Ruby on Rails Application Deployment

29 November 2024 19:10

To successfully complete this DevOps project, follow these steps:

Step 1: Docker

1. **Choose a Ruby on Rails Application:** You can either use one of the provided example applications or create your own simple Rails app. The key is to ensure it uses a PostgreSQL database.
2. **Create a Dockerfile:** Write a Dockerfile for your Rails application. This file will specify the base image (Ruby), install dependencies, set up the application code, and configure the application to run within a Docker container.
3. **Create a Dockerfile for PostgreSQL:** Write a separate Dockerfile for the PostgreSQL database. This file will specify the PostgreSQL base image, set up any necessary configurations, and ensure the database is ready to accept connections.
4. **Build and Test:** Build the Docker images for both the application and the database. Run the containers and test that the application can connect to the database and function correctly.

Step 2: Kubernetes

- **Push Images to Docker Hub (or your registry):** If you haven't already, push your my-rails-app and my-postgres-db images to Docker Hub or your preferred container registry. This will make them accessible to your Kubernetes cluster.
- **Create Kubernetes Manifests:**
  - **Deployment (my-rails-app.yaml):** Create a deployment for your Rails app. Make sure to update the image field with the full path to your Rails image on Docker Hub (e.g., <your_dockerhub_username>/my-rails-app).
  - **Service (my-rails-app-service.yaml):** Create a service to expose your Rails app. You can use a LoadBalancer type if you want to access it externally.
  - **StatefulSet (my-postgres-db.yaml):** Create a StatefulSet for your PostgreSQL database. Update the image field with the full path to your PostgreSQL image.
  - **Ingress (ingress.yaml):** If you want to use an Ingress controller, create an Ingress resource to route traffic to your Rails app service.
- **Set up a Kubernetes Cluster:** Use Minikube, K3d, or any other local Kubernetes cluster.
- **Install Ingress Controller (optional):** If you're using an Ingress, install the Ingress controller (e.g., Nginx Ingress) in your cluster.
- **Apply the Manifests:** Use kubectl apply -f to apply your Kubernetes manifests to the cluster.
- **Verify Deployment:** Use kubectl get pods, kubectl get services, and other kubectl commands to verify that your application is running correctly in the cluster.

1. **Choose a Local Cluster:** Install a local Kubernetes cluster provider like Minikube or K3d.
2. **Write YAML Files:** Create Kubernetes YAML files for your application and database.
   - **Application:** Define a Deployment to manage the Rails application pods.
   - **Database:** Use a StatefulSet to manage the PostgreSQL database pod. This ensures data persistence and stable network identities for the database.
3. **Ingress (Optional):** If you want to expose your application externally, set up an Ingress controller (like NGINX Ingress Controller) and create an Ingress resource to route traffic to your application.
4. **Service Mesh (Optional):** If you need advanced traffic management and observability, consider using a service mesh like Istio or Linkerd.
5. **Deploy and Test:** Apply the YAML files to your Kubernetes cluster. Test that the application is running correctly and accessible.

Step 3: ArgoCD

- **Create a Private GitHub Repository:** Create a private repository on GitHub and commit your Kubernetes manifests to it.
- **Install ArgoCD:** Follow the ArgoCD documentation to install it in your cluster.

- **Configure ArgoCD:**
  - **argocd-cm.yaml:** Create a ConfigMap for ArgoCD with the necessary settings.
  - **argocd-rbac-cm.yaml:** Create a ConfigMap for ArgoCD RBAC configuration.
  - **GitHub Secret:** Create a Kubernetes secret with your GitHub username and a personal access token (PAT) to allow ArgoCD to access your private repo.
  - **application.yaml:** Create an ArgoCD Application resource to define your application and link it to your GitHub repository.
- **Deploy ArgoCD Configuration:** Apply the ArgoCD configuration files to your cluster.
- **Access ArgoCD UI:** Use kubectl port-forward to access the ArgoCD UI and monitor your application deployment.

