5 Essential DevOps Mini Projects

Table of Contents

1. Automated Deployment Pipeline with CI/CD Using Jenkins

- Overview
- Step 1: Prerequisites
- Step 2: Setting Up Jenkins
    - Installation
    - Configurations
- Step 3: Preparing the Application
    - Creating and Pushing a Repository
- Step 4: Configuring the Jenkins Pipeline
    - Writing the Pipeline Script
- Step 5: Enhancing the Pipeline
    - Adding Webhooks
    - Integrating Docker
    - Deploying to a Server
- Step 6: Testing and Debugging
- Step 7: Best Practices
- Key Takeaways

---

2. Infrastructure as Code (IaC) with Terraform

- Overview
- Step 1: Prerequisites

- Step 2: Creating the Terraform Configuration
    - Setting Up the Project Directory
    - Defining Resources: VPC, Subnets, Internet Gateway, and EC2
- Step 3: Initializing and Applying Terraform
    - Initializing Terraform
    - Validating and Applying Configuration
- Step 4: Verifying the Deployment
    - Accessing Resources via the AWS Console
    - Connecting to the EC2 Instance
- Step 5: Enhancements
    - Adding Elastic Load Balancers
    - Securing with Security Groups
- Step 6: Destroying the Resources
- Key Takeaways

---

3. Containerized Application Deployment with Docker Compose

- Overview
- Step 1: Prerequisites
- Step 2: Preparing the Application
    - Creating a Web Application
    - Adding Database Initialization Scripts
- Step 3: Writing Dockerfiles
    - Dockerfile for the Web Application
- Step 4: Creating the docker-compose.yml File
    - Defining Services: App and Database
- Step 5: Building and Running the Application

    - Building Docker Images
    - Running the Containers
- Step 6: Testing and Debugging
- Step 7: Enhancements
    - Adding Environment Variables
    - Scaling Containers
- Step 8: Cleanup
- Key Takeaways

---

4. Kubernetes Deployment for a Multi-Tier Application

- Overview
- Step 1: Prerequisites
- Step 2: Building the Application
    - Creating Web and Database Components
    - Writing Dockerfiles
- Step 3: Writing Kubernetes Manifests
    - Deployment for Web Application
    - Service for Web Application
    - Deployment for Database
    - Service for Database
- Step 4: Deploying the Application
    - Applying Manifests
    - Accessing the Application
- Step 5: Enhancements
    - Adding Probes for Health Checks
    - Using Persistent Volumes

    - Scaling Deployments
- Step 6: Cleanup
- Key Takeaways

---

5. Automated Configuration Management with Ansible

- Overview
- Step 1: Prerequisites
- Step 2: Setting Up the Ansible Inventory
- Step 3: Writing the Ansible Playbook
    - Installing and Configuring Apache
    - Deploying MySQL and Securing Installation
    - Adding PHP and Application Deployment
- Step 4: Running the Playbook
- Step 5: Verifying the Deployment
    - Accessing the Application
    - Checking Service Statuses
- Step 6: Enhancements
    - Adding Variables and Handlers
    - Using Tags for Task Selection
- Step 7: Cleanup
- Key Takeaways

Introduction: Mastering DevOps with Mini Projects

DevOps is a transformative methodology that bridges the gap between development and operations teams, emphasizing automation, collaboration, and efficiency. With the growing demand for DevOps professionals, hands-on experience has become crucial for mastering its principles and tools. This document provides a practical roadmap to learning DevOps through **five essential mini-projects**, each focusing on critical areas of DevOps practices.

These projects are carefully curated to help beginners and intermediate learners build foundational skills while gaining exposure to real-world scenarios. They span key DevOps domains such as **Continuous Integration/Continuous Deployment (CI/CD)**, **Infrastructure as Code (IaC)**, **containerization**, **orchestration**, and **configuration management**. By completing these projects, you'll not only gain technical expertise but also develop an understanding of best practices that are essential for a successful DevOps career.

What You'll Learn

- How to automate software delivery pipelines using Jenkins.
- Deploy and manage infrastructure programmatically with Terraform.
- Containerize and orchestrate multi-tier applications using Docker Compose and Kubernetes.
- Automate configuration and application deployment with Ansible.
- Practical troubleshooting techniques and DevOps best practices.

