**:
  + Focused on end-user experience.
  + Developers need to build everything from scratch.
* **Backstage**:
  + Focused on **developer experience**.
  + Provides tools for developers to manage their software ecosystem (e.g., service discovery, documentation, CI/CD).

**2. Ways to Set Up a Backstage App Locally**

There are several ways to set up a Backstage app on your local environment, depending on your needs and familiarity with the platform:

**a. Using the Backstage CLI (Recommended for Beginners)**

1. Install the Backstage CLI:

npm install -g @backstage/cli

1. Create a new Backstage app:

npx @backstage/create-app

1. Follow the prompts to set up your app (e.g., choose a name, configure authentication).
2. Start the app:

yarn dev

This will start both the frontend and backend servers.

**b. Manual Setup (Advanced)**

1. Clone the Backstage repository:

git clone https://github.com/backstage/backstage.git

1. Install dependencies:

yarn install

1. Configure the app:
   * Modify app-config.yaml to set up authentication, database, and integrations.
   * Set up the database (e.g., PostgreSQL) and update the configuration.
2. Start the app:

yarn dev

**c. Using Docker (For Containerized Environments)**

1. Create a Backstage app using the CLI (as shown above).
2. Build the Docker image:

docker build -t backstage-app .

1. Run the Docker container:

docker run -p 3000:3000 backstage-app

**d. Using a Backstage Template**

* Backstage provides templates for common use cases (e.g., microservices, monorepos).
* You can use these templates to bootstrap your app:

npx @backstage/create-app --template microservice

**3. How Backstage Components are Different from Plugins**

**a. Backstage Components**

* **Definition**:
  + Components are reusable UI elements built with React.
  + They are part of the core Backstage app and provide basic functionality (e.g., catalog table, search bar).
* **Usage**:
  + Components are used to build the core features of Backstage (e.g., the software catalog, TechDocs).
  + They are not standalone features but rather building blocks for the app.

**b. Plugins**

* **Definition**:
  + Plugins are extensions that add new features or integrate with external tools.
  + They can include both frontend (React components) and backend (APIs) logic.
* **Usage**:
  + Plugins are used to extend the functionality of Backstage (e.g., adding a CI/CD dashboard, integrating with Kubernetes).
  + They are standalone features that can be added or removed without modifying the core app.

**c. Key Differences**

| **Aspect** | **Components** | **Plugins** |
| --- | --- | --- |
| **Purpose** | Building blocks for the core app. | Extending the app with new features. |
| **Scope** | Limited to the core app. | Can include frontend and backend logic. |
| **Extensibility** | Not extensible by default. | Designed to be extensible. |
| **Examples** | Catalog table, search bar. | CI/CD plugin, Kubernetes plugin. |

**4. End-to-End Usage of Components and Plugins**

**a. Components**

1. **Catalog Component**:
   * Displays a table of services, components, and resources.
   * Fetches data from the backend catalog API.
   * Used in the core app to provide service discovery.
2. **TechDocs Component**:
   * Renders documentation in the UI.
   * Fetches documentation from the backend or storage system.
   * Used in the core app to provide documentation access.

**b. Plugins**

1. **CI/CD Plugin**:
   * Frontend: Displays CI/CD pipelines and build statuses.
   * Backend: Integrates with Jenkins/GitHub Actions to fetch pipeline data.
   * Usage: Developers use the plugin to monitor and manage CI/CD pipelines.
2. **Kubernetes Plugin**:
   * Frontend: Displays Kubernetes clusters, pods, and deployments.
   * Backend: Integrates with Kubernetes APIs to fetch cluster data.
   * Usage: Developers use the plugin to manage Kubernetes resources.

**c. End-to-End Workflow**

1. **User Interaction**:
   * A user interacts with the Backstage UI (e.g., views the catalog or checks CI/CD status).
2. **API Calls**:
   * The frontend makes API calls to the backend (e.g., to fetch catalog data or pipeline status).
3. **Backend Processing**:
   * The backend processes the request, interacts with external systems (e.g., GitHub, Jenkins), and returns the response.
4. **UI Rendering**:
   * The frontend renders the response using components or plugin UIs.
5. **Plugin Integration**:
   * Plugins extend the functionality by adding new UI components and backend APIs.

The two primary approaches to setting up a Backstage app locally—**using the Backstage CLI** (Option A) and **manual setup** (Option B)—serve different purposes and are suited for different scenarios. Let’s break down the differences, use cases, and when to use each approach.