1. **Create a Private GitHub Repository:** Set up a private repository to store your Kubernetes manifests, Dockerfile, GitOps configurations, and pipeline configuration files.
2. **Install ArgoCD:** Deploy ArgoCD to your Kubernetes cluster.
3. **Configure ArgoCD:**
   - Create an `application.yaml` file to define the application you want ArgoCD to manage.
   - Set up ArgoCD ConfigMaps (`argocd-cm` and `argocd-rbac-cm`) for customization and role-based access control.
   - Configure ArgoCD to connect to your private GitHub repository.
4. **GitOps Workflow:** Push your Kubernetes manifests to the GitHub repository. ArgoCD will automatically detect the changes and deploy them to your cluster.

Step 4: Tekton

1. **Install Tekton:** Install Tekton Pipelines and the Tekton Dashboard in your cluster.
2. **Create Tekton Resources:**
   - **pipeline.yaml:** Define your Tekton pipeline to fetch the source code, build the image, push it to Docker Hub, and update your Kubernetes deployment.
   - **task-kubectl-apply.yaml:** Create a Tekton task to update your Kubernetes deployment with the new image.
   - **Other Tasks (if needed):** Define any other necessary tasks (e.g., git-clone, kaniko) if they are not available in your Tekton installation.
3. **Run the Pipeline:** Use the Tekton Dashboard to create a PipelineRun, provide the required parameters, and start the pipeline.

Remember to:

1. **Install Tekton:** Set up Tekton Pipelines and the Tekton Dashboard in your Kubernetes cluster.
2. **Create a Pipeline:** Define a Tekton Pipeline that performs the following tasks:
   - Download the source code from the public fork of your sample project.
   - Build a Docker image using the Dockerfile.
   - Push the image to Docker Hub.
3. **Test the Pipeline:** Manually trigger the pipeline from the Tekton Dashboard and verify that it successfully builds and pushes the image.

Submission

1. **Prepare the ZIP File:** Create a ZIP file containing all the required configuration files (Kubernetes manifests, Dockerfile, GitOps configurations, pipeline configurations). Remember to exclude any sensitive information like SSH keys or deployment keys.
2. **Record a Video Demo:** Create a video demonstrating the functionality of your application, the deployment process using ArgoCD, and the execution of the Tekton pipeline.
3. **Submit:** Submit the ZIP file, video demo, and any additional relevant information through the provided submission link.

Remember:

- **Thorough Testing:** Test each step of the process thoroughly to ensure everything works as expected.
- **Documentation:** Document your steps and decisions clearly. This will help you during the demo and troubleshooting.
- **Clarity:** Make sure your video demo is clear and concise, highlighting the key aspects of your implementation.
- **Reach Out:** If you have any questions, don't hesitate to contact the provided email address for clarification.

https://docs.google.com/document/d/15S3BIdAd057s88D310X2UjNnVGozZicrGNHB7UV3uPo/edit?

