Backstage

Happy developers make happy code

About Me

Fortune Ndlovu
Associate Software Engineer @RedHat
MSc Computer Science Student at SETU
RHDH Install Methods :: Red Hat Developer Hub

Lets Connect!

Workshop Structure

Step 1: History/Theory of Backstage

Step 2: Pre-Reqs & Environment Setup

Step 3: Installing Backstage

Step 4: Running the App

Step 5: Customize the App Name

Step 6: Setting Up GitHub Authentication

Step 7: Enabling Techdocs

Step 8: (Optional) Adding a plugin

Wrap-Up + Q&A

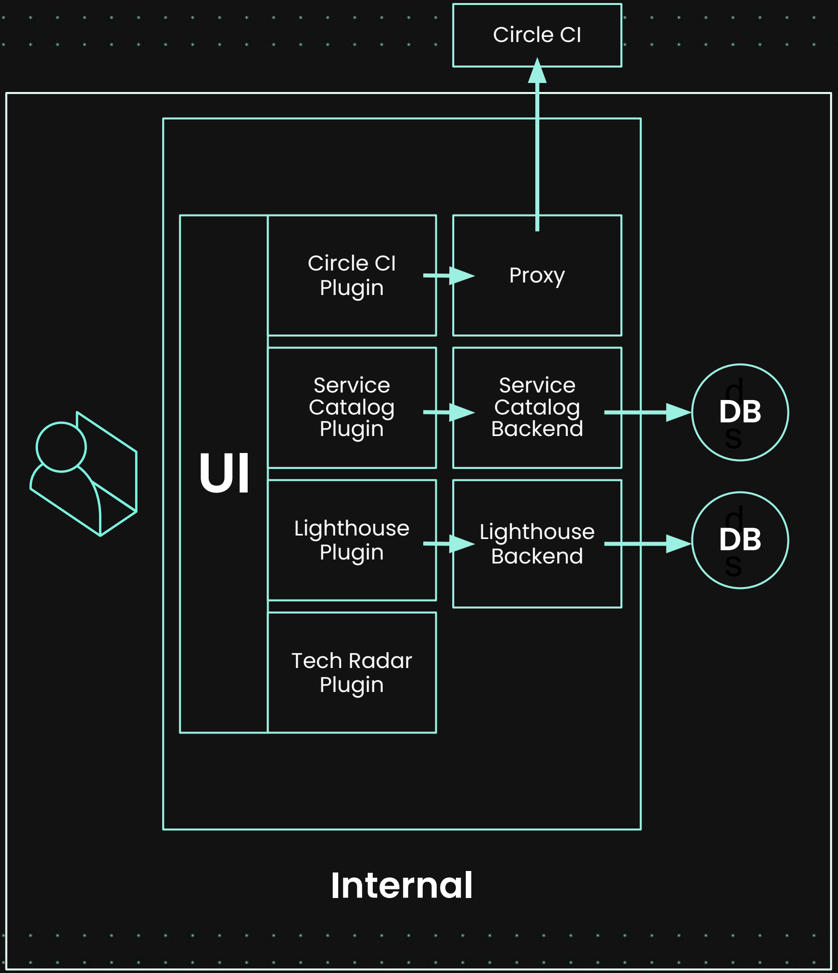
Backstage is an open platform for building developer portals.