**Option A: Using the Backstage CLI**

**What It Does**

* The Backstage CLI (@backstage/cli) provides a streamlined way to create and manage a Backstage app.
* It automates the setup process, including:
  + Generating the app structure.
  + Installing dependencies.
  + Configuring basic settings (e.g., authentication, database).

**When to Use This Approach**

1. **For Beginners**:
   * If you’re new to Backstage, the CLI is the easiest way to get started.
   * It abstracts away the complexity of manual configuration.
2. **Quick Prototyping**:
   * If you want to quickly spin up a Backstage app to test or prototype, the CLI is ideal.
   * It sets up a fully functional app in minutes.
3. **Standard Use Cases**:
   * If your use case aligns with the default Backstage features (e.g., software catalog, TechDocs), the CLI is sufficient.
   * It provides a solid foundation for most developer portal needs.
4. **Limited Customization Needed**:
   * If you don’t need deep customization of the app structure or build process, the CLI is the way to go.

**Pros**

* **Ease of Use**: Minimal setup required.
* **Speed**: Get started in minutes.
* **Best Practices**: Follows Backstage’s recommended structure and configuration.

**Cons**

* **Limited Flexibility**: Less control over the app structure and build process.
* **Generic Setup**: May include features you don’t need.

**Option B: Manual Setup**

**What It Does**

* Manual setup involves cloning the Backstage repository and configuring everything yourself.
* You have full control over the app structure, dependencies, and configuration.

**When to Use This Approach**

1. **Advanced Users**:
   * If you’re familiar with Backstage and want full control over the setup, manual configuration is better.
   * You can customize the app to your exact needs.
2. **Custom Use Cases**:
   * If your use case requires significant customization (e.g., custom plugins, unique build process), manual setup is necessary.
   * You can modify the app structure, add/remove features, and tweak configurations.
3. **Contributing to Backstage**:
   * If you’re contributing to the Backstage open-source project, you’ll need to clone the repository and set it up manually.
4. **Complex Integrations**:
   * If you need to integrate with non-standard tools or services, manual setup gives you the flexibility to do so.

Configure the app:

* + Modify app-config.yaml for authentication, database, and integrations.
  + Set up the database (e.g., PostgreSQL) and update the configuration.

**Pros**

* **Full Control**: Customize every aspect of the app.
* **Flexibility**: Add/remove features and integrations as needed.
* **Learning Opportunity**: Gain a deeper understanding of Backstage’s internals.

**Cons**

* **Complexity**: Requires more time and effort to set up.
* **Error-Prone**: Manual configuration can lead to mistakes if not done carefully.
* **Maintenance**: You’re responsible for keeping the app updated and compatible with new Backstage releases.

**Comparison: CLI vs. Manual Setup**

| **Aspect** | **CLI (Option A)** | **Manual Setup (Option B)** |
| --- | --- | --- |
| **Ease of Use** | Easy for beginners. | Requires technical expertise. |
| **Speed** | Quick setup (minutes). | Slower setup (hours or more). |
| **Customization** | Limited to CLI options. | Full control over app structure/config. |
| **Use Case** | Standard use cases, prototyping. | Custom use cases, advanced integrations. |
| **Learning Curve** | Low. | High. |
| **Maintenance** | Handled by CLI (e.g., updates). | Manual updates and maintenance. |

**When to Use Which Approach**

**Use the CLI (Option A) If:**

* You’re new to Backstage.
* You want to quickly prototype or test Backstage.
* Your use case aligns with the default features (e.g., catalog, TechDocs).
* You don’t need deep customization.

**Use Manual Setup (Option B) If:**

* You’re an advanced user or developer.
* You need full control over the app structure and configuration.
* Your use case requires custom plugins, integrations, or build processes.
* You’re contributing to the Backstage open-source project.

**Hybrid Approach**

You can also start with the CLI and then manually customize the app as needed. For example:

1. Use the CLI to generate the app.
2. Manually modify the app structure, add custom plugins, or tweak configurations.

This approach combines the ease of the CLI with the flexibility of manual setup.

**1. Definitions and When to Use Each**

**a. Plugin**

* **What It Is**:
  + A plugin is an extension that adds new functionality to Backstage.
  + It can include both **frontend (React components)** and **backend (APIs)** logic.
  + Plugins are used to integrate with external tools (e.g., Jenkins, Kubernetes) or add new features (e.g., CI/CD dashboard).
* **When to Use**:
  + When you need to extend Backstage with new features or integrations.
  + Example: Adding a plugin to display Kubernetes cluster information.

