# Deploying a Ruby on Rails API-Only Application to Kubernetes Using Docker and DockerHub

## Overview

This report provides a comprehensive walkthrough for deploying a Rails API-only application using Docker and Kubernetes to EKS. I will cover the process of implementing a rails application that converts currency for a user based on the currency parameters passed. I will also containerize the application with Docker, creating a Kubernetes configuration file, and deploying the application. This setup is ideal for a Rails application that relies on a database service like postgres.

## The Application

The application to be deployed is a currency converter api-only application. This means that the application will take a REST request and return a JSON response. To develop this application, I started with creating a rails application with the command:

rails new currency\_converter --api -d postgresql

This commands bootstraps the application with basic configuration to use postgresql as the preferred database.

Since the application is a currency converter application, created an user accounts table in the database, so users can register, or login, to get an authentication token. Token generation is handled by [jwt](https://jwt.io/) (JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties) a popular token generator, and then this token is used as an authentication token in the header of all following requests to convert a currency pair. Rails provides a generator to generate models which create schema migrations that can be migrated to the database

rails generate model Account email:string password\_digest

and then the controller that handles requests accounts\_controller.rb

# accounts\_controller.rb  
...  
  
def register  
 @account = Account.new(account\_params)  
 if @account.save  
 success(message: "Account created successfully", data: {account: @account, token: @account.token})  
 else  
 unprocessable(errors: @account.errors.messages)  
 end  
end  
  
def login  
 @account = Account.find\_by(email: account\_params[:email])  
 if @account && @account.authenticate(account\_params[:password])  
 success(message: "Logged in successfully", data: {account: @account, token: @account.token})  
 else  
 unauthorized(message: "incorrect email/password")  
 end  
end  
  
private  
def account\_params  
 params.require(:account).permit(:email, :password, :password\_confirmation)  
end  
  
...

As the focus of this report is on the deployment to a cloud service, I would not go into details of how the two functions above work, however the following are classes and controllers that are responsible for currency conversions.

class Currency  
 @req = Requester.new(ENV["FCA\_URL"])  
  
  
 def self.exchange(base\_currency, target\_currency)  
 url = "latest?apikey=#{ENV["FCA\_KEY"]}&base\_currency=#{base\_currency}&currencies=#{target\_currency}"  
  
 @req.get\_request(url)  
 end  
  
 def self.get\_history(currency, start, end\_date)  
 url = "historical?apikey=#{ENV["FCA\_KEY"]}&currencies=#{currency}&date\_from=#{start}&date\_to=#{end\_date}"  
  
 @req.get\_request(url)  
 end  
  
 def self.information(currency)  
 url = "currencies?apikey=#{ENV["FCA\_KEY"]}&currencies=#{currency}"  
  
 @req.get\_request(url)  
 end  
end

For this application, I implemented an API provided by [freecurrencyapi.com](https://freecurrencyapi.com). On this application, sending a POST request, with a base\_currency and a target\_currency would provide a response like:

{  
 "status": "success",  
 "message": "Currency exchanged successfully",  
 "data": {  
 "USD": 1.2604457213  
 }  
}

The application has been uploaded to a github repository for further inspection.

# The Deployment

### Prerequisites

* [**Docker**](https://docs.docker.com/get-docker/): Docker is essential for creating and managing application containers. Install Docker from its official website.
* [**Kubernetes**](https://minikube.sigs.k8s.io/docs/start/): An open-source system for automating deployment, scaling, and management of containerized applications.
* [**Minikube**](https://minikube.sigs.k8s.io/docs/start/): is recommended for setting up a local Kubernetes cluster.

## Step-by-Step Guide

### 1. Dockerizing the Rails Application

**Create a Dockerfile**

A Dockerfile is a script containing commands to assemble a Docker image for an application. The Dockerfile will outline the necessary steps to prepare the application environment.