Why These Projects?

Each project in this guide is designed to focus on a specific skill or tool within the DevOps ecosystem:

1. **CI/CD Pipelines**: Automating the build-test-deploy cycle for faster and more reliable software delivery.
2. **Infrastructure as Code**: Managing cloud infrastructure with minimal manual intervention, enhancing scalability and reproducibility.

3. **Containerization**: Building lightweight, portable applications and managing dependencies effectively.
4. **Orchestration**: Streamlining the deployment of complex, multi-container applications.
5. **Configuration Management**: Simplifying the management of systems and ensuring consistency across environments.

Who Should Use This Guide?

- **Beginners**: New to DevOps and eager to build a portfolio of hands-on projects.
- **Intermediate Learners**: Looking to enhance their knowledge and gain practical experience with real-world tools.
- **Job Seekers**: Aspiring DevOps engineers aiming to demonstrate their skills in interviews.
- **IT Professionals**: Developers, system administrators, and cloud practitioners who want to integrate DevOps practices into their workflows.

How to Use This Guide

Each project is broken down into step-by-step instructions, starting with prerequisites and progressing through implementation, testing, and enhancement. By following the instructions, you'll not only complete functional projects but also understand the underlying concepts and tools. Additionally, the guide emphasizes **best practices**, ensuring that your skills are production-ready.

Start Your DevOps Journey

This compilation of projects equips you with the knowledge and confidence to tackle real-world challenges in DevOps. Whether you're automating deployments, managing infrastructure, or orchestrating containers, these hands-on experiences will provide the foundation you need to thrive in a DevOps role. Dive into the projects, experiment, and master the art of delivering software with speed and reliability!

Project 1: Automated Deployment Pipeline with CI/CD Using Jenkins

Overview

A key aspect of DevOps is automating the software delivery process to make it faster, reliable, and error-free. This project focuses on building a Continuous Integration/Continuous Deployment (CI/CD) pipeline using Jenkins. You will automate tasks such as pulling code from a repository, building and testing the application, and deploying it to a server or containerized environment.

Step-by-Step Implementation

Step 1: Prerequisites

Before diving into the project, set up the following tools and systems:

1. **Operating System**: Use a Linux-based system (Ubuntu/Debian recommended).
2. **Docker**: Ensure Docker is installed for running Jenkins and containerizing applications. Install Docker using:

```
sudo apt update
sudo apt install docker.io -y
```

3. **Jenkins**: Install Jenkins either on a VM, bare metal, or using Docker. To install via Docker:

```
docker run -d --name jenkins -p 8080:8080 -p 50000:50000 jenkins/jenkins:lts
```

Access Jenkins at http://<server-ip>:8080.

4. **Git**: Install Git for version control:

```
sudo apt install git -y
```

5. **Sample Application**: Prepare a simple application (e.g., Node.js, Python, or Java). Create a repository for it on GitHub.

Step 2: Setting Up Jenkins

1. **Access Jenkins**:
    - After installation, unlock Jenkins using the initial admin password:

```
sudo cat /var/jenkins_home/secrets/initialAdminPassword
```

- Complete the setup wizard and install the recommended plugins.

2. **Configure Plugins**:
   - Install essential plugins such as Git, Pipeline, NodeJS, or others relevant to your application's technology stack.
   - Go to Manage Jenkins > Manage Plugins, search for the plugins, and install them.
3. **Set Up Global Tools**:
   - Configure global tool installations under Manage Jenkins > Global Tool Configuration (e.g., JDK, Maven, Node.js).

Step 3: Preparing the Application

1. **Create a GitHub Repository**:
   - Push your application to a GitHub repository:

```
mkdir sample-app
cd sample-app
echo "print('Hello, DevOps!')" > app.py
git init
git add .
git commit -m "Initial commit"
git remote add origin <your-repo-url>
git push -u origin main
```

2. **Define a Basic Application**:
   - If using Python, create a requirements.txt file for dependencies. For Node.js, add a package.json file.

Step 4: Creating a Jenkins Pipeline

1. **Create a New Pipeline**:
   - Open Jenkins, click New Item, select Pipeline, and name it CI/CD Pipeline.