**b. Component**

* **What It Is**:
  + A component is a reusable UI element built with React.
  + Components are used to build the core features of Backstage (e.g., catalog table, search bar).
  + They are not standalone features but rather building blocks for the app.
* **When to Use**:
  + When you need to create or customize UI elements in Backstage.
  + Example: Creating a custom card component to display service metadata.

**c. API**

* **What It Is**:
  + An API is a backend service that handles business logic and data processing.
  + APIs are used to fetch or update data, interact with external systems, or perform computations.
  + In Backstage, APIs are typically built using Node.js and Express.
* **When to Use**:
  + When you need to handle backend logic or integrate with external systems.
  + Example: Creating an API to fetch data from a CI/CD tool like Jenkins.

**d. TechDocs**

* **What It Is**:
  + TechDocs is a documentation system in Backstage.
  + It allows you to write documentation in Markdown or MkDocs and render it in Backstage.
  + TechDocs are stored in source code repositories and generated by the Backstage backend.
* **When to Use**:
  + When you need to provide documentation for your services or components.
  + Example: Writing documentation for a microservice and rendering it in Backstage.

**2. Hello World Examples**

**a. Plugin Example**

**Goal**: Create a simple plugin that displays "Hello, Plugin!" in the Backstage UI.

1. **Create the Plugin**:
   * Use the Backstage CLI to create a new plugin:

npx @backstage/create-plugin

* + Name the plugin hello-world-plugin.

1. **Add Frontend Code**:
   * Open the plugin’s frontend file (src/components/HelloWorldPluginPage.tsx) and add:

import React from 'react';

export const HelloWorldPluginPage = () => (

<div>

<h1>Hello, Plugin!</h1>

</div>

);

1. **Register the Plugin**:
   * Add the plugin to the app’s packages/app/src/App.tsx:

import { HelloWorldPluginPage } from '@internal/plugin-hello-world-plugin';

const routes = (

<FlatRoutes>

<Route path="/hello-world" element={<HelloWorldPluginPage />} />

</FlatRoutes>

);

1. **Run the App**:
   * Start the app:

yarn dev

* + Navigate to http://localhost:3000/hello-world to see "Hello, Plugin!".

**b. Component Example**

**Goal**: Create a reusable React component that displays "Hello, Component!".

1. **Create the Component**:
   * Create a new file HelloWorldComponent.tsx in packages/app/src/components:

import React from 'react';

export const HelloWorldComponent = () => (

<div>

<h2>Hello, Component!</h2>

</div>

);

1. **Use the Component**:
   * Use the component in a page (e.g., HomePage.tsx):

import { HelloWorldComponent } from './components/HelloWorldComponent';

const HomePage = () => (

<div>

<h1>Welcome to Backstage!</h1>

<HelloWorldComponent />

</div>

);

1. **Run the App**:
   * Start the app and navigate to the home page to see "Hello, Component!".

**c. API Example**

**Goal**: Create a simple API that returns "Hello, API!".

1. **Create the API**:
   * Create a new file helloWorldApi.ts in packages/backend/src/apis:

import express from 'express';

const router = express.Router();

router.get('/hello', (req, res) => {

res.send('Hello, API!');

});

export default router;

1. **Register the API**:
   * Add the API to packages/backend/src/index.ts:

import helloWorldApi from './apis/helloWorldApi';

const apiRouter = Router();

apiRouter.use('/hello-world', helloWorldApi);

1. **Test the API**:
   * Start the backend:

yarn dev

* + Navigate to http://localhost:7007/api/hello-world/hello to see "Hello, API!".

**d. TechDocs Example**

**Goal**: Add a simple Markdown file and render it in Backstage.

1. **Create the Documentation**:
   * Add a docs folder to your service’s repository with a index.md file:

# Hello, TechDocs!

This is a sample documentation file.

1. **Configure TechDocs**:
   * Update app-config.yaml to enable TechDocs:

techdocs:

builder: 'local'

generator:

runIn: 'docker'

1. **View the Documentation**:
   * Start the app and navigate to the service’s TechDocs page to see the rendered documentation.