1. **Selecting the Ruby Version:**

* Start by specifying the Ruby version that matches the application’s requirements. since our application is running on ruby version 3.2.1
* ARG RUBY\_VERSION=3.2.1  
  FROM ruby:$RUBY\_VERSION

1. **Installing Dependencies:**

* Install the required libraries and dependencies for Rails to run in the docker container.
* RUN apt-get update -qq && \  
   apt-get install -y build-essential libvips bash bash-completion libffi-dev tzdata postgresql nodejs npm yarn && \  
   apt-get clean && \  
   rm -rf /var/lib/apt/lists/\* /usr/share/doc /usr/share/man

1. **Setting Up the Application Directory:**

* Create a working directory for your Rails application within the image.
* WORKDIR /rails

1. **Configuring Environment Variables:**

* Define the necessary environment variables for Rails to run in a production environment as in when it is deployed.
* ENV RAILS\_LOG\_TO\_STDOUT="1" \  
   RAILS\_SERVE\_STATIC\_FILES="true" \  
   RAILS\_ENV="production" \  
   BUNDLE\_WITHOUT="development"

1. **Installing Gems:**

* Copy the Gemfile and Gemfile.lock to the image and run bundle install.
* COPY Gemfile Gemfile.lock ./  
  RUN bundle install  
    
  # Copy application code  
  COPY . .

1. **Setting Up the Database and Server:**

* Add a script that handles database creation, migration, before starting the Rails server.
* # Give permission to execute the script  
  RUN chmod +x /rails/bin/docker-entrypoint.sh  
    
  ENTRYPOINT ["/rails/bin/docker-entrypoint.sh"]  
    
  EXPOSE 3000  
    
  CMD ["rails", "server", "-b", "0.0.0.0"]
* The docker-entrypoint.sh script:
* #!/bin/bash  
   set -e  
    
   # Check for a server.pid file and remove it if it exists  
   # This file sometimes causes issues when starting the container  
   if [ -f tmp/pids/server.pid ]; then  
   echo "Removing server.pid"  
   rm tmp/pids/server.pid  
   fi  
    
   # Run database migrations  
   echo "Running database migrations..."  
   bundle exec rails db:migrate  
    
   # Then exec the container's main process (what's set as CMD in the Dockerfile)  
   exec "$@"
* **Conclusion:**
* I created a file named Dockerfile without any extensions and add it to the root of the rails application. The final file looks like this:
* FROM ruby:3.2.1  
    
   RUN apt-get update -qq && \  
   apt-get install -y build-essential libvips bash bash-completion libffi-dev tzdata postgresql nodejs npm yarn && \  
   apt-get clean && \  
   rm -rf /var/lib/apt/lists/\* /usr/share/doc /usr/share/man  
    
   WORKDIR /rails  
    
   ENV RAILS\_LOG\_TO\_STDOUT="1" \  
   RAILS\_SERVE\_STATIC\_FILES="true" \  
   RAILS\_ENV="production" \  
   BUNDLE\_WITHOUT="development"  
    
   COPY Gemfile Gemfile.lock ./  
   RUN bundle install  
    
   COPY . .  
    
   RUN chmod +x /rails/bin/docker-entrypoint.sh  
    
   # commands that run everytime during build  
   ENTRYPOINT ["/rails/bin/docker-entrypoint.sh"]  
    
   EXPOSE 3000  
    
   CMD ["rails", "server", "-b", "0.0.0.0"]

### 2. Using docker-compose

**Create a docker-compose.yml**

docker compose is a tool used for defining and running multiple services in a Docker container that may depend on one another to function properly.

This step is unnecessary, but it makes sense to use docker-compose before using kubernetes as it simulates connections between services. This is an attempt to use docker-compose to run multiple services on a docker image before creating a cluster using minikube for kubernetes.

1. **Selecting a docker compose version:**

* Start by specifying the Docker Compose file format version. Here, version 3 is used:
* version: 3

1. **Configure the Database Service:**

* The application depends on a database and the application uses postgres, therefore, create a database container from the official postgresql image and run it as a service
* services:  
   database-service:  
   image: postgres:14.2-alpine  
   container\_name: currency-postgres  
   volumes:  
   - pg\_data:/var/lib/postgresql/data  
   command: "postgres -c 'max\_connections=500'"  
   env\_file:  
   - ".env"  
   ports:  
   - "5432:5432"
* The service to be created is labelled as database-service with a container name currency-postgres. This name would be used to reference this service from the main rails application service to be created below. Included keys and values referenced in the configuration are:
  + volumes a path to where the database data will be stored for persistence. so it would not be lost if the service is restarted
  + the command to execute/start the container
  + environment variables to hold the environment variables needed to create the database like the postgres user and password
  + tcp ports where our services would be available from