2. **Configure the Pipeline**:
    - In the pipeline script, define the stages for building, testing, and deploying the application:

```
pipeline {
  agent any
  stages {
    stage('Clone Repository') {
      steps {
        git branch: 'main', url: 'https://github.com/<username>/<repo>.git'
      }
    }
    stage('Build') {
      steps {
        sh 'echo "Building the application..."'
      }
    }
    stage('Test') {
      steps {
        sh 'echo "Running tests..."'
      }
    }
    stage('Deploy') {
      steps {
        sh 'echo "Deploying the application..."'
      }
    }
  }
```

```
}
```

Step 5: Enhancing the Pipeline

1. **Add Webhooks**:
    - In GitHub, configure a webhook to trigger Jenkins on code changes:
        - Go to your repository settings > Webhooks > Add webhook.
        - Use the URL http://<jenkins-ip>/github-webhook/.
2. **Integrate Docker**:
    - Update the pipeline to build and push a Docker image:

```
stage('Docker Build') {
  steps {
    sh 'docker build -t <username>/<image>:latest .'
    sh 'docker login -u <username> -p <password>'
    sh 'docker push <username>/<image>:latest'
  }
}
```

3. **Deploy to a Container**:
    - Use Docker to run the application:

```
stage('Deploy') {
  steps {
    sh 'docker run -d -p 5000:5000 <username>/<image>:latest'
  }
}
```

Step 6: Running and Monitoring the Pipeline

1. **Trigger the Pipeline**:

- Commit changes to the repository to initiate the pipeline. If webhooks are set up, this will automatically start the build process in Jenkins.

2. **Monitor Pipeline Logs**:
   - In Jenkins, monitor the logs for each stage to ensure the process runs smoothly.

Step 7: Troubleshooting

1. **Common Errors**:
   - **Permission Denied**: Ensure Jenkins has access to Docker commands. Add the Jenkins user to the Docker group:

```
sudo usermod -aG docker jenkins
```

   - **Pipeline Fails**: Check the console output for errors in build, test, or deploy stages.

2. **Retry**:
   - Fix issues in the code or pipeline script and re-trigger the pipeline.

Step 8: Best Practices

1. **Use Parameterized Builds**:
   - Add parameters for environment (e.g., Dev, QA, Prod) to customize the pipeline.
2. **Secure Secrets**:
   - Use Jenkins credentials store to manage sensitive data like DockerHub passwords.
3. **Integrate Testing Tools**:
   - Add automated testing tools like Selenium or JUnit for comprehensive testing.

Key Takeaways

1. **Automation Mastery**: Learn to automate the build-test-deploy cycle using Jenkins.

2. **Docker Skills**: Understand how to containerize applications and manage Docker images.
3. **Pipeline Optimization**: Get hands-on experience with enhancing pipelines using webhooks, environment parameters, and secure credentials.
4. **Debugging Proficiency**: Learn to identify and resolve errors in CI/CD processes.

By completing this project, you'll build foundational skills in DevOps and gain practical experience in implementing CI/CD pipelines, which are essential for modern software delivery practices.

Project 2: Infrastructure as Code (IaC) with Terraform

Overview

Infrastructure as Code (IaC) simplifies the provisioning and management of cloud infrastructure. In this project, you will use Terraform to deploy an AWS infrastructure that includes a Virtual Private Cloud (VPC), subnets, an EC2 instance, a security group, and other essential components.

Step-by-Step Implementation

Step 1: Prerequisites

Ensure the following before starting:

1. **Terraform Installed**: Install Terraform from the official site or via your package manager.

```
sudo apt update
sudo apt install -y terraform
```

2. **AWS Account**: Create an AWS account if you don't have one.
3. **IAM User**: Set up an IAM user with programmatic access and attach a policy like AdministratorAccess.
4. **AWS CLI**: Install and configure the AWS CLI with the IAM user credentials:

```
aws configure
```

5. **Code Editor**: Use VS Code, IntelliJ IDEA, or your preferred editor for writing Terraform code.

