Lab work

Scaling horizontally a web service, revisited

Many ways to invoke an action

Mathieu Bacou
mathieu.bacou@telecom-sudparis.eu

2022 – 2023
Télécom SudParis
Institut Mines-Télécom & Institut Polytechnique de Paris

Part I.
Introduction

1. Overview

Serverless cloud computing is particularly suited to build processing pipelines that are event-based, i.e., run when an event occurs. Examples are sentiment analysis on newly created micro-blogging messages, video recompression on newly uploaded content, or image conversion upon request from a user.

You will use your Apache OpenWhisk (OW) deployment on a Google Kubernetes Engine (GKE) cluster to implement a simple service of image conversion from JPEG to PDF: PDFMagic. This time, it will be built as a cloud function running on a serverless platform, allowing quick elasticity and rapid development, among other advantages.

Over the lab, you will build four different versions of the service:

0. manually invoked through OW's CLI, with the image as parameter;
1. manually invoked through a REST API, with the image as parameter;
2. manually invoked through a REST API, with the image identifier to fetch it from cloud storage as parameter;
3. automatically invoked by a trigger, with the image identifier to fetch it from cloud storage as parameter.

2. Prerequisite

Knowledge of using a terminal is assumed. A good understanding of the serverless notions seen during the lecture are welcome.

You will need a working deployment of Apache OpenWhisk, assumed to be living in Google Cloud Platform for integration with other services. You should have followed a previous lab to deploy and setup it.

The technical requirements are similar to the previous lab, i.e., you need configured `gcloud`[1] and `kubectl` commands, as well as OpenWhisk's CLI, `wsk`.

Alternatively, you may opt to use GCP's Cloud Shell (a shell embedded in GCP's web pages), where all the technical requirements are already fulfilled, and where you can also install `wsk`.

[1] The lab will give `gcloud` and `gsutil` commands to achieve a few tasks, but you may opt to go through the web interface at your own discretion.

Warning

You will work with a paid Google Kubernetes Engine (GKE) from the provider Google Cloud Platform. Underneath, it runs on nodes, i.e., virtual machines (VMs). The pricing model is that you pay for your VM resources as long as the VM, and thus the GKE node, is up, whatever its activity. Thus: *do not forget to destroy the GKE cluster at the end of the lab.*

Part II.
Setup

In this part, you fetch the required base files.

Jump into your work directory and execute the command in listing 1.

Listing 1: Download base files.

```
curl https://www-inf.telecom-sudparis.eu/COURS/CSC5004/practicals/scaling-revisited.tgz |\
    tar --extract --gzip
```

You should obtain two folders:

1. "pdfmagic_base64": files for versions 0 and 1;
2. "pdfmagic_bucket": files for versions 2 and 3.

Part III.
Version 0: manual invocation through CLI, image as parameter

In this part, you will build a first iteration of the PDFMagic action, and invoke it manually with OpenWhisk's CLI to test it. This will be the version 0 of the service PDFMagic.

3. Building the action

Change directory to "pdfmagic_base64". There, you will find:

"__main__.py" function code;
"params.json" parameters for the function, prepared for testing;
"image.jpg" test image for the function.

Information

The special name "__main__.py" – note the double underscores on each side – is Python specific, and indicates the entrypoint to an executable Python module. When built as an OpenWhisk action, the system will run its `main` function.

Information

You can look as "params.json" with `less`: the file is very long, because it includes the base64-encoded string that represents the binary data of the test JPEG image "image.jpg". Base64[a] is a binary-to-text encoding, where a character of the encoding represents 6 bits of binary data. It is used to include binary data as a text string.

[a] *Base64 - Wikipedia*. 2005. URL: `https://en.wikipedia.org/wiki/Base64` (visited on 12/04/2021).

Question 1

Read the first section of OpenWhisk's documentation on Python actions at `https://github.com/apache/openwhisk/blob/master/docs/actions-python.md`.

- Read the code in "__main__.py", and answer the questions:
  - What is the variable `args`?
  - How and under what form is the input image received?
  - Where is it stored ("physically"; think about how and where the function is executed) for conversion?
  - Where is the converted output stored?
  - How and under what form is the converted output sent back?
- What are the differences between the code of this Function-as-a-Service version, and the code of the version deployed as a pod in Kubernetes from a previous lab?

3.1. Packaging the action and language dependency

The action PDFMagic requires a Python dependency: delegator.py.[2] In the context of an OW action, language dependencies are included along with the action's code as a Zip archive. The way they are actually included inside the Zip archive is language-dependent, but most often involves using the language's dependency manager by running it from the Docker image of the language runtime provided by OW.

Information

In the Python world, dependencies are most often fetched from the public repository PyPI[a] using the tool pip. In addition, virtual environments ("virtualenvs") are used to scope the dependencies.

[a] *PyPI - The Python Package Index*. 2018. URL: `https://pypi.org/` (visited on 12/04/2021).

Coding task 1

Run a container from the image of OW's Python runtime to create a virtualenv containing the action's language dependency.

For this, you need to:

- run a container from the image "openwhisk/action-python-v3.9";
- run under the same user and group IDs as your user's on the host, to make the virtualenv owned by your user;[a]
  - look at the documentation of `docker run` to find the option,
  - you can get the ID of your user and group respectively with `id -u` and `id -g`;
- mount the current working directory under "/tmp" inside the container, to create the virtualenv there;
- change the image entrypoint to `bash`;

The commands you need to execute inside the container are given in listing 2. To execute them, you can use the flag `-c` of Bash to give it a command to execute instead of starting interactively.

[2] delegator.py allows delegating to a subprocess the execution of the conversion command. PyPI page: `https://pypi.org/project/delegator.py/`.