**3. End-to-End Example**

1. **Create the Plugin**:
   * Follow the **Plugin Example** to create the hello-world-plugin.
2. **Add the Component**:
   * Follow the **Component Example** to create the HelloWorldComponent.
3. **Create the API**:
   * Follow the **API Example** to create the helloWorldApi.
4. **Add TechDocs**:
   * Follow the **TechDocs Example** to add documentation.
5. **Integrate Everything**:
   * Use the component in the plugin’s page.
   * Call the API from the plugin’s frontend.
   * Link to the TechDocs page from the plugin.
6. **Run the App**:
   * Start the app and navigate to the plugin’s page to see:
     + "Hello, Plugin!"
     + "Hello, Component!"
     + Data from the API.
     + A link to the TechDocs page.

Scenario -

1. **Displays a "Hello, Plugin!" message** (Plugin).
2. **Uses a reusable "Hello, Component!" component** (Component).
3. **Fetches data from a backend API** (API).
4. **Links to a TechDocs page** (TechDocs).

**End-to-End Example**

**1. Setup the Backstage App**

1. **Create a Backstage App**:

npx @backstage/create-app

Name the app my-backstage-app.

1. **Start the App**:

cd my-backstage-app

yarn dev

This will start both the frontend (localhost:3000) and backend (localhost:7007).

**2. Create a Plugin**

**Goal**: Create a plugin that displays "Hello, Plugin!" and uses a reusable component.

1. **Generate the Plugin**:

npx @backstage/create-plugin

Name the plugin hello-world-plugin.

1. **Add the Plugin Code**:
   * Open packages/plugin-hello-world-plugin/src/components/HelloWorldPluginPage.tsx and replace its content with:

import React from 'react';

import { HelloWorldComponent } from './HelloWorldComponent';

export const HelloWorldPluginPage = () => (

<div>

<h1>Hello, Plugin!</h1>

<HelloWorldComponent />

</div>

);

1. **Create the Reusable Component**:
   * Create a new file HelloWorldComponent.tsx in the same folder:

import React from 'react';

export const HelloWorldComponent = () => (

<div>

<h2>Hello, Component!</h2>

</div>

);

1. **Register the Plugin**:
   * Open packages/app/src/App.tsx and add the plugin route:

import { HelloWorldPluginPage } from '@internal/plugin-hello-world-plugin';

const routes = (

<FlatRoutes>

<Route path="/hello-world" element={<HelloWorldPluginPage />} />

</FlatRoutes>

);

1. **Test the Plugin**:
   * Navigate to http://localhost:3000/hello-world to see:
     + "Hello, Plugin!"
     + "Hello, Component!"

**3. Create a Backend API**

**Goal**: Create a backend API that returns "Hello, API!".

1. **Create the API**:
   * Create a new file helloWorldApi.ts in packages/backend/src/apis:

import express from 'express';

const router = express.Router();

router.get('/hello', (req, res) => {

res.send('Hello, API!');

});

export default router;

1. **Register the API**:
   * Open packages/backend/src/index.ts and add the API:

import helloWorldApi from './apis/helloWorldApi';

const apiRouter = Router();

apiRouter.use('/hello-world', helloWorldApi);

1. **Test the API**:
   * Start the backend (yarn dev) and navigate to http://localhost:7007/api/hello-world/hello to see "Hello, API!".

**4. Fetch Data from the API in the Plugin**

**Goal**: Fetch data from the backend API and display it in the plugin.

1. **Update the Plugin**:
   * Open packages/plugin-hello-world-plugin/src/components/HelloWorldPluginPage.tsx and modify it to fetch data:

import React, { useEffect, useState } from 'react';

import { HelloWorldComponent } from './HelloWorldComponent';

export const HelloWorldPluginPage = () => {

const [apiResponse, setApiResponse] = useState('');

useEffect(() => {

fetch('/api/hello-world/hello')

.then((response) => response.text())

.then((data) => setApiResponse(data))

.catch((error) => console.error(error));

}, []);

return (

<div>

<h1>Hello, Plugin!</h1>

<HelloWorldComponent />

<p>API Response: {apiResponse}</p>

</div>

);

};

1. **Test the Plugin**:
   * Navigate to http://localhost:3000/hello-world to see:
     + "Hello, Plugin!"
     + "Hello, Component!"
     + "API Response: Hello, API!"

**5. Add TechDocs**

**Goal**: Add a TechDocs page and link to it from the plugin.

1. **Create Documentation**:
   * Add a docs folder to your service’s repository with an index.md file:

# Hello, TechDocs!

This is a sample documentation file.

