# ****Self-Service Portal for MEAN/MERN Deployment with Backstage Integration****

## ****📌 Objective****

This project aims to build a **Self-Service Portal** that allows users to deploy **MEAN/MERN stack applications** seamlessly. The portal integrates with **Backstage**, automates infrastructure using **Terraform**, and deploys applications on **Google Kubernetes Engine (GKE) Autopilot** using **Harness CI/CD**.

## ****📍 Architecture Overview****

### ****🔹 High-Level Components****

1. **Backstage Developer Portal** → Self-service UI for deployments
2. **MERN/MEAN Stack Application** → User-defined application to be deployed
3. **Node.js Backend** → API handling deployment requests
4. **Terraform** → Infrastructure automation (GKE, VPC, IAM, Storage)
5. **Harness CI/CD** → Automating Kubernetes deployment
6. **Google Kubernetes Engine (GKE) Autopilot** → Hosting the application

### ****🔹 Architecture Diagram****

User → Backstage UI → Node.js Backend → Terraform → GKE Autopilot → Harness CI/CD

## ****📌 Project Workflow****

1. **User selects a template** (MEAN/MERN) in **Backstage UI**.
2. **Node.js backend** triggers **Terraform** to create required infrastructure on **GCP**.
3. Once **GKE Autopilot** is ready, the **code is pushed to GitHub**.
4. **Harness CI/CD** pipeline deploys the application on **GKE**.
5. **Backstage UI** shows deployment status and logs.

### ****Developing a Self-Service Employee Onboarding System in Backstage****

## ****1. Define Requirements****

* Collect essential employee details (e.g., Full Name, Email, Role, Team).
* Define the onboarding workflow (Form Submission → Approval → Provisioning).
* Design the Backstage UI card for a seamless user experience.

## ****2. Set Up Backstage Environment****

* Ensure **Backstage is installed and running** on your system.
* If not, follow the official Backstage documentation to set it up.

**📌 User Stories**

| **ID** | **User Story** | **Description** |
| --- | --- | --- |
| **US-1** | **As a user, I want to deploy a MEAN/MERN stack application from a web UI** | So that I can quickly set up my project on **GKE Autopilot** |
| **US-2** | **As a user, I want to see a Backstage Card for my deployments** | So that I can track all my **service requests** |
| **US-3** | **As a user, I want to trigger GitHub code pull & setup using Backstage** | So that my service is automatically cloned & configured |
| **US-4** | **As a user, I want to trigger a CI/CD pipeline using a Backstage card** | So that my application is deployed using **Harness & GKE** |
| **US-5** | **As a user, I want to view the final deployed application via a Backstage card** | So that I can easily access my running web app |

## ****3. Create a React Form for Onboarding****

Use react-hook-form or formik to create an onboarding form in React.

### ****Example: Employee Onboarding Form (React)****

import React from 'react';

import { useForm } from 'react-hook-form';

const OnboardingForm = () => {

const { register, handleSubmit } = useForm();

const onSubmit = async (data) => {

try {

const response = await fetch('/api/onboard', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify(data),

});

if (response.ok) {

alert('Onboarding request submitted successfully!');

} else {

alert('Failed to submit onboarding request.');

}

} catch (error) {

console.error('Error:', error);

}

};

return (

<form onSubmit={handleSubmit(onSubmit)}>

<label>Full Name:

<input {...register('fullName', { required: true })} />

</label>

<label>Email:

<input type="email" {...register('email', { required: true })} />

</label>

<label>Role:

<select {...register('role', { required: true })}>

<option value="developer">Developer</option>

<option value="designer">Designer</option>

</select>

</label>

<button type="submit">Submit</button>

</form>

);

};

export default OnboardingForm;

## ****4. Build the Backend API for Onboarding****

Create an API to process onboarding requests using **Node.js (Express)** or **Python (Flask)**.

### ****Example: Express.js API****

javascript

CopyEdit

const express = require('express');

const app = express();

app.use(express.json());

app.post('/api/onboard', (req, res) => {

const employeeData = req.body;

console.log('Received onboarding data:', employeeData);

// Process data (e.g., save to database, trigger workflows)

res.status(200).send('Onboarding request received');

});

app.listen(3000, () => console.log('Server running on port 3000'));

## ****5. Integrate the Onboarding Form into Backstage****

* Create a **custom plugin** in Backstage to display the onboarding form.
* Use createPlugin and createCard utilities to integrate the form.