tab=t.0

To Convert whole repository into the one code.

```
const fs = require('fs');
const path = require('path');
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
const AUTO_INCLUDE = process.argv.includes('--auto-include');
async function promptUser(question) {
  if (AUTO_INCLUDE) {
    console.log(`${question} y`);
    return 'y';
  }

  return new Promise((resolve) => {
    rl.question(question, (answer) => {
      resolve(answer);
    });
  });
}
async function selectFiles(currentDir, excludePatterns) {
  const selectedFiles = [];
  const files = await fs.promises.readdir(currentDir);
  for (const file of files) {
    const filePath = path.join(currentDir, file);
    const stats = await fs.promises.stat(filePath);
    if (stats.isDirectory()) {
      if (!excludePatterns.includes(file)) {
        const includeFolder = await promptUser(`Include folder '${file}'?
(y/n) `);
        if (includeFolder.toLowerCase() === 'y') {
          const subFiles = await selectFiles(filePath, excludePatterns);
          selectedFiles.push(...subFiles);
        }
      }
    } else {
      const includeFile = await promptUser(`Include file '${file}'? (y/n) `);
      if (includeFile.toLowerCase() === 'y') {
        selectedFiles.push(filePath);
      }
    }
  }
  return selectedFiles;
}
async function mergeFiles(selectedFiles, outputFilePath) {
  let mergedContent = '';
  for (const filePath of selectedFiles) {
    try {
      const fileContent = await fs.promises.readFile(filePath, 'utf-8');
      const sectionHeader = `\n${filePath.toUpperCase()} CODE IS BELOW\n`;
      mergedContent += sectionHeader + fileContent + '\n';
    } catch (error) {
      console.error(`Error reading file ${filePath}: ${error.message}`);
    }
  }
  await fs.promises.writeFile(outputFilePath, mergedContent);
}
async function createOutputDirectory(outputDirPath) {
```

```
  try {
    await fs.promises.access(outputDirPath);
  } catch (error) {
    await fs.promises.mkdir(outputDirPath);
  }
}
function getTimestampedFileName() {
  const timestamp = new Date().toISOString().replace(/:/g, '-');
  return `merged-repo-${timestamp}.txt`;
}
async function main() {
  const currentDir = process.cwd();
  console.log('Select files and folders to include in the merge:');
  const excludePatterns = ['node_modules', '.git', '.vscode', '.idea']; // Add
more patterns if needed
  const selectedFiles = await selectFiles(currentDir, excludePatterns);
  const outputDirName = 'llm_text_transcripts';
  const outputDirPath = path.join(currentDir, outputDirName);
  await createOutputDirectory(outputDirPath);
  const outputFileName = getTimestampedFileName();
  const outputFilePath = path.join(outputDirPath, outputFileName);
  await mergeFiles(selectedFiles, outputFilePath);
  console.log(`Merged repository saved to: ${outputFilePath}`);
  rl.close();
}
main().catch((error) => {
  console.error('An error occurred:', error);

  rl.close();
});
```

★ Commands to install ruby on Linux

→ sudo apt update
→ sudo apt install build-essential zlib1g-dev libssl-dev libreadline-dev libyaml-dev libsqlite3-dev sqlite3 libxml2-dev libxslt1-dev autoconf libgmp-dev
→ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
→ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc

→ echo 'eval "$(rbenv init -)"' >> ~/.bashrc
→ source ~/.bashrc
→ git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
→ rbenv install -l
→ rbenv global 3.2.2
→ ruby -v
→ gem install rails
→

Now uploading application on ubuntu.

→ git push -f origin main

Now inside on ubuntu linux application repository we will create and docker image of the application.

→ vim Dockerfile

→ FROM ruby:3.1

→ WORKDIR /app

```
COPY Gemfile Gemfile.lock ./
RUN bundle install

COPY . .

RUN bundle exec rails db:create
RUN bundle exec rails db:migrate

EXPOSE 3000

CMD ["rails", "server", "-b", "0.0.0.0"]

vim Dockerfile.postgres

FROM postgres:14

ENV POSTGRES_USER=myuser
ENV POSTGRES_PASSWORD=mypassword
ENV POSTGRES_DB=mydatabase

docker build -t my-rails-app .

docker-compose.yml

version: "3.9"
services:
  web:
    build: .
    ports:
      - "3000:3000"
    depends_on:
      - db
    environment:
      - DATABASE_HOST=db
      - DATABASE_USER=myuser
      - DATABASE_PASSWORD=mypassword
      - DATABASE_NAME=mydatabase
  db:
    image: postgres:14
    environment:
      - POSTGRES_USER=myuser
      - POSTGRES_PASSWORD=mypassword
      - POSTGRES_DB=mydatabase
```

Explanation:

- depends_on: This ensures that the db container starts before the web container.
- environment: These variables provide the database connection details to your Rails app. The DATABASE_HOST is set to db, which is the service name of your PostgreSQL container within the Docker Compose network. This allows the Rails app to connect to the database container.