1. **Configure TechDocs**:
   * Update app-config.yaml to enable TechDocs:

techdocs:

builder: 'local'

generator:

runIn: 'docker'

1. **Link to TechDocs**:
   * Open packages/plugin-hello-world-plugin/src/components/HelloWorldPluginPage.tsx and add a link:

import { Link } from '@backstage/core-components';

export const HelloWorldPluginPage = () => {

const [apiResponse, setApiResponse] = useState('');

useEffect(() => {

fetch('/api/hello-world/hello')

.then((response) => response.text())

.then((data) => setApiResponse(data))

.catch((error) => console.error(error));

}, []);

return (

<div>

<h1>Hello, Plugin!</h1>

<HelloWorldComponent />

<p>API Response: {apiResponse}</p>

<Link to="/docs/default/component/my-service">

View TechDocs

</Link>

</div>

);

};

1. **Test TechDocs**:
   * Navigate to http://localhost:3000/hello-world and click the "View TechDocs" link to see the rendered documentation.

**Final Output**

1. **Plugin Page**:
   * Displays "Hello, Plugin!".
   * Uses a reusable component to display "Hello, Component!".
   * Fetches and displays "Hello, API!" from the backend API.
   * Links to a TechDocs page.
2. **TechDocs Page**:
   * Displays the documentation created in the docs folder.

**Directory Structure**

my-backstage-app/

├── packages/

│ ├── app/

│ │ └── src/

│ │ └── App.tsx

│ ├── backend/

│ │ └── src/

│ │ └── apis/

│ │ └── helloWorldApi.ts

│ └── plugin-hello-world-plugin/

│ └── src/

│ └── components/

│ ├── HelloWorldPluginPage.tsx

│ └── HelloWorldComponent.tsx

├── app-config.yaml

└── docs/

└── index.md

Scenario - Backstage Setup on Ubuntu vm 22.04 [Frontend at Port 4000 & Backend on port 8000] . Using Node 18.

**1. Prerequisites**

Ensure your Ubuntu VM has the following installed:

Node, NPM, Yarn NVM and has Docker, Git setup

**2. Install the Backstage CLI**

Run the following command to install the Backstage CLI globally:

sudo npm install -g @backstage/cli

**3. Create a New Backstage App**

Use the Backstage CLI to create a new app:

npx @backstage/create-app

During the setup:

* Provide a name for your app (e.g., my-backstage-app).
* The CLI will scaffold the project with both frontend and backend code.

**4. Configure Frontend and Backend Ports**

By default, Backstage runs the frontend and backend on the same port. To run them separately on ports 4000 (frontend) and 8000 (backend), follow these steps:

**a. Configure the Backend Port**

1. Open the app-config.yaml file in the root of your project.
2. Locate the backend section and update the listen configuration:

backend:

listen:

port: 8000

**b. Configure the Frontend Port**

1. Open the app-config.yaml file.
2. Locate the app section and update the baseUrl for the backend:

app:

baseUrl: http://localhost:4000

1. ~~Open the packages/app/package.json file.~~
2. ~~Update the start script to specify the frontend port:~~

~~"start": "backstage-cli app:serve --port 4000",~~

**5. Start the Backend**

1. Navigate to the backend directory:

cd packages/backend

1. Start the backend server:

yarn start

The backend will now run on http://localhost:8000.

**6. Start the Frontend**

1. Open a new terminal window.
2. Navigate to the root of your Backstage project:

cd my-backstage-app

1. Start the frontend server:

PORT=4000 yarn start

The frontend will now run on <http://localhost:4000>.

**7. Verify Frontend and Backend Connection**

1. Open your browser and navigate to http://localhost:4000.
2. The frontend should load, and it should be able to communicate with the backend running on http://localhost:8000.
3. Check the network requests in your browser's developer tools to ensure the frontend is successfully making requests to the backend.