### ****Example: Onboarding Card Component****

import React from 'react';

import { Content, Page } from '@backstage/core-components';

import OnboardingForm from './OnboardingForm';

const OnboardingCard = () => (

<Page themeId="tool">

<Content>

<h1>Employee Onboarding</h1>

<OnboardingForm />

</Content>

</Page>

);

export default OnboardingCard;

## ****6. Register the Plugin in Backstage****

Create a **Backstage Plugin** and register it.

### ****Example: Plugin Registration****

import { createPlugin } from '@backstage/core';

import OnboardingCard from './OnboardingCard';

export const onboardingPlugin = createPlugin({

id: 'onboarding',

register({ router }) {

router.registerRoute('/', OnboardingCard);

},

});

## ****7. Add the Onboarding Card to Backstage Homepage****

Display the onboarding card on the homepage or entity page.

### ****Example: Homepage Integration****

import { HomePageToolkit } from '@backstage/plugin-home';

import OnboardingForm from './OnboardingForm';

import PersonIcon from '@material-ui/icons/Person';

const HomePage = () => (

<HomePageToolkit

tools={[

{

label: 'Employee Onboarding',

icon: <PersonIcon />,

content: <OnboardingForm />,

},

]}

/>

);

## ****8. Handle Form Submission****

Modify the onSubmit function to send data to the backend API.

const onSubmit = async (data) => {

try {

const response = await fetch('/api/onboard', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify(data),

});

if (response.ok) {

alert('Onboarding request submitted successfully!');

} else {

alert('Failed to submit onboarding request.');

}

} catch (error) {

console.error('Error:', error);

}

};

## ****🛠 Step-by-Step Implementation****

### ****1️⃣ Set Up Backstage****

1. Install Backstage

npx @backstage/create-app@latest

cd backstage-app

yarn install

yarn dev

1. Create a **custom plugin** for the deployment portal.
2. Add a **React form** for user inputs (stack, repo URL, GCP settings).

#### **📌 Code: Backstage UI Plugin**

import React from 'react';

import { useForm } from 'react-hook-form';

const DeploymentForm = () => {

const { register, handleSubmit } = useForm();

const onSubmit = async (data) => {

const response = await fetch('/api/deploy', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify(data),

});

if (response.ok) {

alert('Deployment request submitted successfully!');

} else {

alert('Failed to submit deployment request.');

}

};

return (

<form onSubmit={handleSubmit(onSubmit)}>

<label>Project Name:

<input {...register('projectName', { required: true })} />

</label>

<label>Stack:

<select {...register('stack', { required: true })}>

<option value="mern">MERN</option>

<option value="mean">MEAN</option>

</select>

</label>

<label>GitHub Repo URL:

<input {...register('repoUrl', { required: true })} />

</label>

<button type="submit">Deploy</button>

</form>

);

};

export default DeploymentForm;

### ****2️⃣ Node.js Backend****

Handles **deployment requests** from **Backstage UI** and triggers **Terraform**.

#### **📌 Code: Node.js API**

const express = require('express');

const { exec } = require('child\_process');

const app = express();

app.use(express.json());

app.post('/api/deploy', (req, res) => {

const { projectName, stack, repoUrl } = req.body;

// Execute Terraform Script

exec(`terraform apply -auto-approve -var="project\_name=${projectName}" -var="repo\_url=${repoUrl}" -var="stack=${stack}"`, (error, stdout, stderr) => {

if (error) {

console.error(`Error: ${error.message}`);

return res.status(500).send('Deployment failed');

}

console.log(`Output: ${stdout}`);

res.status(200).send('Deployment initiated successfully');

});

});

app.listen(4000, () => console.log('Backend running on port 4000'));

### ****3️⃣ Terraform for GKE Autopilot****

Creates **GKE cluster, VPC, IAM roles, and storage**.