1. **Adding the main application service:**

* The application also needs a service to run properly from the container. to do so, define the configuration for that service as shown below.
* service:  
   app-service:  
   build: .  
   command: "./bin/rails server"  
   image: urchymanny/currency-converter:latest  
   container\_name: currency-converter-server  
   env\_file:  
   - ".env"  
   volumes:  
   - currency\_storage:/rails/storage  
   - .:/rails # this line adds a bind mount. this means any change made in the Working dir is automatically reflected on the running version  
   depends\_on:  
   - database-service  
   ports:  
   - "3000:3000"
* In the above code snippet, a service labelled app-service which would build from the current folder is added with the container name currency-converter-server, and run the command rails server.
  + Added an env\_file configuration to load environment variables from a .env file on the root directory of the application, a file that contains secret credentials.
  + Also configured the volumes directory as persistent storage for the application.
  + Added a “bind mount” .:/rails\* in the volumes configuration which binds the application on our local filesystem to the application in the working directory of the docker container. **NB** that the /rails is exactly the same as the WORK\_DIR in the Dockerfile
  + Configured the depends\_on to make the app-service service depend on the database-service service that was created earlier
  + lastly, mapped the container service to port 3000

1. **Adding the Volumes**

* It is important that the application data like the database state or created files are not lost when the container is restarted or recreated hence, there is a need to create some sort of persistent storage.
* volumes:  
   postgres\_data: {}  
   app-storage: {}
* This creates a persistent storage that are referenced by the app-service and database-service services above.
* **Conclusion:**
* Create a file named docker-compose.yml file in the root of your rails application. It should look like this:
* version: "3"  
   services:  
   database-service:  
   image: postgres:14.2-alpine  
   container\_name: currency-postgres  
   volumes:  
   - pg\_data:/var/lib/postgresql/data  
   command: "postgres -c 'max\_connections=500'"  
   env\_file:  
   - ".env"  
   ports:  
   - "5432:5432"  
   app-service:  
   build: .  
   command: "./bin/rails server"  
   image: urchymanny/currency-converter:latest  
   container\_name: currency-converter-server  
   env\_file:  
   - ".env"  
   volumes:  
   - currency\_storage:/rails/storage  
   - .:/rails # this line adds a bind mount. this means any change made in the Working dir is automatically reflected on the running version  
   depends\_on:  
   - database-service  
   ports:  
   - "3000:3000"  
    
   volumes:  
   pg\_data: {}  
   currency\_storage: {}

### 3. Kubernetes Configurations

**What you need**

To deploy an application or service to kubernetes, you need to first understand what components it depends on to run completely. For kubernetes setup, a few configurations are needed. A persistent volume which contains data that should be persistent across multiple instances of the application running on countless machines; A deployment configuration that defines how an application should be deployed, from what image, how many CPUs, memory, replicas, etc; a service configuration that defines how a deployment should be accessed, amongst other configurations. I have covered a few that are necessary for this application’s deployment in this report.

* Deployment configuration
* Service Configuration
* Persistent Volume Storage
* ConfigMap
* Secrets

1. Deployment configuration

* A deployment configuration in Kubernetes defines how to deploy a containerized application.
* apiVersion: apps/v1  
  kind: Deployment
* Key elements include:
  + Name: Identifies the deployment.
  + Image: Specifies the Docker image to deploy.
  + Environment Variables: Sets necessary environment variables for the application.
  + Volumes: Defines the storage volumes used by the deployment.

1. Service configuration -

* A service in Kubernetes maps a deployment to a service name and port, allowing the application to be accessed
* apiVersion: v1  
  kind: Service

1. Persistent volume

* Persistent volumes in Kubernetes are used for data that should persist across restarts and deployments of the cluster. They are particularly important for stateful applications like databases.
* apiVersion: v1  
  kind: PersistentVolume

1. Persistent volume claim

* A persistent volume claim (PVC) is a request for storage by a user. It abstracts the details of how storage is provided from how it is consumed
* apiVersion: v1  
  kind: PersistentVolumeClaim

1. ConfigMap

* A ConfigMap is used to store non-confidential data in key-value pairs. ConfigMaps can be used to store settings, configuration files, and other non-sensitive data required by pods.
* apiVersion: v1  
  kind: ConfigMap

1. Secrets Configuration

* Secrets are similar to ConfigMaps but are specifically intended for sensitive data like passwords, tokens, or keys. They are stored more securely and should be used for confidential data. Values of keys in the secrets must be base64-encoded. Kubernetes Secrets ensure that this sensitive information is not exposed in plain text and is securely managed.
* apiVersion: v1  
  kind: Secret  
  type: Opaque

Before configuring the application for kubernetes, minikube and kubectl should be installed on the machine(macOS is assumed in this case)

$ brew install minikube

After installing minikube, minikube kubectl command will be available, but to make life easier, create an alias

alias kubectl="minikube kubectl --"

Now kubectl and minikube is available and can be used in the following instructions below.