Step 2: Create the Terraform Configuration

1. **Set Up the Project Directory**:
    - Create a directory for the Terraform project:

```
mkdir terraform-project
cd terraform-project
```

2. **Define the Main Configuration File**:

- Create a file named main.tf and add the provider configuration:

```
provider "aws" {
  region = "us-east-1"
}
```

3. **Create a VPC**:
   - Define a resource block for the VPC in main.tf:

```
resource "aws_vpc" "main_vpc" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = {
    Name = "MainVPC"
  }
}
```

4. **Add Subnets**:
   - Define public and private subnets:

```
resource "aws_subnet" "public_subnet" {
  vpc_id                  = aws_vpc.main_vpc.id
  cidr_block              = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone       = "us-east-1a"
  tags = {
    Name = "PublicSubnet"
  }
}
```

```
resource "aws_subnet" "private_subnet" {
  vpc_id            = aws_vpc.main_vpc.id
  cidr_block        = "10.0.2.0/24"
  availability_zone = "us-east-1a"
  tags = {
    Name = "PrivateSubnet"
  }
}
```

5. **Add an Internet Gateway and Route Table**:

```
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main_vpc.id
  tags = {
    Name = "InternetGateway"
  }
}

resource "aws_route_table" "public_rt" {
  vpc_id = aws_vpc.main_vpc.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }

  tags = {
    Name = "PublicRouteTable"
```

```
  }
}

resource "aws_route_table_association" "public_rt_assoc" {
  subnet_id      = aws_subnet.public_subnet.id
  route_table_id = aws_route_table.public_rt.id
}
```

6. **Launch an EC2 Instance**:
    - Add a security group and an EC2 instance to main.tf:

```
resource "aws_security_group" "web_sg" {
  name_prefix = "web-sg"
  vpc_id      = aws_vpc.main_vpc.id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

```
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "WebServerSG"
  }
}

resource "aws_instance" "web_server" {
  ami           = "ami-0c02fb55956c7d316"  # Amazon Linux 2 AMI
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public_subnet.id
  security_groups = [aws_security_group.web_sg.name]

  tags = {
    Name = "WebServer"
  }
}
```

Step 3: Initialize and Apply Terraform

1. **Initialize Terraform**:

```
terraform init
```

2. **Validate the Configuration**:

```
terraform validate
```

3. **Plan the Deployment**:

```
terraform plan
```

This command shows the resources that Terraform will create.

4. **Apply the Configuration**:

```
terraform apply
```

Confirm the prompt with yes. Terraform will provision the infrastructure.

Step 4: Verify the Deployment

1. **Access the AWS Console**:
    - Verify that the VPC, subnets, internet gateway, and EC2 instance are created in the AWS Management Console.
2. **SSH into the EC2 Instance**:
    - Use the private key corresponding to your key pair to access the EC2 instance:

```
ssh -i <private-key.pem> ec2-user@<public-ip>
```

3. **Test Connectivity**:
    - Install and configure a basic web server (e.g., Apache or Nginx):

```
sudo yum install httpd -y
sudo systemctl start httpd
sudo systemctl enable httpd
echo "Hello, Terraform!" | sudo tee /var/www/html/index.html
```

    - Access the web server in your browser using the instance's public IP.

Step 5: Destroy the Resources

To clean up and avoid unnecessary costs, destroy the resources:

```
terraform destroy
```

Key Takeaways

1. **IaC Fundamentals**: Learn to define, provision, and manage infrastructure using Terraform.
2. **AWS Resources**: Understand the basics of VPCs, subnets, internet gateways, security groups, and EC2 instances.
3. **Automation**: Automate infrastructure provisioning to minimize manual efforts and reduce errors.
4. **Reusability**: Build reusable and modular Terraform code for different environments.

This project is a hands-on introduction to Terraform and AWS infrastructure, laying a strong foundation for more advanced DevOps automation and cloud management practices.

Project 3: Containerized Application Deployment with Docker Compose

Overview

Docker Compose simplifies the deployment of multi-container applications. In this project, you will containerize a simple web application and its database, then use Docker Compose to define and manage the deployment. This approach is commonly used in real-world scenarios for local development and testing.

Step-by-Step Implementation

Step 1: Prerequisites

Ensure you have the following:

1. **Docker Installed**: Install Docker on your system. For Linux:

```
sudo apt update
sudo apt install docker.io -y
```

2. **Docker Compose Installed**: Install Docker Compose to manage multi-container applications:

```
sudo curl -L "https://github.com/docker/compose/releases/download/$(curl -s https://api.github.com/repos/docker/compose/releases/latest | grep tag_name | cut -d '"' -f 4)/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

3. **Sample Application**: Use a web application (e.g., Python Flask or Node.js) that connects to a database like MySQL.