#### **📌 Code: Terraform Configuration**

provider "google" {

project = var.project\_name

region = "us-central1"

}

resource "google\_container\_cluster" "autopilot" {

name = "gke-autopilot-cluster"

location = "us-central1"

enable\_autopilot = true

}

variable "project\_name" {}

variable "repo\_url" {}

variable "stack" {}

### ****4️⃣ Harness CI/CD Pipeline****

1. **Triggers on GitHub Push**
2. **Builds Docker Image & Deploys to GKE**

#### **📌 Harness CI/CD YAML**

pipeline:

name: GKE Deployment

identifier: GKE\_Deployment

stages:

- stage:

name: Build

identifier: Build

steps:

- step:

name: Build Docker Image

identifier: BuildDocker

type: BuildAndPushDockerRegistry

spec:

connectorRef: DockerHub

repository: my-gke-app

tag: latest

- stage:

name: Deploy to GKE

identifier: Deploy

steps:

- step:

name: Deploy

identifier: DeployK8s

type: K8sRollingDeploy

spec:

connectorRef: GKE

namespace: default

manifests:

- manifest:

identifier: K8sManifest

type: K8sManifest

spec:

store:

type: Git

spec:

connectorRef: GitHub

repoName: my-gke-app

paths:

- k8s/deployment.yaml

### ****5️⃣ Deploy & Validate****

1. Run **Backstage UI**

yarn dev

1. Start **Node.js API**

node server.js

1. Initialize **Terraform**

terraform init

terraform apply -auto-approve

1. Verify **GKE Deployment**

kubectl get pods -n default

## ****📦 Codebase in GitHub****

A complete **GitHub repository** is required for this project. You can create it and structure it as:

/self-service-portal

│── backstage-app/ # Backstage UI

│── backend-api/ # Node.js Backend

│── terraform/ # Terraform GKE Setup

│── harness-ci/ # Harness CI/CD Pipeline

│── mean-mern-app/ # Sample MERN/MEAN App

│── README.md

**📍 Backstage Self-Service Web App Implementation**

We will implement a **React-based self-service web app** inside **Backstage** as a **custom plugin**.

**📌 UI Design**

The UI consists of:  
✅ **Deployment Form** (User inputs project details)  
✅ **Backstage Cards** (Trigger actions)  
✅ **Deployment Status & Output Link**

**1️⃣ Backstage Plugin: Self-Service Portal**

**📌 Steps to Create the Backstage Plugin**

1. Navigate to Backstage project

cd backstage-app

1. Generate a new plugin

yarn create-plugin self-service

1. Modify **src/plugins/self-service/index.ts**

import { createPlugin } from '@backstage/core';

import SelfServicePage from './SelfServicePage';

export const selfServicePlugin = createPlugin({

id: 'self-service',

register({ router }) {

router.addRoute('/', SelfServicePage);

},

});

1. Update **src/plugins/self-service/SelfServicePage.tsx**

**2️⃣ React Web App for Self-Service Portal**

A **React-based form** is used to submit deployment requests.

**📌 Code: SelfServicePage.tsx**

import React, { useState } from 'react';

const SelfServicePage = () => {

const [projectName, setProjectName] = useState('');

const [stack, setStack] = useState('mern');

const [repoUrl, setRepoUrl] = useState('');

const [status, setStatus] = useState('');

const deployProject = async () => {

const response = await fetch('/api/deploy', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ projectName, stack, repoUrl }),

});

if (response.ok) {

setStatus('Deployment started!');

} else {

setStatus('Deployment failed.');

}

};

return (

<div>

<h2>Deploy Your Application</h2>

<label>Project Name:

<input type="text" value={projectName} onChange={(e) => setProjectName(e.target.value)} />

</label>

<label>Stack:

<select value={stack} onChange={(e) => setStack(e.target.value)}>

<option value="mern">MERN</option>

<option value="mean">MEAN</option>

</select>

</label>

<label>GitHub Repo URL:

<input type="text" value={repoUrl} onChange={(e) => setRepoUrl(e.target.value)} />

</label>

<button onClick={deployProject}>Deploy</button>

<p>{status}</p>

</div>

);

};

export default SelfServicePage;