## Deploying a rails application to kubernetes

Deploying the rails application to kubernetes requires, creating the components highlighted above for all the services that the application depends on like rails application, postgres, redis, sidekiq, etc. For the purpose of this implementation, the application would require a rails application deployment and the postgres deployment as in our docker-compose setup.

At this stage, it is assumed that the Docker image has been built successfully and pushed to the DockerHub registry. Kubernetes will attempt to pull the image from the official registry, using the driver that is specified.

### The Rails application configuration

Docker credentials is needed to authorize kubernetes pull requests as the deployment configuration would be pulling the image from the official DockerHub registry. Hence, run the following command (replace <username>, <password> and <docker-email>) to add these credentials to kubernetes.

kubectl create secret docker-registry regcred --docker-server=https://index.docker.io/v1/--docker-username=<username> --docker-password=<Password> --docker-email=<docker-email>

This command would create a kubernetes secret with the name regcred that would be referenced by the

Also, environment variables can be created from an existing .env file to avoid manually converting all the environment variables into base64 encoded values and adding them to a secret.yaml configuration file. Run the following command to generate a kubernetes secret.

kubectl create secret generic my-secret --from-env-file=.env-new

This would create a kubernetes secret with the name my-secret from the .env-new file available in the rails application. Here is an example of the .env file:

RAILS\_ENV=production  
POSTGRES\_HOST=db

Create a a file to configure the application persistent storage app-pv.yaml

apiVersion: v1  
kind: PersistentVolume  
metadata:  
 name: currency-app-pv  
spec:  
 capacity:  
 storage: 1Gi  
 volumeMode: Filesystem  
 accessModes:  
 - ReadWriteOnce  
 storageClassName: standard  
 hostPath:  
 path: /rails/storage  
 type: DirectoryOrCreate

The metadata.name property specifies the name of the volume to be created for our rails application. spec.hostPath specifies the path to the directory where our application data will be stored. Please refer to the kubernetes document for more information about each of these properties.

This persistent storage requires a separate claim configuration app-pvc.yaml:

apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
 name: currency-app-pvc  
spec:  
 resources:  
 requests:  
 storage: 1Gi  
 volumeName: currency-app-pv  
 accessModes:  
 - ReadWriteOnce  
 storageClassName: standard

the spec.volumeName maps the claim to the volume configured in the rails-pv configuration.

Next, Create a file app-deployment.yaml that contains the configuration for the deployment of the application.

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: currency-app # sets the name of the deployment  
spec:  
 replicas: 1 # defines the number of replicas per node  
 selector:  
 matchLabels:  
 app: currency-app # defines which pod the deployment is for  
 template: # defines a template for each replica  
 metadata:  
 labels:  
 app: currency-app # defines the label to be applied to pods created from this deployment  
 spec:  
 containers:  
 - name: currency-app # defines the name of the container within each pod  
 image: urchymanny/currency-converter:v3 # Image from which the container is created (usually from a registry)  
 envFrom:  
 - secretRef:  
 name: my-secret # using the secret created from our .env file  
 resources:  
 limits:  
 memory: "128Mi"  
 cpu: "200m"  
 ports:  
 - containerPort: 3000  
 volumeMounts:  
 - mountPath: /rails/storage  
 name: currency-app-pv # name of the persistent volume to be mounted for the container  
 restartPolicy: Always  
 imagePullSecrets:  
 - name: regcred # using the secret created from our registry credentials  
 volumes:  
 - name: currency-app-pv  
 persistentVolumeClaim:  
 claimName: currency-app-pvc

Eventually creating a service for the container will be the last necessary step rails-service.yaml:

apiVersion: v1  
kind: Service  
metadata:  
 name: currency-app-service # Sets the name of the Service  
spec:  
 selector:  
 app: currency-app # Maps this service to a pod with a matching label  
 ports:  
 - protocol: TCP # Specifies the protocol used by the service (TCP/UDP)  
 port: 3000 # The port on which the service will be exposed  
 targetPort: 3000 # The port on the pod to which this service will forward traffic  
 nodePort: 30000  
 type: LoadBalancer # The type of service, LoadBalancer exposes the service externally

The most important property in the service is the selector.app which maps the created service to a pod with the same label.