Clean Up and Retry: Sometimes, cleaning up old containers, images, and networks can help resolve issues. Try these commands:

```
docker-compose down --rmi all --volumes
docker system prune -a
```

So the big challenge came needs to run the postgre container mainually from the image and then assign its ip address to the conif database.yml file in db section so that rails image can be build and run the container while connecting to the postgre database using ip.

```
docker run -d --name my-postgres-db -e POSTGRES_USER=myuser -e
POSTGRES_PASSWORD=mypassword -e POSTGRES_DB=mydatabase my-postgres-db
af24e2ee2956b5b7e650311b9133c2cd853d474816aca3dcd35a043e1c1e6725

docker run -d --name my-rails-app -p 3000:3000 -e DATABASE_HOST=my-postgres-db -e
DATABASE_USER=myuser -e DATABASE_PASSWORD=mypassword -e
DATABASE_NAME=mydatabase my-rails-app
92995ffd7f95fed10781a4fa1302dcf5588378db61afda65ba9b2a3c89525df0

docker ps -a
```

Creating kubernetes Manifests - deploymnets , services, and statefulset.

Deployment (my-rails-app.yaml):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-rails-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-rails-app
  template:
    metadata:
      labels:
        app: my-rails-app
    spec:
      containers:
      - name: my-rails-app
        image: my-rails-app # Use your Docker Hub image later
        ports:
        - containerPort: 3000
        env:
        - name: DATABASE_HOST
          value: my-postgres-db # Service name of your PostgreSQL
        - name: DATABASE_USER
          value: myuser
        - name: DATABASE_PASSWORD
          value: mypassword
        - name: DATABASE_NAME
          value: mydatabase
```

Service (my-rails-app-service.yaml):

```
apiVersion: v1
kind: Service
metadata:
  name: my-rails-app-service
```

```
spec:
  selector:
    app: my-rails-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 3000
  type: LoadBalancer
```

StatefulSet (my-postgres-db.yaml):

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-postgres-db
spec:
  serviceName: "my-postgres-db"
  replicas: 1
  selector:
    matchLabels:
      app: my-postgres-db
  template:
    metadata:
      labels:
        app: my-postgres-db
    spec:
      containers:
      - name: my-postgres-db
        image: my-postgres-db # Use your Docker Hub image later
        ports:
        - containerPort: 5432
        env:
        - name: POSTGRES_USER
          value: myuser
        - name: POSTGRES_PASSWORD
          value: mypassword
        - name: POSTGRES_DB
          value: mydatabase 1
        volumeMounts:
        - name: postgres-persistent-storage
          mountPath: /var/lib/postgresql/data
  volumeClaimTemplates:
  - metadata:
      name: postgres-persistent-storage
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
```

ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-rails-app-ingress
```

```
spec:
  rules:
  - host: my-rails-app.local # Replace with your domain/hostname
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-rails-app-service
            port:
              number: 80
```

Now setting up the kubernetes cluster.

```
minikube start
minikube addons enable ingress

kubectl apply -f my-rails-app.yaml
kubectl apply -f my-rails-app-service.yaml
kubectl apply -f my-postgres-db.yaml
kubectl apply -f ingress.yaml

kubectl get service -n ingress-nginx
kubectl get pods -n ingress-nginx
```

Now as we can see in pod rails app not running as we have not pushed the images to the docker hub yet so let's do it.

```
docker login
```

and now tagging both the images.

```
docker tag my-rails-app:latest sansugupta/my-rails-app:latest
docker tag my-postgres-db:latest sansugupta/my-postgres-db:latest
```

Now push the rails images.