**3️⃣ Backstage Cards for Workflow Actions**

Each **Backstage Card** triggers a separate service:  
✅ **Card 1:** Clone Repo from GitHub  
✅ **Card 2:** Deploy to GKE using Terraform  
✅ **Card 3:** Trigger Harness CI/CD Pipeline  
✅ **Card 4:** View Deployed Web App

**📌 Code: DeploymentCard.tsx**

import React, { useState } from 'react';

const DeploymentCard = () => {

const [status, setStatus] = useState('Waiting for action');

const triggerAction = async (endpoint: string) => {

const response = await fetch(`/api/${endpoint}`, { method: 'POST' });

if (response.ok) {

setStatus(`Triggered: ${endpoint}`);

} else {

setStatus('Failed to trigger');

}

};

return (

<div>

<h3>Self-Service Actions</h3>

<button onClick={() => triggerAction('clone-repo')}>Clone GitHub Repo</button>

<button onClick={() => triggerAction('deploy-gke')}>Deploy to GKE</button>

<button onClick={() => triggerAction('trigger-cicd')}>Trigger Harness CI/CD</button>

<button onClick={() => window.open('https://deployed-app-url.com')}>View App</button>

<p>{status}</p>

</div>

);

};

export default DeploymentCard;

**4️⃣ Node.js Backend API**

Handles Backstage requests.

const express = require('express');

const { exec } = require('child\_process');

const app = express();

app.use(express.json());

app.post('/api/clone-repo', (req, res) => {

exec(`git clone ${req.body.repoUrl} repo-folder`, (error) => {

if (error) return res.status(500).send('Cloning failed');

res.send('Repository cloned');

});

});

app.post('/api/deploy-gke', (req, res) => {

exec(`terraform apply -auto-approve`, (error) => {

if (error) return res.status(500).send('Deployment failed');

res.send('GKE deployment started');

});

});

app.post('/api/trigger-cicd', (req, res) => {

exec(`harness-cli deploy -p project-id`, (error) => {

if (error) return res.status(500).send('CI/CD pipeline failed');

res.send('CI/CD pipeline triggered');

});

});

app.listen(4000, () => console.log('API Running on 4000'));

**5️⃣ Deploy & Verify**

1️⃣ **Run Backstage UI**

yarn dev

2️⃣ **Start Node.js API**

node server.js

3️⃣ **Run Terraform**

terraform init

terraform apply -auto-approve

4️⃣ **Verify GKE Deployment**

kubectl get pods -n default

**📦 Codebase in GitHub**

| **Folder** | **Description** |
| --- | --- |
| backstage-app/ | Backstage UI Plugin |
| backend-api/ | Node.js API |
| terraform/ | Terraform for GKE |
| harness-ci/ | Harness CI/CD Pipeline |
| mean-mern-app/ | Sample App |

To **automate the entire Self-Service Portal** using **GoLang and Terraform**, we need:

✅ **Go Lang API** to handle requests & automation  
✅ **Terraform** for **GKE deployment**  
✅ **GitHub Actions** or **Harness CI/CD** to automate deployments  
✅ **Backstage Plugin** for UI  
✅ **Execution Flow & Architecture Diagram**

**📌 End-to-End Setup for Self-Service Portal with GoLang & Terraform**

**🛠 Architecture Diagram (Rough Sketch)**

[ User ]

|

▼

[ Backstage Self-Service Portal ]

|

▼

[ GoLang API ] -- Calls Terraform to Deploy --> [ GKE ]

|

▼

[ GitHub Repo Clone ] -- Deploys App --> [ Harness CI/CD ]

**1️⃣ Setup: GoLang API for Automation**

This **GoLang API** will:  
✅ Handle API requests from **Backstage**  
✅ Execute **Terraform** to deploy GKE  
✅ Trigger **CI/CD Pipeline**

**📌 GoLang API (main.go)**

package main

import (

"encoding/json"

"fmt"

"log"

"net/http"

"os/exec"

)