Created at

Spotify®

Donated to

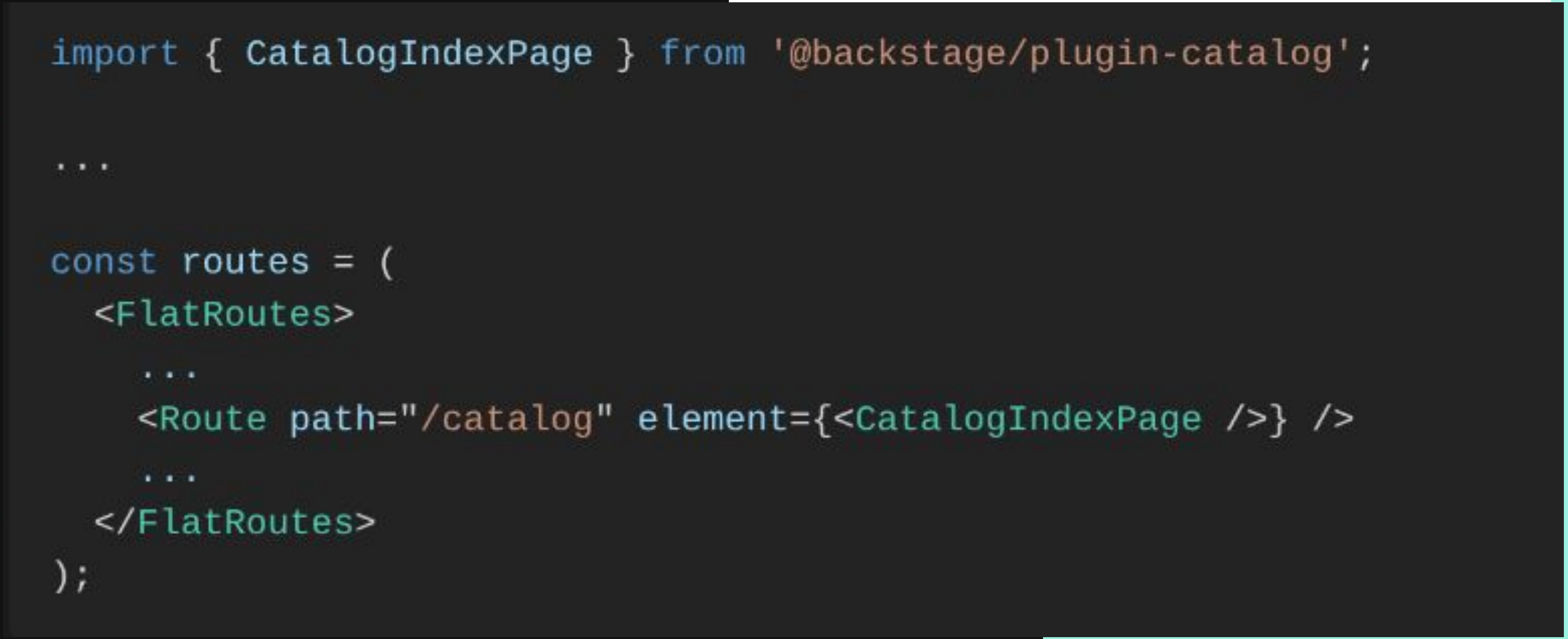
CLOUD NATIVE
COMPUTING FOUNDATION



A true platform, inside and out

- ❏ A small team maintains the core features of the Backstage app
- ❏ Different platform teams build and maintain all the other plugins
- ❏ Feature teams use the plugins to build and maintain their software (and provide feedback directly to the plugin owners)

Installing Plugins



```
import { CatalogIndexPage } from '@backstage/plugin-catalog';

...

const routes = (
  <FlatRoutes>
    ...
    <Route path="/catalog" element={<CatalogIndexPage />} />
    ...
  </FlatRoutes>
);
```

Step 2: Pre-Reqs & Environment Setup

Repository that will guide you through the workshop:

https://github.com/Fortune-Ndlovu/backstage-devconf-lab

Why do we need the pre-reqs:

Step 2: Pre-Reqs & Environment Setup

Ensures smooth installation process

Application functions correctly

Node.js: Backstage's backend and frontend are built on Node.js

Yarn: package manager Backstage uses it to manage the project's dependencies.

What's happening under the hood:
Step 3: Installing Backstage

npx @backstage/create-app

The create-app script scaffolds basic directory structure and files and triggers the installation process.

Yarn fetches all the necessary code libraries, connects the frontend and backend packages, and compiles the code to make it runnable.

At this point the scaffolded template is ready for yarn start.

Step 4: What does the folder structure mean?

This structure is what's known as a **monorepo**, which is a single repository that holds multiple distinct but related projects (or packages)

```
app // The project manager, overseeing everything
 ├── app-config.yaml // The projects settings and rule book
 ├── catalog-info.yaml // Add components to your instance
 ├── package.json // Central control file for the entire project
 └── packages // Contains all the individual parts of the project
      ├── app // The user facing storefront
      └── backend // The warehouse and business logic
```

What's happening under the hood:
Step 5: Customize App Name

Edit File → Restart Server → Backend reads the file → Frontend requests config from Backend via API → Frontend UI is rendered with the new data.

This means you can change backend configurations (like database connections or authentication secrets) without having to rebuild your frontend, and the frontend gets all the configuration it needs from a single, secure source: the backend API.

What's happening under the hood:
Step 6: Setting GitHub Authentication

1-Registering the App with GitHub

- First, we create an OAuth App in GitHub settings.
- We must provide two key URLs:
    - Homepage URL (localhost:3000): The address of our Backstage frontend.
    - Authorization Callback URL (localhost:7007/...): The specific backend API endpoint that GitHub will send the user back to after they approve the login.
- GitHub gives us a public Client ID and a private Client Secret.

What's happening under the hood:
Step 6: Setting GitHub Authentication

2-Configuring the Backstage Backend

- We securely store the Client ID and Client Secret in the app-config.local.yaml file. This file is for local secrets and isn't checked into Git.
- We then install and register the GitHub Auth Backend Plugin. This adds the necessary server-side code to handle the entire authentication handshake with GitHub.

What's happening under the hood:
Step 6: Setting GitHub Authentication

3-Updating the Backstage Frontend

- We edit the App.tsx file in the frontend package.
- This code modification adds the "Sign in with GitHub" button to the login page, giving users a way to start the process.