```
docker push <your_dockerhub_username>/my-postgres-db:latest
docker push <your_dockerhub_username>/my-rails-app:latest
```

Now pushed all the changes and manifests files into the git repository.

and now we will setup argo cd to connect the repository and deploy the application.

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

ArgoCD Configuration:

Create argocd-cm.yaml:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
```

```
  namespace: argocd
data:
  url: https://kubernetes.default.svc # Replace if necessary
  repo.server: github.com # Or your Git provider
```

Create argocd-rbac-cm.yaml:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-rbac-cm
  namespace: argocd
data:
  policy.csv: |
    g, *, role:admin
```

Create a secret for your GitHub repository access: You'll need to create a Kubernetes secret that stores your GitHub username and a personal access token (PAT) with read access to your repository. This allows ArgoCD to access your private repo.

```
apiVersion: v1
kind: Secret
metadata:
  name: github-repo-secret
  namespace: argocd
stringData:
  username: <your_github_username>
  password: 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
```

Create a argocd-server.yaml

```
apiVersion: apps/v1
kind: Deployment
# ... other parts of the deployment ...
spec:
  template:
    spec:
      containers:
      - name: argocd-server
        # ... other container settings ...
        volumeMounts:
        - name: argocd-cm
          mountPath: /path/to/config/in/container
      volumes:
      - name: argocd-cm
        configMap:
          name: argocd-cm
```

```
kubectl apply -f argocd-cm.yaml
kubectl apply -f argocd-rbac-cm.yaml
kubectl apply -f github-repo-secret.yaml
kubectl apply -f application.yaml
```

Access ArgoCD UI: Use kubectl port-forward service/argocd-server 8080:443 to access the ArgoCD

UI and monitor your application deployment.

kubectl port-forward service/argocd-server -n argocd 8080:443

To get the password.

kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d

RUN BOTH CONATINERS MINIKUBECLUSTER
kubectl port-forward service/argocd-server -n argocd 8080:443

kubectl get pods -n argocd

kubectl delete namespace argocd
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

kubectl port-forward service/argocd-server -n argocd 8080:443

Now we will install tekton pipeline and the tekton dashboard in the cluster.

kubectl apply --filename https://storage.googleapis.com/tekton-releases/pipeline/latest/release.yaml

Tetkon Triggers

kubectl apply --filename \
https://storage.googleapis.com/tekton-releases/triggers/latest/release.yaml
kubectl apply --filename \
https://storage.googleapis.com/tekton-releases/triggers/latest/interceptors.yaml

Tetkon Dashboard

kubectl apply --filename https://storage.googleapis.com/tekton-releases/dashboard/latest/release.yaml

Install the Task

kubectl apply -f https://api.hub.tekton.dev/v1/resource/tekton/task/kaniko/0.6/raw

Create a pipeline.yaml:

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: my-rails-app-pipeline
spec:
  workspaces:
  - name: shared-workspace
  params:
  - name: image-name
    type: string
    default: <your_dockerhub_username>/my-rails-app
  tasks:
```

```
    - name: fetch-repository
      taskRef:
        name: git-clone
      workspaces:
      - name: output
        workspace: shared-workspace
      params:
      - name: url
        value: <your_public_github_repo_url>
      - name: revision
        value: main # Or your branch name
    - name: build-and-push-image
      taskRef:
        name: kaniko
      runAfter:
        - fetch-repository
      workspaces:
      - name: source
        workspace: shared-workspace
      params:
      - name: IMAGE
        value: $(params.image-name)
      - name: DOCKERFILE
        value: Dockerfile
    - name: update-k8s-deployment
      taskRef:
        name: kubectl-apply
      runAfter:
        - build-and-push-image
      params:
      - name: PATH_TO_YAML_FILE
        value: my-rails-app.yaml
      - name: Yaml_parameters
        value: "image=$(params.image-name)"
```

Create a task-kubectl-apply.yaml: This task updates your Kubernetes deployment with the new image.

