Deploy your app with Code Engine

Introduction

Now that you have Gradio to help you generate a user interface for an application, let's see how you can run applications on IBM Cloud and access it with a public URL using IBM Code Engine.

Learning objectives

At the end of this project, you will be able to:

- Create a container image
- Create the files required to deploy your app in a container
- Create your Code Engine project
- Build a container image with Code Engine
- Deploy a containerized app using Code Engine

Container images and containers

Code Engine lets you run your apps in containers on IBM Cloud. A **container** is an isolated environment or place where an application can run independently. Containers can run anywhere, such as on operating systems, virtual machines, developer's machines, physical servers, and so on. This allows the containerized application to run anywhere as well and the isolation mechanism ensures that the running application will not interfere with the rest of the system.

Containers are created from container images. A container image is basically a snapshot or a blueprint that indicates what will be in a container when it runs. Therefore, to deploy a containerized app, you first need to create the app's container image.

Creating the container image

The files required to deploy your app in a container are as follows:

- You can use the Gradio framework for generating the user interface of the app,for example, the Python script that contains the code to create and launch the **gradio.Interface** can be named as `demo.py`.
- The source code of the app has its dependencies, such as libraries that the code uses. Hence, you need a `requirements.txt` file that specifies all the libraries the source code depends on.
- You need a file that shows the container runtime the steps for assembling the container image, named as `Dockerfile`.

Let's create the three files. Open a terminal and while you are in the `home/project` directory, make a new directory `myapp` for storing the files and go into the directory with the following command:

```
1. 1
2. 2
```

```
1. mkdir myapp
2. cd myapp
```

Copied! Executed!

Now that you are in the `myapp` directory, run the following command to create the files:

```
1. 1
```

```
1. touch demo.py Dockerfile requirements.txt
```

Copied! Executed!

If you open the file explorer, you will see the files you created.

myapp Directory

Next, let's take a closer look at what should be included in each of the three files.

Step 1: Creating requirements.txt

If you are a Data Scientist, you may be familiar with the `pip3 install <library-name>` command for installing libraries. By using a `requirements.txt` file that consists of all libraries you need, you can install all of them into your environment at once with the command `pip3 install -r requirements.txt`.

Your goal is to deploy the app in a container; thus, all the dependencies need to go into the container as well.

Let's create a `requirements` file to cover all the libraries and dependencies your app may need.

Open the `requirements.txt` file in `/myapp` and paste the following library names into the file.

```
1. 1
```

```
1. gradio==4.44.0
```

Copied!

Testing requirements.txt locally

Since you have already installed the required packages in the "QuickStart Gradio" section, you don't need to install them again here. However, it's important to test the application locally before launching it through Docker to ensure that it runs smoothly without any errors.

You can test to see if the file works correctly by executing the following command in the terminal of your current CloudIDE working environment:

```
1. 1
```

```
1. pip3 install -r requirements.txt
```

Copied! Executed!

Note: Make sure you are in the `myapp` directory and your virtual environment `my_env` created previously is activated. This allows the libraries to be installed into your virtual environment only.

Now that you have the libraries you need, let's write the Python code for the text-generation model demo.

Step 2: Creating demo.py

The following code guides you creating a simple Gradio web application. If you want to know more about Gradio, please refer to here.

Open the `demo.py` file in `/myapp` and fill the empty script with the following code.

```
1. 1
2. 2
```

```
3
4
5
6
7
8
9
10
11
12
```

```
import gradio as gr

def greet(name, intensity):
    return "Hello, " + name + "!" * int(intensity)

demo = gr.Interface(
    fn=greet,
    inputs=["text", "slider"],
    outputs=["text"],
)

demo.launch(server_name="0.0.0.0", server_port= 7860)
```

Copied!

Let's test your application and make sure it can run properly.

Testing demo.py locally

Open your terminal and go to myapp directory with `cd myapp` command.

If you have successfully executed `pip3 install -r requirements.txt` in the previous step, you are good to go. If not, please run the command now to have the required libraries installed in your working environment.