// Request struct

type DeployRequest struct {

RepoURL string `json:"repo\_url"`

Stack string `json:"stack"`

}

func deployHandler(w http.ResponseWriter, r \*http.Request) {

var req DeployRequest

err := json.NewDecoder(r.Body).Decode(&req)

if err != nil {

http.Error(w, "Invalid request", http.StatusBadRequest)

return

}

// Clone GitHub Repository

cmd := exec.Command("git", "clone", req.RepoURL, "./app")

if err := cmd.Run(); err != nil {

http.Error(w, "Failed to clone repo", http.StatusInternalServerError)

return

}

// Deploy to GKE using Terraform

cmd = exec.Command("terraform", "apply", "-auto-approve")

if err := cmd.Run(); err != nil {

http.Error(w, "Terraform deployment failed", http.StatusInternalServerError)

return

}

// Trigger CI/CD

cmd = exec.Command("harness-cli", "deploy", "-p", "project-id")

if err := cmd.Run(); err != nil {

http.Error(w, "CI/CD Trigger failed", http.StatusInternalServerError)

return

}

w.WriteHeader(http.StatusOK)

fmt.Fprint(w, "Deployment Started Successfully")

}

func main() {

http.HandleFunc("/deploy", deployHandler)

log.Println("Server running on port 8080")

log.Fatal(http.ListenAndServe(":8080", nil))

}

**📌 Steps to Run:**

go mod init self-service

go mod tidy

go run main.go

**2️⃣ Terraform Configuration for GKE**

Terraform will:  
✅ **Provision GKE Cluster**  
✅ **Deploy the MEAN/MERN App**

**📌 main.tf**

provider "google" {

project = "your-gcp-project-id"

region = "us-central1"

}

resource "google\_container\_cluster" "gke\_cluster" {

name = "mern-mean-cluster"

location = "us-central1"

initial\_node\_count = 2

}

resource "kubernetes\_deployment" "mern\_app" {

metadata {

name = "mern-app"

namespace = "default"

}

spec {

replicas = 2

selector {

match\_labels = {

app = "mern"

}

}

template {

metadata {

labels = {

app = "mern"

}

}

spec {

container {

image = "gcr.io/your-project/mern-app:latest"

name = "mern-container"

port {

container\_port = 3000

}

}

}

}

}

}

**📌 Run Terraform**

terraform init

terraform apply -auto-approve

**3️⃣ Automate CI/CD using Harness**

✅ Harness deploys the app on **GKE**  
✅ Triggers pipeline via **GoLang API**

**📌 Harness Pipeline**

pipeline:

name: Deploy MEAN/MERN App

stages:

- name: Build

type: CI

steps:

- name: Build Docker Image

image: node:14

commands:

- npm install

- npm run build

- docker build -t gcr.io/your-project/mern-app:latest .

- docker push gcr.io/your-project/mern-app:latest

- name: Deploy

type: CD

steps:

- name: Deploy to GKE

type: Kubernetes

spec:

manifests:

- manifestType: K8s

filePath: "k8s/deployment.yaml"

**📌 Run Harness Deployment**

harness-cli deploy -p project-id

**4️⃣ Backstage UI Integration**

✅ **React-based UI** to trigger deployments  
✅ Calls **GoLang API** for automation

**📌 Backstage Plugin UI (SelfService.tsx)**

import React, { useState } from 'react';

const SelfService = () => {

const [repoUrl, setRepoUrl] = useState('');

const [stack, setStack] = useState('mern');

const [status, setStatus] = useState('');

const deploy = async () => {

const response = await fetch('/api/deploy', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ repoUrl, stack }),

});

if (response.ok) {

setStatus('Deployment started!');

} else {

setStatus('Deployment failed.');

}

};

return (

<div>

<h2>Deploy Your Application</h2>

<input type="text" value={repoUrl} onChange={(e) => setRepoUrl(e.target.value)} />

<select value={stack} onChange={(e) => setStack(e.target.value)}>

<option value="mern">MERN</option>

<option value="mean">MEAN</option>

</select>

<button onClick={deploy}>Deploy</button>

<p>{status}</p>

</div>

);

};

export default SelfService;