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: kubectl-apply
spec:
  params:
  - name: PATH_TO_YAML_FILE
    type: string
    description: Path to the YAML file to apply
  - name: Yaml_parameters
    type: string
    description: 'Set parameters for the Yaml file'
  steps:
  - name: apply-manifests
    image: lachlanevenson/k8s-kubectl:latest # Or another kubectl image
    command:
    - /bin/bash
    args:
    - -c
```

```
      - |
        sed -i "s@<your_dockerhub_username>/my-rails-app@$Yaml_parameters@g"
    $PATH_TO_YAML_FILE
        kubectl apply -f $PATH_TO_YAML_FILE
```

```
sudo lsof -i -P | grep LISTEN

kubectl get services --all-namespaces
```

To run Tetkon Dashboard on port 8081 local.

```
kubectl port-forward service/tekton-dashboard -n tekton-pipelines 8081:9097

kubectl describe service -n tekton-pipelines tekton-dashboard
kubectl logs -n tekton-pipelines <tekton-dashboard-pod-name>
kubectl get pods -n tekton-pipelines
kubectl rollout restart deployment -n tekton-pipelines tekton-dashboard
```

2. Configure Git Credentials

- **Create a Kubernetes secret:** You need a secret to store your Git credentials (username and personal access token or password). This will allow your Tekton pipeline to authenticate with your Git repository.

```
apiVersion: v1
kind: Secret
metadata:
  name: git-credentials
type: kubernetes.io/basic-auth
data:
  username: <base64-encoded-username>
  password: <base64-encoded-pat-or-password>
```

- Replace <base64-encoded-username> with the base64-encoded value of your Git username.
- Replace <base64-encoded-pat-or-password> with the base64-encoded value of your Git personal access token (PAT) or password.

- 1. Replace Placeholders

- Open your pipeline.yaml file in a text editor.
- Locate the following placeholders and replace them with your actual values:
  - <your_public_github_repo_url>: The URL of your public GitHub repository where your Rails app code is located.
  - <image_with_git_and_sed>: A Docker image that has git and sed installed (e.g., ubuntu:latest, alpine/git).
  - <old_image_tag>: The current image tag in your my-rails-app.yaml deployment manifest that you want to replace.
  - <new_image_tag>: The new image tag you want to use (e.g., my-rails-app:v2, my-rails-app:latest).
  - <image_with_argocd_cli>: A Docker image that has the Argo CD CLI installed (e.g., argoproj/argocd).

Define the Tasks

You'll need to create separate YAML files to define the update-manifest-task and sync-argocd-task Tasks. These tasks will contain the steps that you previously had directly in the Pipeline.
Here's an example of how you can define the update-manifest-task:

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: update-manifest-task
spec:
  workspaces:
  - name: source
  steps:
  - name: update-image-tag
    image: ubuntu:latest
    workingDir: $(workspaces.source.path)
    command:
      - /bin/bash
    args:
      - -c
      - |
        sed -i 's/my-rails-app.yaml/my-rails-app:latest/g' my-rails-app.yaml
        git config user.email "tekton@example.com"
        git config user.name "Tekton"
        git add my-rails-app.yaml
        git commit -m "Update image tag"
        git config credential.helper 'store --file=/etc/git-credentials'
        git push origin main
    volumeMounts:
    - name: git-credentials
      mountPath: /etc/git-credentials
  volumes:
  - name: git-credentials
    secret:
      secretName: git-credentials
```

Similarly, you can create a sync-argocd-task.yaml file to define the sync-argocd-task.

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: sync-argocd-task
spec:
  steps:
  - name: argocd-sync
    image: argoproj/argocd
    command:
      - /bin/bash
    args:
      - -c
      - |
        argocd app sync my-rails-app
```

Explanation

- apiVersion: This specifies the API version for the Tekton Task object.

- kind: This indicates that the YAML defines a Tekton Task.
- metadata.name: This sets the name of the task to sync-argocd-task. You'll use this name to refer to the task in your pipeline.yaml.
- spec.steps: This section defines the steps that will be executed within the task.
    - name: The name of the step (argocd-sync).
    - image: The Docker image to use for this step (argoproj/argocd, which contains the Argo CD CLI).
    - command: The command to execute within the container (/bin/bash).
    - args: The arguments to pass to the command. In this case, it's a script that runs argocd app sync my-rails-app to synchronize your Argo CD application.

Apply the tasks

Apply these task definitions to your cluster:

```
kubectl apply -f update-manifest-task.yaml -n tekton-pipelines
kubectl apply -f sync-argocd-task.yaml -n tekton-pipelines
```

```
kubectl describe service -n tekton-pipelines tekton-pipelines-webhook
kubectl logs -n tekton-pipelines <webhook-pod-name>
kubectl get pods -n tekton-pipelines -l app=tekton-pipelines-webhook

kubectl get pods -n tekton-pipelines

kubectl get pods -n tekton-pipelines -l app=tekton-pipelines-webhook -o
jsonpath='{.items[0].spec.containers[0].image}'

kubectl apply -f update-manifest-task.yaml -n tekton-pipelines
kubectl apply -f sync-argocd-task.yaml -n tekton-pipelines
    kubectl apply -f pipeline.yaml -n tekton-pipelines
```

Replace with your PVC name

- claimName: pvc-for-workspace
    - This line is within the workspaces section of the pipelinerun.yaml file.
    - Replace "pvc-for-workspace" with the actual name of the Persistent Volume Claim (PVC) you created for your Tekton workspace.
    - If you haven't created a PVC yet, you'll need to create one. Here's a simple example of a PVC YAML file:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: my-workspace-pvc
 namespace: tekton-pipelines
spec:
 accessModes:
  - ReadWriteOnce
 resources:
  requests:
   storage: 1Gi
```

Create Pipelinerun.yaml -

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
```

```
  metadata:
    name: my-rails-app-run-1
    namespace: tekton-pipelines
  spec:
    pipelineRef:
      name: my-rails-app-pipeline
    params:
    - name: url
      value: https://github.com/sansugupta/Budget-App.git
    - name: revision
      value: main
    serviceAccountName: dockerhub-service-account
    workspaces:
    - name: shared-workspace
      persistentVolumeClaim:
        claimName: my-workspace-pvc
```

Docker Service Account -

```
kubectl get serviceaccounts --all-namespaces
kubectl get serviceaccounts -n tekton-pipelines
vim dockerhub-secret.yaml
```

Here's how to create the service account and secret:

1. **Create the secret:**
    - Create a YAML file named dockerhub-secret.yaml with the following content:

```
apiVersion: v1
kind: Secret
metadata:
 name: dockerhub-secret
 namespace: tekton-pipelines
type: kubernetes.io/dockerconfigjson
data:
 .dockerconfigjson: <base64-encoded-docker-config>
```

Replace <base64-encoded-docker-config> with the base64-encoded string of your Docker Hub credentials in the following format:

```
{
 "auths": {
  "https://index.docker.io/v1/": {
   "auth": "<base64-encoded-username-and-password>"
  }
 }
}
```

Replace <base64-encoded-username-and-password> with the base64-encoded string of your_dockerhub_username:your_dockerhub_password.

Create the service account:

- Create a YAML file named dockerhub-service-account.yaml with the following content:

```
apiVersion: v1
kind: ServiceAccount
metadata:
 name: dockerhub-service-account
 namespace: tekton-pipelines
```

secrets:
- name: dockerhub-secret

sanskar@SANSKAR:~/Budget-App$ echo -n 'sansugupta:jhoncena@966' | base64
c2Fuc3VndXB0YTpqaG9uY2VuYUA5NjY=
vim config.json