Run the following command in the terminal:

```
1
```

```
python3 demo.py
```

Copied! Executed!

If you run this script properly, you should see the following result in your terminal:

demo.py

The result shows that your app is downloaded and the app is running on http://0.0.0.0:7860/. Click on the button below to access the web app hosted in the CloudIDE:

Web application

You can execute `ctrl+c` to shut down the application.

Now let's create the `Dockerfile` which shows the container runtime what to do with your files for constructing the container image.

Step 3: Creating Dockerfile

The `Dockerfile` is the blueprint for assembling a container image.

Open the `Dockerfile` in `/myapp` and paste the following commands into the file.

```
1
2
3
4
5
6
7
8
9
```

```
FROM python:3.10

WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip3 install --no-cache-dir -r requirements.txt

COPY . .

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

Copied!

What does the Dockerfile do?

FROM python:3.10

Docker images can be inherited from other images. Therefore, instead of creating your own base image, you will use the official Python image `python:3.10` that already has all the tools and packages that you need to run a Python application.

WORKDIR /app

To facilitate the running of your commands, let's create a working directory `/app`. This instructs Docker to use this path as the default location for all subsequent commands. By creating the directory, you do not have to type out full file paths but can use relative paths based on the working directory.

COPY requirements.txt requirements.txt

Before you run `pip3 install`, you need to get your requirements.txt file into your image. You can use the `COPY`command to transfer the contents. The `COPY`command takes two parameters. The first parameter indicates to the Docker what file(s) you would like to copy into the image. The second parameter indicates to the Docker the location where the file(s) need to be copied. You can move the `requirements.txt` file into your working directory `/app`.

RUN pip3 install –no-cache-dir -r requirements.txt

Once you have your `requirements.txt` file inside the image, you can use the `RUN` command to execute the command `pip3 install --no-cache-dir -r requirements.txt`. This works exactly the same as if you were running the command locally on your machine, but this time the modules are installed into the image.

COPY

At this point, you have an image that is based on Python version 3.10 and you have installed your dependencies. The next step is to add your source code to the image. You will use the `COPY` command just like you did with your `requirements.txt` file above to copy everything in your current working directory to the file system in the container image.

CMD ["python", "demo.py"]

Now, you have to indicate to the Docker what command you want to run when your image is executed inside a container. You use the CMD command. Docker will run the `python demo.py` command to launch your app inside the container.

Now that all three files have been created, let's bring in the Code Engine for building the container image.

IBM Code Engine project

IBM Code Engine is a fully managed, serverless platform that runs your containerized workloads, including web apps, micro-services, event-driven functions, or batch jobs. Code Engine even builds container images for you from your source code. All these workloads can seamlessly work together because they are all hosted within the same Kubernetes infrastructure. The Code Engine experience is designed so that you can focus on writing code and not on the infrastructure that is needed to host it.

Creating your Code Engine (CE) project

To deploy serverless apps using Code Engine you'll need a project. A **project** is a grouping of Code Engine entities such as applications, jobs, and builds. Projects are used to manage resources and provide access to its entities.

In this guided project, you already have a CE project set up for you so you don't need to go through creating and configuring the project parameters. Now it's time to click on the button below to create your project!

Create Code Engine project in IDE

Wait for 3 to 5 minutes and you should see the following indicating that your Code Engine project is ready to use.

Code Engine CLI Button

Launching the Code Engine (CE) CLI

Once your project is ready, click on the **Code Engine CLI** button to launch your project in the terminal. The new terminal that just opened should show information about your current CE project, such as the name and ID of your project and the region where your project is deployed.

Project Details

As you created your current project, you have resources on IBM Cloud for the project, such as CPU runtime, memory, storage, and so on.

You can access the information of your project in Cloud IDE by clicking on the "Code engine" page and selecting "Project Summary".

You can check for limits and quota usage of this project's allocated resources by running this command:
Note: Surround your project name with double quotes if it has space in the characters.