**5️⃣ Execution Flow**

**🛠 Steps to Deploy Everything in One Go**

1️⃣ Start **GoLang API**

go run main.go

2️⃣ Initialize & Deploy **Terraform**

sh

CopyEdit

terraform init

terraform apply -auto-approve

3️⃣ Start **Backstage**

yarn dev

4️⃣ Deploy via **Harness CI/CD**

harness-cli deploy -p project-id

**🚀 Complete Setup Guide: Self-Service Deployment Portal on Ubuntu VM**

This guide will walk you through setting up:  
✅ **Ubuntu VM** with **Node.js 18+, Docker, Python, Go, Terraform**  
✅ Running **Backstage**, **GoLang API**, and **Terraform**  
✅ Calling **Go from React** and **Terraform from React**  
✅ **Automating Deployment to GKE**

**1️⃣ Ubuntu VM: Install Required Software**

**Login to your Ubuntu VM and run:**

sudo apt update && sudo apt upgrade -y

**📌 Install Node.js 18+**

curl -fsSL https://deb.nodesource.com/setup\_18.x | sudo -E bash -

sudo apt install -y nodejs

node -v # Check version

**📌 Install Docker**

sudo apt install -y docker.io

sudo systemctl enable docker

sudo systemctl start docker

docker --version # Verify installation

**📌 Install Python**

sudo apt install -y python3 python3-pip

python3 --version # Verify installation

**📌 Install GoLang**

wget https://go.dev/dl/go1.21.0.linux-amd64.tar.gz

sudo tar -C /usr/local -xzf go1.21.0.linux-amd64.tar.gz

echo 'export PATH=$PATH:/usr/local/go/bin' >> ~/.bashrc

source ~/.bashrc

go version # Verify installation

**📌 Install Terraform**

wget https://releases.hashicorp.com/terraform/1.6.0/terraform\_1.6.0\_linux\_amd64.zip

unzip terraform\_1.6.0\_linux\_amd64.zip

sudo mv terraform /usr/local/bin/

terraform --version # Verify installation

**2️⃣ Setup GoLang API**

**📌 Create and Run Go API**

mkdir ~/self-service-portal && cd ~/self-service-portal

go mod init self-service

touch main.go

nano main.go

Paste the following code inside main.go:

package main

import (

"encoding/json"

"fmt"

"log"

"net/http"

"os/exec"

)

type DeployRequest struct {

RepoURL string `json:"repo\_url"`

}

func deployHandler(w http.ResponseWriter, r \*http.Request) {

var req DeployRequest

err := json.NewDecoder(r.Body).Decode(&req)

if err != nil {

http.Error(w, "Invalid request", http.StatusBadRequest)

return

}

// Execute Terraform

cmd := exec.Command("terraform", "apply", "-auto-approve")

if err := cmd.Run(); err != nil {

http.Error(w, "Terraform deployment failed", http.StatusInternalServerError)

return

}

w.WriteHeader(http.StatusOK)

fmt.Fprint(w, "Deployment Started Successfully")

}

func main() {

http.HandleFunc("/deploy", deployHandler)

log.Println("Go Server running on port 8080")

log.Fatal(http.ListenAndServe(":8080", nil))

}

**Run the Go API:**

go run main.go

**3️⃣ Setup Terraform**

**📌 Terraform main.tf**

mkdir ~/self-service-portal/terraform && cd ~/self-service-portal/terraform

nano main.tf

Paste the following Terraform script:

provider "google" {

project = "your-gcp-project-id"

region = "us-central1"

}

resource "google\_container\_cluster" "gke\_cluster" {

name = "mern-mean-cluster"

location = "us-central1"

initial\_node\_count = 2

}

**Initialize and Apply Terraform:**

terraform init

terraform apply -auto-approve

**4️⃣ Setup React Frontend**

**📌 Install React & Create Project**

cd ~/self-service-portal

npx create-react-app frontend

cd frontend

npm install

**📌 Update React Frontend to Call Go API**

Edit src/App.js:

import React, { useState } from 'react';

function App() {

const [repoUrl, setRepoUrl] = useState('');

const [status, setStatus] = useState('');

const deploy = async () => {

const response = await fetch('http://localhost:8080/deploy', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ repo\_url: repoUrl }),

});

if (response.ok) {

setStatus('Deployment Started!');