```
{
    "auths": {
      "https://index.docker.io/v1/": {
        "auth": "c2Fuc3VndXB0YTpqaG9uY2VuYUA5NjY="
      }
    }
  }
```

sanskar@SANSKAR:~/Budget-App$ base64 config.json
ewogICAgImF1dGhzIjogewogICAgICAiaHR0cHM6Ly9pbmRleC5kb2NrZXIuaW8vdjEvIjogewog
ICAgICAgICJhdXRoIjogImMyRnVjM1ZuZFhCMFlUcHFhRzl1WTJWdVlVQTVOalk9IiAKICAgICAg
fQogICAgfQogIH0=

kubectl apply -f dockerhub-secret.yaml
kubectl apply -f dockerhub-service-account.yaml

kubectl get serviceaccounts -n tekton-pipelines
kubectl get secrets -n tekton-pipelines

Git-Clone for Tekton -

kubectl apply -f https://api.hub.tekton.dev/v1/resource/tekton/task/git-clone/0.9/raw -n tekton-pipelines

tekton-dashboard-role.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: tekton-dashboard-role
  namespace: tekton-pipelines
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["get", "list", "watch"]
```

tekton-dashboard-rolebinding.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: tekton-dashboard-rolebinding
  namespace: tekton-pipelines
subjects:
- kind: ServiceAccount
  name: tekton-dashboard
roleRef:
  kind: Role
  name: tekton-dashboard-role
```

```
  apiGroup: rbac.authorization.k8s.io
```

```
kubectl apply -f tekton-dashboard-role.yaml
kubectl apply -f tekton-dashboard-rolebinding.yaml
```

Trouble Shooting commands for CrashLoopBackoff -

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

kubectl apply -f dockerhub-secret.yaml
kubectl apply -f dockerhub-service-account.yaml
kubectl apply --filename \
https://storage.googleapis.com/tekton-releases/triggers/latest/release.yaml
kubectl apply --filename \
https://storage.googleapis.com/tekton-releases/triggers/latest/interceptors.yaml

kubectl apply -f argocd-cm.yaml
kubectl apply -f argocd-rbac-cm.yaml
kubectl apply -f github-repo-secret.yaml
kubectl apply -f application.yaml

kubectl port-forward service/argocd-server -n argocd 8080:443
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d
kubectl get pods -n argocd

docker run -d --name my-postgres-db -e POSTGRES_USER=myuser -e POSTGRES_PASSWORD=mypassword -e POSTGRES_DB=mydatabase my-postgres-db
af24e2ee2956b5b7e650311b9133c2cd853d474816aca3dcd35a043e1c1e6725

docker run -d --name my-rails-app -p 3000:3000 -e DATABASE_HOST=my-postgres-db -e DATABASE_USER=myuser -e DATABASE_PASSWORD=mypassword -e DATABASE_NAME=mydatabase my-rails-app
92995ffd7f95fed10781a4fa1302dcf5588378db61afda65ba9b2a3c89525df0

minikube delete
minikube start --disk-size=50g

kubectl delete -f https://storage.googleapis.com/tekton-releases/pipeline/previous/v0.65.0/release.yaml

kubectl delete namespace tekton-pipelines
kubectl apply -f https://storage.googleapis.com/tekton-releases/pipeline/previous/v0.65.0/release.yaml
kubectl get pods -n tekton-pipelines
```

Things to apply in Tekton-pipelines.

```
kubectl apply -f git-credentials.yaml -n tekton-pipelines
kubectl apply -f update-manifest-task.yaml -n tekton-pipelines
kubectl apply -f sync-argocd-task.yaml -n tekton-pipelines
kubectl apply -f pipeline.yaml -n tekton-pipelines
kubectl apply -f my-workspace-pvc.yaml -n tekton-pipelines
kubectl apply -f pipelinerun.yaml
```

→ kubectl port-forward service/tekton-dashboard -n tekton-pipelines 8081:9097

Tetkon Dashboard

→ kubectl apply --filename https://storage.googleapis.com/tekton-releases/dashboard/latest/release.yaml

Install the Task

→ **kubectl apply -f https://api.hub.tekton.dev/v1/resource/tekton/task/kaniko/0.6/raw**

sanskar@SANSKAR:~/Budget-App$ kubectl get configmaps -n tekton-pipelines -l app.kubernetes.io/part-of=tekton-dashboard