In **conclusion**, A deployment was configured to run a single (1) replica of a pod that would be built from the image specified. Then a service is created for the pod with a port and targetPort.

### The Postgres Database configuration

Following the configuration of the rails application, the configuration for the database is going to be quite easy. The needed components are more or less the same; a deployment, service, persistent storage and persistent storage claim.

pg-deployment.yaml:

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: currency-pg  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: currency-pg  
 template:  
 metadata:  
 labels:  
 app: currency-pg  
 spec:  
 containers:  
 - name: currency-pg  
 image: postgres:14.2-alpine  
 envFrom:  
 - secretRef:  
 name: my-secret  
 resources:  
 limits:  
 memory: "128Mi"  
 cpu: "200m"  
 ports:  
 - containerPort: 5432  
 volumeMounts:  
 - name: currency-pg-pv  
 mountPath: /var/lib/postgresql/data  
 subPath: data  
 volumes:  
 - name: currency-pg-pv  
 persistentVolumeClaim:  
 claimName: currency-pg-pvc

pg-pv.yaml:

apiVersion: v1  
kind: PersistentVolume  
metadata:  
 name: currency-pg-pv  
spec:  
 capacity:  
 storage: 1Gi  
 volumeMode: Filesystem  
 accessModes:  
 - ReadWriteOnce  
 storageClassName: standard  
 hostPath:  
 path: /var/lib/postgresql/data  
 type: DirectoryOrCreate

pg-pvc.yaml:

apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
 name: currency-pg-pvc  
spec:  
 resources:  
 requests:  
 storage: 1Gi  
 volumeMode: Filesystem  
 accessModes:  
 - ReadWriteOnce  
 volumeName: currency-pg-pv  
 storageClassName: standard

and lastly, the service would be created with the configuration below: pg-service.yaml:

apiVersion: v1  
kind: Service  
metadata:  
 name: currency-pg-service  
spec:  
 selector:  
 app: currency-pg  
 ports:  
 - protocol: TCP  
 port: 5432  
 targetPort: 5432

At this point, the rails application and a postgres database have been successfully configured for kubernetes, however we still need to configure the database to depend on the database service that is deployed by pg-deployment.yaml. This is relatively easy to do, just edit the rails application config/database.yml file to have the following:

default: &default  
 adapter: postgresql  
 encoding: unicode  
 pool: <%= ENV.fetch("RAILS\_MAX\_THREADS") { 5 } %>  
  
production:  
 <<: \*default  
 database: <%= ENV['POSTGRES\_DB'] %>  
 host: currency-db # as the database host would be the service created.  
 username: <%= ENV['POSTGRES\_USER'] %>  
 password: <%= ENV['POSTGRES\_PASSWORD'] %>

The host of the production database would be the postgres service configured by pg-service.yaml that was created.

The application is ready to be deployed to kubernetes. The following command uses kubectl to apply the configurations to kubernetes minikube cluster.

kubectl apply -f <folder>

Assuming that all the configuration yaml files are in a folder named /deploy/, the command above will apply all configuration files to kubernetes.

However, This can be redundant, so a .sh script can me made to automatically execute the application of the configuration and removal of all configuration.

#!/bin/bash  
  
# Function to deploy resources  
deploy() {  
 kubectl create secret docker-registry regcred --docker-server=https://index.docker.io/v1/ --docker-username="$DOCKER\_USERNAME" --docker-password="$DOCKER\_PASSWORD" --docker-email="$DOCKER\_EMAIL"  
 kubectl create secret generic my-secret --from-env-file=.env-new  
  
 echo "Deploying PostgreSQL..."  
 kubectl apply -f deploy/pg-pv.yaml  
 kubectl apply -f deploy/pg-deployment.yaml  
 kubectl apply -f deploy/pg-service.yaml  
  
 echo "Waiting for PostgreSQL to be ready..."  
 kubectl wait --for=condition=ready pod -l app=currency-pg --timeout=60s  
  
 echo "Deploying Rails application..."  
 kubectl apply -f deploy/app-pv.yaml  
 kubectl apply -f deploy/app-deployment.yaml  
 kubectl apply -f deploy/app-service.yaml  
  
 echo "Deployment process completed successfully."  
  