**8. Accessing the App from Another Machine**

If you want to access the Backstage app from another machine:

1. Ensure the ports (4000 and 8000) are open on your Ubuntu VM's firewall:

sudo ufw allow 4000

sudo ufw allow 8000

1. Replace localhost in the app-config.yaml with your VM's IP address:

app:

baseUrl: http://<VM\_IP>:4000

backend:

baseUrl: http://<VM\_IP>:8000

1. Restart both the frontend and backend servers.

**Detailed Configuration for app-config.yaml**

The app-config.yaml file is crucial for defining the connection between the frontend and backend. Here's how to configure it properly:

**1. Update app.baseUrl and backend.baseUrl**

* If you're running the app locally (on the same machine), use localhost:

app:

baseUrl: http://localhost:4000

backend:

baseUrl: http://localhost:8000

listen:

port: 8000

* If you're running the app on a VM and want to access it from another machine, replace localhost with the VM's IP address (e.g., 192.168.1.100):

app:

baseUrl: http://192.168.1.100:4000

backend:

baseUrl: http://192.168.1.100:8000

listen:

port: 8000

**2. Update cors Configuration for Backend**

Ensure the backend allows requests from the frontend by updating the cors configuration in app-config.yaml:

backend:

cors:

origin: http://192.168.1.100:4000

methods: [GET, POST, PUT, DELETE]

credentials: true

**Steps to Verify Connectivity**

1. **Start the Backend**:
   * Navigate to the packages/backend directory:

cd packages/backend

* + Start the backend:

yarn start

* + Verify the backend is running by visiting http://localhost:8000 or http://<VM\_IP>:8000 in your browser. You should see a welcome message or API documentation.

1. **Start the Frontend**:
   * Navigate to the root of your Backstage project:

cd my-backstage-app

* + Start the frontend:

yarn start

* + Verify the frontend is running by visiting http://localhost:4000 or http://<VM\_IP>:4000 in your browser. The Backstage UI should load.

1. **Check Network Requests**:
   * Open your browser's developer tools (F12).
   * Go to the "Network" tab and check if the frontend is successfully making requests to the backend (e.g., http://<VM\_IP>:8000/api/...).

# **🚀 Deploy Backstage (RoadieHQ Image) on Azure Container Apps using Azure Portal**

## **🔹 Step 1: Log in to Azure Portal**

1. Open [**Azure Portal**](https://portal.azure.com/)
2. Click on **"Create a resource"** (left sidebar)
3. In the search bar, type **"Container Apps"** and select **"Create"**

## **🔹 Step 2: Create a Resource Group**

1. In the **"Container Apps"** creation page, under **Project Details**:
   * Click **"Create new"** under **Resource Group**
   * Name it: **backstage-rg**
   * Select **Region**: Choose a region close to your users (e.g., **East US**)
   * Click **Next: Environment >**

## **🔹 Step 3: Create a Container Apps Environment**

1. Click **"Create new"** under **Container Apps Environment**
2. Name it: **backstage-env**
3. Select **Region** (same as before)
4. Keep **VNet integration disabled** (unless you need networking)
5. Click **Next: App Settings >**

## **🔹 Step 4: Deploy Backstage Container**

1. **App Name**: backstage-app
2. **Container Image Source**: **"Docker Hub or other registry"**
3. **Image URL**: roadiehq/community-backstage-image:latest
4. **CPU & Memory**: Choose **1 CPU, 2 GiB RAM**
5. **Ingress Settings**:
   * **Enable Ingress**
   * **External Access**: ✅ **Yes**
   * **Port**: 3000
6. Click **Next: Review + Create**
7. Click **Create**

## **🔹 Step 5: Get Backstage URL**

1. After deployment, go to **"Container Apps"** in the Azure Portal
2. Click on **"backstage-app"**
3. Go to **"Ingress" → "FQDN"** (Fully Qualified Domain Name)
4. Copy the URL and open it in your browser:

http://<your-backstage-url>

# **🚀 Deploy Backstage (RoadieHQ Image) on Azure Kubernetes Service (AKS) using Azure Portal**

## **🔹 Step 1: Log in to Azure Portal**

1. Open [**Azure Portal**](https://portal.azure.com/)
2. Click on **"Create a resource"** (left sidebar)
3. In the search bar, type **"Kubernetes Service"** and select **"Create"**

## **🔹 Step 2: Create an AKS Cluster**

1. **Subscription**: Choose your Azure subscription
2. **Resource Group**: Click **"Create new"**, name it: backstage-aks-rg
3. **Cluster name**: backstage-aks
4. **Region**: Select a region (e.g., **East US**)
5. **Node size**: Choose Standard\_B2s (2 vCPUs, 4 GiB RAM) for a lightweight setup
6. **Node count**: 1 (Increase later if needed)
7. Click **Review + Create** → **Create**

🎉 **Wait for deployment to complete (~5-10 mins).**

## **🔹 Step 3: Connect to Your AKS Cluster**

Once AKS is deployed:

1. Open the **Kubernetes Services** page in Azure Portal
2. Click on **backstage-aks**
3. Open **"Connect"** (left menu)
4. Copy and run the **kubectl command** in **Azure Cloud Shell**:

az aks get-credentials --resource-group backstage-aks-rg --name backstage-aks

## **🔹 Step 4: Deploy Backstage to AKS**

### ****Create a Kubernetes Deployment****

1. Open **Azure Cloud Shell**
2. Create a file backstage-deployment.yaml:

nano backstage-deployment.yaml

1. Copy and paste the following YAML:

apiVersion: apps/v1

kind: Deployment

metadata:

name: backstage

spec:

replicas: 1

selector:

matchLabels:

app: backstage

template:

metadata:

labels:

app: backstage

spec:

containers:

- name: backstage

image: roadiehq/community-backstage-image:latest

ports:

- containerPort: 7000

1. Save the file (**CTRL+X → Y → ENTER**)
2. Apply the deployment:

kubectl apply -f backstage-deployment.yaml

## **🔹 Step 5: Expose Backstage with a LoadBalancer**

1. Create a **service.yaml** file:

nano backstage-service.yaml

1. Copy and paste:

apiVersion: v1

kind: Service

metadata:

name: backstage-service

spec:

selector:

app: backstage

ports:

- protocol: TCP

port: 80

targetPort: 7000

type: LoadBalancer

1. Save the file (**CTRL+X → Y → ENTER**)
2. Apply the service:

kubectl apply -f backstage-service.yaml

## **🔹 Step 6: Get Backstage Public URL**

1. Run:

kubectl get service backstage-service

1. Copy the **EXTERNAL-IP** from the output.
2. Open it in your browser:

http://<EXTERNAL-IP>

# **Deploy RoadieHQ Backstage on Minikube (Ubuntu 20.04)**

## **🔹 Step 1: Install Minikube & Dependencies**

Ensure you have the required dependencies installed.

### ****1️⃣ Install Required Packages****

sudo apt update && sudo apt upgrade -y

sudo apt install -y curl wget apt-transport-https

### ****2️⃣ Install Docker****

sudo apt install -y docker.io

sudo usermod -aG docker $USER

newgrp docker

### ****3️⃣ Install Kubectl****

curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl"

chmod +x kubectl

sudo mv kubectl /usr/local/bin/

### ****4️⃣ Install Minikube****

curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

sudo install minikube-linux-amd64 /usr/local/bin/minikube

### ****5️⃣ Start Minikube****

minikube start --driver=docker

🚀 **Minikube will start a Kubernetes cluster locally.**

## **🔹 Step 2: Deploy Backstage to Minikube**

### ****1️⃣ Create a Deployment File****

nano backstage-deployment.yaml

Paste the following (**uses port 7000**):

apiVersion: apps/v1

kind: Deployment

metadata:

name: backstage

spec:

replicas: 1

selector:

matchLabels:

app: backstage

template:

metadata:

labels:

app: backstage

spec:

containers:

- name: backstage

image: roadiehq/community-backstage-image:latest

ports:

- containerPort: 7000

Save the file (**CTRL+X → Y → ENTER**).

### ****2️⃣ Apply the Deployment****

kubectl apply -f backstage-deployment.yaml

## **🔹 Step 3: Expose Backstage with a Service**

### ****1️⃣ Create a Service File****

nano backstage-service.yaml

Paste the following:

apiVersion: v1

kind: Service

metadata:

name: backstage-service

spec:

selector:

app: backstage

ports:

- protocol: TCP

port: 80

targetPort: 7000

type: NodePort

Save the file (**CTRL+X → Y → ENTER**).

### ****2️⃣ Apply the Service****

kubectl apply -f backstage-service.yaml

## **🔹 Step 4: Access Backstage Locally**

1. Get the service details:

kubectl get service backstage-service

1. Find the **NodePort** (e.g., 32000).
2. Get Minikube’s IP:

minikube ip

1. Access Backstage in your browser:

php-template

CopyEdit

http://<MINIKUBE-IP>:<NODEPORT>

Example:

http://192.168.49.2:32000

# **🚀 Deploy RoadieHQ Backstage with PostgreSQL & pgAdmin using Docker Compose (Minikube Ready)**

## **🔹 Step 1: Create a** docker-compose.yaml **File**

nano docker-compose.yaml

### ****🔹 Step 2: Paste the Following Configuration****

version: '3.8'

services:

postgres:

image: postgres:15

container\_name: backstage-db

restart: always

environment:

POSTGRES\_USER: backstage

POSTGRES\_PASSWORD: backstage

POSTGRES\_DB: backstage

ports:

- "5432:5432"

pgadmin:

image: dpage/pgadmin4

container\_name: backstage-pgadmin

restart: always

environment:

PGADMIN\_DEFAULT\_EMAIL: admin@backstage.com

PGADMIN\_DEFAULT\_PASSWORD: admin

ports:

- "5050:80"

depends\_on:

- postgres

backstage:

image: roadiehq/community-backstage-image:latest

container\_name: backstage-app

restart: always

environment:

DATABASE\_URL: "postgres://backstage:backstage@postgres:5432/backstage"

ports:

- "7000:7000"

depends\_on:

- postgres

### ****🔹 Step 3: Deploy Everything in One Command****

docker-compose up -d

### ****🔹 Step 4: Access the Services****

* **Backstage UI** → http://localhost:7000
* **pgAdmin UI** → http://localhost:5050
  + Login: admin@backstage.com / admin
  + Connect to DB: Host: postgres, User: backstage, Pass: backstage

# **🚀 Deploy RoadieHQ Backstage + PostgreSQL + pgAdmin on Minikube**

## **🔹 Step 1: Enable Minikube Docker & Create a Namespace**

eval $(minikube docker-env)

kubectl create namespace backstage

## **🔹 Step 2: Create Kubernetes Deployment & Services**

### ****1️⃣ Create**** backstage.yaml

nano backstage.yaml

Paste the following:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: postgres-pvc

namespace: backstage

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: postgres

namespace: backstage

spec:

replicas: 1

selector:

matchLabels:

app: postgres

template:

metadata:

labels:

app: postgres

spec:

containers:

- name: postgres

image: postgres:15

env:

- name: POSTGRES\_USER

value: "backstage"

- name: POSTGRES\_PASSWORD

value: "backstage"

- name: POSTGRES\_DB

value: "backstage"

ports:

- containerPort: 5432

volumeMounts:

- mountPath: /var/lib/postgresql/data

name: postgres-storage

volumes:

- name: postgres-storage

persistentVolumeClaim:

claimName: postgres-pvc

---

apiVersion: v1

kind: Service

metadata:

name: postgres

namespace: backstage

spec:

selector:

app: postgres

ports:

- protocol: TCP

port: 5432

targetPort: 5432

type: ClusterIP

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: backstage

namespace: backstage

spec:

replicas: 1

selector:

matchLabels:

app: backstage

template:

metadata:

labels:

app: backstage

spec:

containers:

- name: backstage

image: roadiehq/community-backstage-image:latest

env:

- name: DATABASE\_URL

value: "postgres://backstage:backstage@postgres:5432/backstage"

ports:

- containerPort: 7000

---

apiVersion: v1

kind: Service

metadata:

name: backstage

namespace: backstage

spec:

selector:

app: backstage

ports:

- protocol: TCP

port: 7000

targetPort: 7000

type: NodePort

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: pgadmin

namespace: backstage

spec:

replicas: 1

selector:

matchLabels:

app: pgadmin

template:

metadata:

labels:

app: pgadmin

spec:

containers:

- name: pgadmin

image: dpage/pgadmin4

env:

- name: PGADMIN\_DEFAULT\_EMAIL

value: "admin@backstage.com"

- name: PGADMIN\_DEFAULT\_PASSWORD

value: "admin"

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: pgadmin

namespace: backstage

spec:

selector:

app: pgadmin

ports:

- protocol: TCP

port: 5050

targetPort: 80

type: NodePort

Save and exit (**CTRL+X → Y → ENTER**).

## **🔹 Step 3: Deploy to Minikube**

kubectl apply -f backstage.yaml

## **🔹 Step 4: Access Backstage & pgAdmin**

### ****Get Minikube IP****

minikube ip

### ****Find NodePort****

kubectl get svc -n backstage

Example Output:

backstage NodePort 10.96.0.1 <none> 7000:32000/TCP

pgadmin NodePort 10.96.0.2 <none> 5050:32001/TCP

### ****Access in Browser****

* **Backstage:** http://<MINIKUBE\_IP>:<NODEPORT>  
  Example: http://192.168.49.2:32000
* **pgAdmin:** http://<MINIKUBE\_IP>:<NODEPORT>  
  Example: http://192.168.49.2:32001
  + **Login:** admin@backstage.com / admin
  + **Database Host:** postgres