What's happening under the hood:
Step 6: Setting GitHub Authentication

3-The Result: The Login Handshake

- When a user clicks "Sign In," they are redirected to GitHub to approve the login.
- GitHub then redirects them back to our backend with a temporary code.
- The backend securely exchanges that code and its Client Secret for a permanent access token, confirming the user's identity and logging them into Backstage.

What's happening under the hood:
Step 7: Enabling Tech Docs

mkdocs.yml:

- This is the primary configuration file for MkDocs, the Python-based static site generator that TechDocs uses as its engine. You are telling MkDocs:
  - site_name: This sets the title within the generated HTML.
  - nav: This defines the navigation tree. MkDocs will use this to create the sidebar menu in the final output.
  - plugins: [techdocs-core]: This is the most critical part for Backstage integration. You are loading a special MkDocs plugin provided by the Backstage team. This plugin injects custom CSS for styling, modifies the HTML output to be compatible with the Backstage UI, and enables features like the "Edit this page" button.

What's happening under the hood:
Step 7: Enabling Tech Docs

/docs/index.md:

- You are creating the raw source material. MkDocs is configured by default to look for a docs directory and will parse any Markdown (.md) files it finds there into HTML.

catalog-info.yaml Annotation:

- This is the metadata "glue" that connects a Catalog Component to its documentation. When Backstage processes this component, it reads the annotations section. The backstage.io/techdocs-ref: dir:. annotation specifically tells the TechDocs backend: "For this component, the source code for its documentation is located in the same directory (dir:.) as this catalog-info.yaml file." This is how Backstage knows *where* to find your Markdown files in the Git repository.

What's happening under the hood:
Step 8: AI Assistant In Backstage (Ollama + LLaMa 3)

1. Ollama (The LLM Server):
   - Ollama is a lightweight, extensible server application that runs locally on your machine. Its sole job is to download, manage, and serve large language models through a simple REST API.
   - Under the Hood (ollama serve): This command starts a background daemon that listens for HTTP requests, by default on port 11434. It's the engine waiting for a prompt.
   - Under the Hood (ollama pull llama3): This command connects to Ollama's model registry, downloads the multi-gigabyte LLaMA 3 model files, and stores them in a local cache, making them available to the ollama serve process.

What's happening under the hood:
Step 8: AI Assistant In Backstage (Ollama + LLaMa 3)

LLaMA 3 (The Brain):

- This is the actual LLM. It's a massive neural network trained by Meta that is capable of understanding and generating human-like text. It runs *on* the Ollama server.

@roadiehq/rag-ai-backend (The Backstage API Layer):

- This is a pre-built Backstage backend plugin. Its purpose is to provide the API endpoints (/api/rag-ai/...) and business logic required for an AI chat experience. As its name suggests, it's primarily designed for RAG (Retrieval-Augmented Generation), but you are cleverly using it in a non-RAG, direct-to-model mode.

What's happening under the hood:
Step 8: AI Assistant In Backstage (Ollama + LLaMa 3)

his is what happens in milliseconds when we ask, "What does Deep Learning mean?":

1. Frontend (React): The <RagModal /> component captures our question from the input field.
2. Frontend (API Call): It makes an HTTP POST request to the Backstage backend endpoint: http://localhost:7007/api/rag-ai/ask. The body of the request is a JSON object containing our question.
3. Backend (Routing): The Backstage backend server receives the request and, based on the URL, routes it to the ragAi.router we registered.
4. Backend (Plugin Logic): The API handler for the /ask route inside the @roadiehq/rag-ai-backend plugin is executed.
5. Backend (RAG Skip): The plugin attempts to run the retrieval pipeline, but since we provided the dummyRetrievalPipeline, it does nothing and immediately proceeds.
6. Backend (LangChain Connector): The plugin takes our question and passes it to the model.invoke("What does...") method on the Ollama LangChain object.

What's happening under the hood:
Step 8: AI Assistant In Backstage (Ollama + LLaMa 3)

1. Backend -> Ollama (HTTP): The LangChain Ollama object constructs a low-level HTTP POST request to http://localhost:11434/api/generate and sends your prompt to the Ollama server.
2. Ollama -> LLaMA 3: The Ollama server passes the prompt to the loaded LLaMA 3 model for processing.
3. LLaMA 3 -> Ollama: LLaMA 3 generates the answer and streams it back to the Ollama server.
4. Ollama -> Backend: The Ollama server streams the response back to your Backstage backend via the open HTTP connection.
5. Backend -> Frontend: The Backstage backend streams the final answer back to the <RagModal /> component in your browser as the response to its initial API call.
6. Frontend (Render): The <RagModal /> component receives the response, updates its React state, and renders the AI's answer on the screen.

Thank you

Fortune Ndlovu
Associate Software Engineer @RedHat
MSc Computer Science Student at SETU
RHDH Install Methods :: Red Hat Developer Hub

Lets Connect!