 # minikube service currency-app-service  
}  
  
# Function to reverse deployment  
reverse() {  
 echo "Deleting Rails application resources..."  
 kubectl delete -f deploy/app-service.yaml  
 kubectl delete -f deploy/app-deployment.yaml  
 kubectl delete -f deploy/app-pv.yaml  
  
 echo "Waiting for Rails resources to be deleted..."  
 sleep 10  
  
 echo "Deleting PostgreSQL resources..."  
 kubectl delete -f deploy/pg-service.yaml  
 kubectl delete -f deploy/pg-deployment.yaml  
 kubectl delete -f deploy/pg-pv.yaml  
  
 echo "Waiting for PostgreSQL resources to be deleted..."  
 sleep 10  
  
 echo "Deleting created secrets..."  
 kubectl delete secret my-secret  
 kubectl delete secret regcred  
  
 echo "Reversal process completed successfully."  
}  
  
# Check for command line argument  
case "$1" in  
 deploy)  
 deploy  
 ;;  
 reverse)  
 reverse  
 ;;  
 \*)  
 echo "Usage: $0 {deploy|reverse}"  
 exit 1  
 ;;  
esac

This script can be saved in the root directory of the rails application

* Make sure docker is installed and running before running this script
* Start minikube before running this script
* minikube start
* To run the executable deploy.sh, permissions must be given to the file first.
* $ run chmod +x deploy.sh  
  $ ./deploy.sh deploy
* The result of running the script above will be something similar to the table below
* |-----------|---------|-------------|---------------------------|  
  | NAMESPACE | NAME | TARGET PORT | URL |  
  | --------- | ------- | ----------- | ------------------------- |  
  | default | ead-app | 3000 | http://192.168.49.2:30582 |  
  |-----------|---------|-------------|---------------------------|  
  🏃 Starting tunnel for service ead-app.  
  |-----------|---------|-------------|---------------------------|  
  | NAMESPACE | NAME | TARGET PORT | URL |  
  | --------- | ------- | ----------- | ------------------------- |  
  | default | ead-app | | http://127.0.0.1:51975 |  
  |-----------|---------|-------------|---------------------------|
* This shows us the IP address and port which our application is externally available from, in the example above, this address is http://127.0.0.1:51975

# Conclusion: Deploying Applications with Docker and Kubernetes

Deploying an application using Docker and Kubernetes involves a series of systematic steps to ensure a smooth and efficient process. This process can be summarized as follows:

1. **Create a Docker Image**: The initial step involves creating a Docker image for your application. This image encapsulates the application, its dependencies, and the runtime environment, ensuring consistency across various deployment stages.
2. **Prepare Kubernetes Configuration Files**: Once the Docker image is ready, the next step is to prepare a set of Kubernetes configuration files. These files define how your application should be deployed, managed, and scaled in a Kubernetes cluster. Key configurations include:
   * **Deployment**: Specifies how the application’s containers should be run and managed.
   * **Service**: Defines how to expose the application (or a part of it) to the network, potentially making it accessible from outside the Kubernetes cluster.
   * **Persistent Volumes and Claims**: Ensure that data persists across container restarts and re-deployments.
   * **ConfigMaps and Secrets**: Manage configuration data and sensitive information securely.
3. **Apply Configurations to the Kubernetes Cluster**: With the configuration files ready, the next step is to apply them to your Kubernetes cluster. This step creates the necessary Kubernetes resources (like Deployments, Services, etc.) and starts the process of deploying your application within the cluster.
4. **Expose the Application to External Clients**: Finally, the application is made accessible to external clients. This is typically achieved through a Kubernetes Service of the type LoadBalancer or NodePort, which routes external traffic to the correct pods within the cluster.

In summary, deploying an application with Docker and Kubernetes requires careful planning and execution, involving image creation, configuration file preparation, application of these configurations to a Kubernetes cluster, and exposure of the application to the external world. This process ensures a scalable, manageable, and efficient deployment, leveraging the strengths of containerization and orchestration.

# Deploying the configuration to an EKS cluster

Once the configuration has been tested and deployed successfully in the minikube local cluster, we are sure that the application would run on any cluster or group of clusters. This makes it easy for us to deploy the applications to a kubernetes cluster on the cloud on a managed service like Amazon Elastic Kubernetes Service.