Step 2: Prepare the Application

1. **Create a Project Directory**:

```
mkdir docker-compose-project
cd docker-compose-project
```

2. **Write the Web Application Code**:
    - Create a app.py file (for a Python Flask app):

```
from flask import Flask
import os
import mysql.connector

app = Flask(__name__)

@app.route('/')
def index():
    try:
        db_connection = mysql.connector.connect(
            host=os.getenv('DB_HOST'),
            user=os.getenv('DB_USER'),
            password=os.getenv('DB_PASSWORD'),
            database=os.getenv('DB_NAME')
        )
        cursor = db_connection.cursor()
        cursor.execute("SELECT 'Hello, Docker Compose!' AS message")
        message = cursor.fetchone()[0]
        return f"<h1>{message}</h1>"
    except Exception as e:
        return f"<h1>Error: {e}</h1>"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

3. **Define Dependencies**:
    - Create a requirements.txt file:

```
flask
mysql-connector-python
```

4. **Prepare a Database Initialization Script**:
    - Create a db-init.sql file:

```
CREATE DATABASE app_db;
CREATE USER 'app_user'@'%' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON app_db.* TO 'app_user'@'%';
FLUSH PRIVILEGES;

USE app_db;
CREATE TABLE messages (id INT AUTO_INCREMENT PRIMARY KEY, message TEXT);
INSERT INTO messages (message) VALUES ('Hello, Docker Compose!');
```

Step 3: Write the Dockerfiles

1. **Web Application Dockerfile**:
    - Create a Dockerfile for the Flask app:

```
FROM python:3.9-slim

WORKDIR /app

COPY . /app

RUN pip install -r requirements.txt
```

```
CMD ["python", "app.py"]
```

2. **Database Dockerfile (Optional)**:
    - You can use the official MySQL image, so no custom Dockerfile is needed.

Step 4: Create the docker-compose.yml File

Define the services for the web application and the database:

```
version: "3.8"

services:

  app:

    build:

      context: .

    ports:

      - "5000:5000"

    environment:

      DB_HOST: db

      DB_USER: app_user

      DB_PASSWORD: password

      DB_NAME: app_db

    depends_on:

      - db


  db:

    image: mysql:8.0

    environment:

      MYSQL_ROOT_PASSWORD: rootpassword

    volumes:
```

```
    - db_data:/var/lib/mysql
    - ./db-init.sql:/docker-entrypoint-initdb.d/init.sql
  ports:
    - "3306:3306"

volumes:
 db_data:
```

Step 5: Build and Run the Application

1. **Build the Docker Images**:

```
docker-compose build
```

2. **Run the Containers**:

```
docker-compose up
```

This command starts the app and db services.

3. **Access the Application**:
    - Open your browser and go to http://localhost:5000. You should see the message "Hello, Docker Compose!"

Step 6: Test and Debug

1. **Verify Database Connection**:
    - Use the MySQL CLI to check the database:

```
docker exec -it <db-container-id> mysql -u root -p
```

    - Query the app_db database to verify data.
2. **Check Logs**:
    - View the logs for the application and database:

```
docker-compose logs app
```

```
docker-compose logs db
```

Step 7: Enhance the Application

1. **Add Environment Variables**:
    - Create a .env file for sensitive data:

```
DB_HOST=db
DB_USER=app_user
DB_PASSWORD=password
DB_NAME=app_db
```

    - Update docker-compose.yml to use the .env file:

```
env_file:
 - .env
```

2. **Add Persistent Data**:
    - Ensure the database data persists even after restarting the containers using Docker volumes.
3. **Scale the Application**:
    - Modify docker-compose.yml to scale the app service:

```
docker-compose up --scale app=3
```

Step 8: Stop and Cleanup

1. **Stop the Containers**:

```
docker-compose down
```

2. **Remove Volumes (Optional)**:
    - To delete persistent data, use:

```
docker-compose down -v
```

Key Takeaways

1. **Multi-Container Applications**: Learn how to deploy applications with multiple dependent services.
2. **Networking in Docker**: Understand how Docker Compose links containers and manages networking.
3. **Environment Variables**: Practice using environment variables to pass sensitive data securely.
4. **Persistent Storage**: Use volumes to ensure data persists across container restarts.

This project introduces you to containerized deployments and Docker Compose's capabilities, preparing you for more complex use cases in real-world scenarios.

Project 4: Kubernetes Deployment for a Multi-Tier Application

Overview

Kubernetes (K8s) is a powerful platform for automating the deployment, scaling, and management of containerized applications. In this project, you will deploy a multi-tier application (e.g., a web application with a backend database) on Kubernetes using deployments, services, and persistent volumes.

Step-by-Step Implementation

Step 1: Prerequisites

Ensure the following tools are installed and configured:

1. **Kubernetes Cluster**: Use a local cluster like Minikube or a cloud-managed service (e.g., AWS EKS, Azure AKS, or Google GKE).
2. **kubectl**: The Kubernetes CLI tool for managing resources.
3. **Docker**: To build container images for your application.
4. **YAML File Editor**: Use a code editor like VS Code with YAML support.

Step 2: Application Overview

We will deploy a sample multi-tier application:

- **Frontend**: A Node.js or Python Flask web application.
- **Backend Database**: MySQL.

Step 3: Build the Application

1. **Prepare the Web Application**:
    - Create a simple Node.js application (app.js):

```
const express = require('express');
const mysql = require('mysql');
```

```
const app = express();
const port = 3000;

const db = mysql.createConnection({
   host: process.env.DB_HOST,
   user: process.env.DB_USER,
   password: process.env.DB_PASSWORD,
   database: process.env.DB_NAME,
});

app.get('/', (req, res) => {
  db.query('SELECT "Welcome to Kubernetes Deployment!" AS message', (err, result) => {
      if (err) {
          res.send('Database connection error: ' + err);
      } else {
          res.send(result[0].message);
      }
   });
});

app.listen(port, () => console.log(`App listening on port ${port}`));
```

- Add a package.json file:

```
{
  "name": "k8s-demo",
  "version": "1.0.0",
```

```
 "main": "app.js",
 "dependencies": {
  "express": "^4.17.1",
  "mysql": "^2.18.1"
 }
}
```

- Create a Dockerfile for the web application:

```
FROM node:14
WORKDIR /app
COPY . /app
RUN npm install
CMD ["node", "app.js"]
```

2. **Database Initialization**:
   - Create an SQL script (init.sql) for database initialization:

```
sql
CopyEdit
CREATE DATABASE app_db;
CREATE USER 'app_user'@'%' IDENTIFIED BY 'password';
GRANT ALL PRIVILEGES ON app_db.* TO 'app_user'@'%';
FLUSH PRIVILEGES;

USE app_db;
CREATE TABLE messages (id INT AUTO_INCREMENT PRIMARY KEY, message TEXT);
INSERT INTO messages (message) VALUES ('Welcome to Kubernetes Deployment!');
```

Step 4: Write Kubernetes Manifests

1. **Deployment for the Web Application**:
    - Create web-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: web-deployment
spec:
 replicas: 2
 selector:
  matchLabels:
   app: web
 template:
  metadata:
   labels:
    app: web
  spec:
   containers:
   - name: web
     image: <your-dockerhub-username>/k8s-demo:latest
     ports:
     - containerPort: 3000
     env:
     - name: DB_HOST
       value: "mysql-service"
     - name: DB_USER
```

```
        value: "app_user"
      - name: DB_PASSWORD
        value: "password"
      - name: DB_NAME
        value: "app_db"
```

2. **Service for the Web Application**:
    - Create web-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
 name: web-service
spec:
 selector:
  app: web
 ports:
 - protocol: TCP
  port: 80
  targetPort: 3000
 type: NodePort
```

3. **Deployment for MySQL Database**:
    - Create mysql-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: mysql-deployment
spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:5.7
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: "rootpassword"
```

4. **Service for MySQL Database**:
    - Create mysql-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
spec:
  selector:
    app: mysql
```