```
1. 1
```

```
1. ibmcloud ce project get --name PROJECT_NAME
```

Copied! Executed!

Next, you are going to use these resources to deploy your app.

Building a Container image with Code Engine

In a Code Engine build, you need to define a source that points to a place where your source code and Dockerfile reside, it could be a public or private Git repository, or a local source if you want to run the build using the files in your working directory.

Since you have created all the files in the `myapp` directory, you are going to build the image from a local source, and then upload the image to your container registry with the registry access that you provide. Let's first change the working directory to `/myapp` where you have the source code and other files.

if you are not in the myapp folder change the working directory to it:

```
1. 1
```

```
1. cd myapp
```

Copied! Executed!

Container registries

A container registry, or registry, is a service that stores container images. For example, IBM Cloud Container Registry and Docker Hub are container registries.

Images that are used by IBM Cloud Engine are typically stored in a registry that can either be accessible by the public (public registry) or set up with limited access for a small group of users (private registry).

Code Engine requires access to container registries to complete the following actions:

- To store and retrieve local files when a build is run from a local source
- To store a newly created container image as an output of an image build
- To pull a container image to run an app or job

Your IBM Cloud Container Registry

When you created the Code Engine project, Code Engine also created a registry access secret for you to a private IBM Cloud Container Registry namespace. You can find the provided ICR namespace and the registry secret in the **Code Engine** tab.

Note that to use the provided ICR namespace to store docker images you will need to include `--registry-secret icr-secret` in most of your CE commands. **Code Engine handles many of the underlying details of the interactions between the system and your registry.**

Creating the build configuration

To create a build configuration that pulls code from a local directory, use the `build create` command and specify the `build-type` as `local`.

Since you also need Code Engine to store the image in the IBM Cloud Container Registry, you need to provide your registry access secret so that Code Engine can access and push the build result to the registry in your namespace.

Run the following command in your Code Engine CLI to create a build configuration:

```
1. 1
2. 2
3. 3
4. 4
```

```
5. 5
```

```
1. ibmcloud ce build create --name build-local-dockerfile1 \
2.                          --build-type local --size large \
3.                          --image us.icr.io/${SN_ICR_NAMESPACE}/myapp1 \
4.                          --registry-secret icr-secret
5.                          /
```

Copied! Executed!

After the command, the following result should be displayed in the terminal:

Build

With the `build create` command, you create a build configuration called `build-local-dockerfile1` and you specify `local` as the value for `--build-type`. The size of the build defines how many resources such as CPU cores, memory, and disk space are assigned to the build. You specify `size` as `large` if your model pipeline that will be downloaded into a container requires lots of resources to run.

You also provide the location of the image registry, which is the namespace `us.icr.io/${SN_ICR_NAMESPACE}` in the IBM Cloud Container Registry. You can also replace `${SN_ICR_NAMESPACE}` with your ICR namespace provided by Code Engine. Your ICR namespace can be seen in *Code Engine page -> project information*.

The container image will be tagged (named) as `myapp1`. You specify the `--registry-secret` option to access the registry with the `icr-secret`.

Submitting and running the build configuration

To submit a build run from a build configuration with the CLI that pulls the source from a local directory, use the `buildrun submit` command. This command requires the name of the build configuration and the path to your local source. Other optional arguments can be specified.

When you submit a build that pulls code from a local directory, your source code is packed into an archive file and uploaded to your IBM Cloud Container Registry instance. Note that you can only target the IBM Cloud Container Registry for your local builds. The source image is created in the same namespace as your build image.

Run the following command in your Code Engine CLI to submit and run the build configuration:

```
1. 1
2. 2
3. 3
4. 4
```

```
1. ibmcloud ce buildrun submit --name buildrun-local-dockerfile1 \
2.                             --build build-local-dockerfile1 \
3.                             --source .
4.                             /
```

Copied! Executed!

Submit the build run from the directory,`/myapp`, where your source code resides. The above command runs a build that is called `buildrun-local-dockerfile1` and uses the `build-local-dockerfile1` build configuration that you just created. The `--source` option specifies the path to the source on the local machine.

It will take ~3-5 minutes for all the steps in a buildrun to finish running.

After you run the command, it should take approximately 3-5 minutes for you to see the following message in the CLI:

buildrun

Note: In case you encounter below issue after executing the above command, you may still continue with the next steps and ignore the error message.

Buildrun

To monitor the progress of the buildrun, use the following command:

```
1
```

```
ibmcloud ce buildrun get -n buildrun-local-dockerfile1
```

Copied! Executed!

Once you see the status showing `Succeeded` (same as the following screenshot), that means your container image has been created successfully and pushed to the registry under your namespace.

Buildrun Succeeded

Now that the container image is ready, you need to pull the image from the Container Registry and deploy a containerized application using the image!

Deploying a Containerized app using Code Engine

In the previous step, you created and pushed the app's image to your namespace in the Container Registry. Let's deploy the app by referencing the `us.icr.io/${SN_ICR_NAMESPACE}/myapp1` image in the Container Registry.

Creating your application

Code Engine lets you deploy an application that uses an image stored in IBM Cloud Container Registry with the `ibmcloud app create` command.

Run the following command in your Code Engine CLI to deploy the app:

```
1
2
3
4
```

```
ibmcloud ce application create --name demo1 \
                              --image us.icr.io/${SN_ICR_NAMESPACE}/myapp1  \
                              --registry-secret icr-secret --es 2G \
                              --port 7860 --minscale 1
```

Copied! Executed!

After you run the command, it should take approximately 3-5 minutes for you to see the following message in the CLI:

DeployedApp

- The deployed app will be called `demo1`.
- It references (uses) the image `us.icr.io/sn-labs-xintongli/myapp1`.
- It uses `2G` of ephemeral storage in the container. For smaller applications, a default value for the ephemeral storage, i.e. `--es 400MB`, might be enough. You need 2G because of the size of the GPT2 model.

- You need to specify the port number `7860` for the application so that the public network connections and requests can be directed to the application when they arrive at the server.
- You set `--minscale 1` to make sure that your app will keep running even though there are no continuous requests. This is important because otherwise, you would need to wait for the app to start running every time you access its URL.

Accessing your application

This means your app has been deployed and you can access it now! To obtain the URL of your app, run `ibmcloud ce app get --name demo1 --output url`. Click on the URL returned, and you should be able to see your app running in your browser!

You can also create a custom domain and assign it to your app. For information about deploying an app with a custom domain through Cloudflare, see the Configuring a Custom Domain for Your Code Engine Application.

Conclusion

Congratulations on reaching this milestone!

In conclusion, you have successfully navigated through the intricate process of deploying an application on IBM Cloud using Gradio and IBM Code Engine. This journey began with understanding container images and containers, essential for isolating and running applications independently in diverse environments. You then embarked on the practical task of creating a container image, involving the assembly of key files such as `demo.py`, `requirements.txt`, and `Dockerfile`.

Testing these components locally ensured your application's smooth functionality before deploying it through Docker. Subsequently, you delved into the realms of IBM Code Engine, a robust platform that not only hosts your containerized workloads but also simplifies the deployment process by handling the underlying infrastructure complexities.

The final steps of your journey involved leveraging IBM Cloud's resources to build and push your container image to a registry, followed by deploying your application and making it accessible through a public URL. This comprehensive process, from local testing to cloud deployment, highlights the seamless integration of Gradio, Docker, and IBM Code Engine, culminating in a successful application deployment on IBM Cloud.

The skills and knowledge you have gained through this process are invaluable. You have mastered the art of deploying applications in a cloud environment, a vital competency in today's technology-driven world. Celebrate your achievement and look forward to leveraging these skills in your future projects and endeavors.

Author

Sina Nazeri

Connect with me on Linkedin.