} else {

setStatus('Deployment Failed.');

}

};

return (

<div>

<h2>Deploy App</h2>

<input

type="text"

placeholder="GitHub Repo URL"

value={repoUrl}

onChange={(e) => setRepoUrl(e.target.value)}

/>

<button onClick={deploy}>Deploy</button>

<p>{status}</p>

</div>

);

}

export default App;

**Run React App:**

npm start

**5️⃣ How to Call Terraform from React**

You **can't directly call Terraform from React** because Terraform needs CLI execution. Instead,  
✅ **Call Go API** ➝ **Go Calls Terraform**

Modify main.go:

cmd := exec.Command("terraform", "apply", "-auto-approve")

This **executes Terraform** when React calls the Go API.

**6️⃣ Automating the Entire Process**

You can **automate the setup** using a **Bash script**:

nano setup.sh

Paste:

#!/bin/bash

echo "Setting up Self-Service Portal..."

# Install dependencies

sudo apt update && sudo apt install -y nodejs docker.io python3 go unzip

# Setup Go API

mkdir ~/self-service-portal && cd ~/self-service-portal

go mod init self-service

cat <<EOT >> main.go

package main

import ("encoding/json"; "fmt"; "log"; "net/http"; "os/exec")

type DeployRequest struct { RepoURL string `json:"repo\_url"` }

func deployHandler(w http.ResponseWriter, r \*http.Request) {

var req DeployRequest

json.NewDecoder(r.Body).Decode(&req)

cmd := exec.Command("terraform", "apply", "-auto-approve")

cmd.Run()

w.WriteHeader(http.StatusOK)

fmt.Fprint(w, "Deployment Started Successfully")

}

func main() { http.HandleFunc("/deploy", deployHandler); log.Fatal(http.ListenAndServe(":8080", nil)) }

EOT

go run main.go &

# Setup Terraform

mkdir ~/self-service-portal/terraform && cd ~/self-service-portal/terraform

cat <<EOT >> main.tf

provider "google" { project = "your-gcp-project-id"; region = "us-central1" }

resource "google\_container\_cluster" "gke\_cluster" { name = "mern-mean-cluster"; location = "us-central1"; initial\_node\_count = 2 }

EOT

terraform init && terraform apply -auto-approve

# Setup React

cd ~/self-service-portal

npx create-react-app frontend

cd frontend

npm install

cat <<EOT >> src/App.js

import React, { useState } from 'react';

function App() {

const [repoUrl, setRepoUrl] = useState('');

const [status, setStatus] = useState('');

const deploy = async () => {

const response = await fetch('http://localhost:8080/deploy', { method: 'POST', headers: { 'Content-Type': 'application/json' }, body: JSON.stringify({ repo\_url: repoUrl }) });

if (response.ok) setStatus('Deployment Started!'); else setStatus('Deployment Failed.');

};

return (<div><h2>Deploy App</h2><input type="text" placeholder="GitHub Repo URL" value={repoUrl} onChange={(e) => setRepoUrl(e.target.value)} /><button onClick={deploy}>Deploy</button><p>{status}</p></div>);

}

export default App;

EOT

npm start

Run: chmod +x setup.sh ./setup.sh

**Backstage UI Cards for Self-Service Portal**

1. **User Onboarding to a Project** 🚀
   * Self-register for a project.
   * Capture user details, roles, and service selection.
   * Stores onboarding requests and approvals.
2. **Register for a Service** 🏗️
   * Users can select a service:
     + **Reusable Components Practice [MEAN Stack]**
     + **Reusable Software Practice [MERN Stack]**
   * Triggers approval workflows if needed.
3. **Deploy a Service** 🛠️
   * Automates deployment of:
     + **MEAN Stack** (*Reusable Components Practice*)
     + **MERN Stack** (*Reusable Software Practice*)
   * Uses Terraform & Harness CI/CD.
4. **Trigger CI/CD Pipeline** 🔄
   * Allows users to trigger a CI/CD pipeline in Harness.io.
   * Supports version selection and rollback.
5. **View Deployment & Pipeline Status** 📊
   * Real-time updates on deployments & pipeline executions.