```
  ports:
  - protocol: TCP
    port: 3306
    targetPort: 3306
  clusterIP: None
```

Step 5: Deploy the Application

1. **Apply the Kubernetes Manifests**:

```
kubectl apply -f mysql-deployment.yaml
kubectl apply -f mysql-service.yaml
kubectl apply -f web-deployment.yaml
kubectl apply -f web-service.yaml
```

2. **Verify the Deployments**:
    - Check pods:

```
kubectl get pods
```

    - Check services:

```
kubectl get svc
```

3. **Access the Application**:
    - Retrieve the NodePort of the web-service:

```
kubectl get svc web-service
```

    - Open your browser and navigate to http://<node-ip>:<node-port> to see the application.

Step 6: Enhance and Test

1. **Scaling**:
    - Scale the web application:

```
kubectl scale deployment web-deployment --replicas=4
```

- Verify:

```
kubectl get pods
```

2. **Persistence**:
    - Use PersistentVolumes and PersistentVolumeClaims for database storage.
3. **Health Checks**:
    - Add liveness and readiness probes to the web-deployment.yaml:

```
livenessProbe:
  httpGet:
    path: /
    port: 3000
  initialDelaySeconds: 3
  periodSeconds: 5
readinessProbe:
  httpGet:
    path: /
    port: 3000
  initialDelaySeconds: 3
  periodSeconds: 5
```

Step 7: Cleanup

1. **Delete the Resources**:

```
kubectl delete -f .
```

Key Takeaways

1. **Multi-Tier Architecture**: Understand how to deploy a web app and database on Kubernetes.

2. **Kubernetes Basics**: Learn about deployments, services, scaling, and health checks.
3. **Persistence**: Practice managing data with PersistentVolumes.
4. **Application Scaling**: Gain insights into Kubernetes' scalability features.

This project will solidify your Kubernetes knowledge and prepare you for deploying more complex, production-grade applications.

Project 5: Automated Configuration Management with Ansible

Overview

Ansible is a popular tool for automating configuration management and application deployment. In this project, you will use Ansible to automate the installation of a LAMP stack (Linux, Apache, MySQL, PHP) on a server and deploy a sample PHP application.

Step-by-Step Implementation

Step 1: Prerequisites

Before starting, ensure you have the following:

1. **Ansible Installed**: Install Ansible on your local machine:

```
sudo apt update
sudo apt install ansible -y
```

2. **Target Server**: A Linux server (e.g., Ubuntu or CentOS) where Ansible can deploy the configurations. Ensure SSH access is set up.
3. **Inventory File**: A file listing the IP addresses or hostnames of your target servers.

Step 2: Set Up the Ansible Inventory

1. **Create an Inventory File**:
    - Create a file named inventory.ini:

```
[webservers]
192.168.1.100 ansible_user=ubuntu
ansible_ssh_private_key_file=~/.ssh/id_rsa
```

2. Replace 192.168.1.100 with your server's IP address and ensure the correct path to your SSH private key.

Step 3: Write the Ansible Playbook

1. **Create a Playbook File**:
    - Create a file named lamp_playbook.yml:

```
---
- name: Setup LAMP Stack
  hosts: webservers
  become: yes
  tasks:
    # Task 1: Update system packages
    - name: Update and upgrade apt packages
      apt:
        update_cache: yes
        upgrade: dist

    # Task 2: Install Apache
    - name: Install Apache
      apt:
        name: apache2
        state: present

    # Task 3: Start and enable Apache
    - name: Ensure Apache is running
      service:
        name: apache2
        state: started
        enabled: true
```

```
  # Task 4: Install MySQL
  - name: Install MySQL server
    apt:
      name: mysql-server
      state: present

  # Task 5: Secure MySQL installation
  - name: Run MySQL secure installation
    mysql_secure_installation:
      login_user: root
      login_password: ''
      new_password: 'rootpassword'
      remove_anonymous_users: yes
      disallow_root_login_remotely: yes
      remove_test_database: yes
      state: present

  # Task 6: Install PHP
  - name: Install PHP and required modules
    apt:
      name:
        - php
        - php-mysql
      state: present
```

```
  # Task 7: Deploy Sample PHP Application
   - name: Create PHP file
     copy:
       dest: /var/www/html/index.php
       content: |
         <?php
         $conn = new mysqli("localhost", "root", "rootpassword");
         if ($conn->connect_error) {
           die("Connection failed: " . $conn->connect_error);
         }
         echo "Connected successfully. Welcome to Ansible-LAMP!";
         ?>
```

Step 4: Run the Playbook

1. **Execute the Playbook**:
    - Run the following command:

```
ansible-playbook -i inventory.ini lamp_playbook.yml
```

2. **Monitor the Output**:
    - Ansible will perform the tasks defined in the playbook. Verify that all tasks complete successfully.

Step 5: Verify the Deployment

1. **Access the Web Server**:
    - Open a browser and navigate to http://<server-ip> to see the deployed PHP application.
2. **Check Apache Status**:
    - SSH into the server and run:

```
sudo systemctl status apache2
```

3. **Verify MySQL**:
    - Log in to the MySQL server to confirm the secure installation:

```
mysql -u root -p
```

Step 6: Enhance the Playbook

1. **Add Variables**:
    - Use variables to make the playbook reusable for different environments:

```
vars:
  mysql_root_password: "rootpassword"
  php_test_file: |
    <?php
    $conn = new mysqli("localhost", "root", "{{ mysql_root_password }}");
    if ($conn->connect_error) {
      die("Connection failed: " . $conn->connect_error);
    }
    echo "Connected successfully. Welcome to Ansible-LAMP!";
    ?>
```

2. **Add Handlers**:
    - Use handlers to restart Apache only when necessary:

```
handlers:
  - name: Restart Apache
    service:
      name: apache2
      state: restarted
```

```
tasks:
 - name: Copy PHP file
   copy:
     dest: /var/www/html/index.php
     content: "{{ php_test_file }}"
   notify: Restart Apache
```

3. **Add Tags**:
    - Add tags to execute specific tasks:

```
tasks:
 - name: Install Apache
   apt:
     name: apache2
     state: present
   tags: apache
```

    - Run specific tags:

```
ansible-playbook -i inventory.ini lamp_playbook.yml --tags "apache"
```

Step 7: Clean Up

1. **Remove the Installed Packages**:
    - Add a cleanup task in the playbook to uninstall the LAMP stack:

```
- name: Remove LAMP stack
  apt:
    name:
      - apache2
      - mysql-server
      - php
```

```
  state: absent
```

2. **Delete Files**:
    - Remove application files:

```
- name: Remove PHP file
  file:
    path: /var/www/html/index.php
    state: absent
```

Key Takeaways

1. **Automation with Ansible**: Learn how to automate the installation and configuration of software on remote servers.
2. **Reusability**: Create reusable playbooks with variables, handlers, and tags.
3. **Configuration Management**: Master deploying multi-component stacks with Ansible.
4. **Troubleshooting**: Gain experience in debugging Ansible playbooks and logs.

This project builds your understanding of configuration management and prepares you for automating larger-scale deployments in a professional setting.

Conclusion

The **five essential DevOps mini-projects** covered in this guide are carefully designed to provide practical, hands-on experience with the most important tools and practices in the DevOps ecosystem. By working on these projects, you will master the critical areas of DevOps, such as **automation with Jenkins**, **infrastructure management with Terraform**, **containerization with Docker Compose**, **orchestration with Kubernetes**, and **configuration management with Ansible**.

These projects are not just exercises—they reflect real-world scenarios and industry practices that are widely adopted by organizations globally. By completing these, you will build a strong portfolio that demonstrates your proficiency in modern DevOps methodologies, which you can confidently showcase in your **resume**. Employers value practical experience, and these projects will give you an edge in job applications and interviews.

For job seekers aspiring to roles like **DevOps Engineer**, **Cloud Engineer**, or **Site Reliability Engineer (SRE)**, these projects serve as proof of your technical expertise and problem-solving abilities. They highlight your ability to automate, optimize, and manage workflows effectively—skills that are essential for thriving in a DevOps role.

Start implementing these projects today to enhance your resume, gain confidence in DevOps tools, and pave your way to a successful career in the field of DevOps. Your journey to becoming a skilled DevOps professional begins here!