Listing 2: Make the virtualenv with the action's Python dependency.

```
cd tmp
virtualenv virtualenv
source virtualenv/bin/activate
pip install delegator.py
```

[a] In OW's runtime image, processes run as root, so the virtualenv would also be owned by root on the host without this setting.

The result of the above question is a "virtualenv" directory created in the current working directory. Package it with the action's code in a Zip archive as instructed in listing 3.

Listing 3: Package PDFMagic action with dependencies.

```
zip --recurse-paths pdfmagic_base64.zip __main__.py virtualenv
```

3.2. Customizing the language runtime image with a native dependency

In addition, the action PDFMagic requires a native dependency: ImageMagick.[3] OW actions run in Docker containers, which means that in order to add a dependency to the "native" runtime (i.e., outside the language ecosystem), you must customize OW's Docker runtime image.

Coding task 2

Write a Dockerfile to customize OW's Python runtime by adding ImageMagick. In other words:

- Write a Dockerfile from the Docker image of OW's Python runtime, that installs ImageMagick.
- Push the built image (refer to a previous lab to do so) to Docker Hub to make it publicly available.

Here is what you need to know, to write the Dockerfile:

- OW's Python runtime is named `openwhisk/action-python-v3.9`;
- it is based on the official Python Docker image, itself based on Ubuntu, where packages can be installed with:

[3] ImageMagick provides the conversion command `convert`.

Listing 4: Install packages in OW's Python runtime.

```
# In a Dockerfile, it is cleaner and more efficient to run all the
# commands in the same RUN instruction.
apt-get update
apt-get install -y PKGS...
# To make clean image layers, remove files produced as a side-effect of
# the commands.
rm -rf /var/lib/apt/lists/*
```

- the package you need to install is named `imagemagick`;
- for security reasons, ImageMagick prohibits converting images to PDF: to lift the restriction, your Dockerfile must also run the following command after installing ImageMagick:

Listing 5: Allow ImageMagick to convert to PDF.

```
# No need to understand it, but knowing sed is good for your culture!
sed -i '/disable␣ghostscript␣format␣types/,+6d'\
    /etc/ImageMagick-6/policy.xml
```

Information

If you remain stuck at this question, you may use this image in the following questions: `matbac/action-python-v3.9:imagemagick-pdf`.

3.3. Creating the action

Everything is now prepared, so create the action! Use the command `wsk` to interact with your OpenWhisk deployment.

Coding task 3

Create a package named "pdfmagic" in your user namespace.

Coding task 4

Create the action under the name "pdfmagic/v0", i.e., its name is "v0", and it is created under the package "pdfmagic".

In this case, don't forget that you must provide the customized Docker runtime image in addition to the archive of your function's code and its language dependencies.

Information

Specifying a Docker image is only needed when customizing the runtime image with native dependencies, like in your case here. When using the default OW's language runtime, but creating the action from a Zip archive, you must specify the action *kind* (flag `--kind`), i.e., its implementation language and version, to tell OW what runtime must be used to run it. Otherwise, OW is able to guess the kind from the code file.

Question 2

Why is it important to prepare as much of the action runtime as possible before actually invoking it? In other words, why is it preferable to package all the dependencies statically, and paying the development cost of a custom Docker runtime image, instead of using the generic OpenWhisk's runtime and installing dependencies upon invocation?

4. Testing the action

Coding task 5

Test your PDFMagic action: invoke it with the parameter file "params.json". Then, check that it ran correctly by checking the activations.

Information

Because of how the v0 of the function is coded, the output PDF file is included as a very long base64 string in the result of the invocation. You can run the command in listing 6 to extract the converted PDF to a local file "image.pdf". Open it to check that the conversion worked!

Listing 6: Check activation result of PDFMagic.

```
# tail is used to skip the first line, output by wsk but not part of the JSON
# activation data.
# jq is used to parse the JSON activation data to extract the PDF result.
# Replace the beginning of the pipeline with the command to get the activation.
COMMAND TO GET THE ACTIVATION RECORD |\
    tail -n+2 | jq -r '.response.result.pdf' | base64 -d > image.pdf
```

Information

If you are working on Google Cloud Shell, you cannot display the PDF. Verify that it is indeed a valid PDF file with the command `file PDF`; it should output:

```
image.pdf: PDF document, version 1.7, 1 pages
```

Congratulations! You successfully deployed an initial version of the service PDFMagic as a serverless cloud function.

For now, it can only be invoked through OW's control interface, i.e., with its CLI. In the next part, you will build version 1 of PDFMagic to invoke it through a REST API.

Part IV.
Version 1: manual invocation through REST API, image as parameter

In this part, you will setup PDFMagic to be accessible through a REST API. Thanks to your serverless platform, OpenWhisk, there is nothing server-like to deploy: the feature is provided by the platform.

Information

A REST API[a] is a web-based API that maps HTTP methods and URIs to services, transferring additional data in a standard format. In general, users and systems interact with web applications through this kind of APIs.

For example, sending a `GET` request to `/api/v1/animals` may be interpreted as requesting a list of animals, while sending a `POST` request to the same URL may be interpreted as adding an animal to the list, with the data provided in the request body.

[a] *Representational state transfer - Wikipedia*. 2018. URL: `https://en.wikipedia.org/wiki/REST_API` (visited on 12/04/2021).

To expose an action under an API endpoint, you must make it a web action.

Coding task 6

Modify the code to make PDFMagic a web action.

To help you, read OW's documentation on web actions at `https://github.com/apache/openwhisk/blob/master/docs/webactions.md`. You web action should:

- read the input image as a base64-encoded string from the HTTP request body;
- return the converted image as PDF content `application/pdf` in the HTTP response (similar to returning an `image/png` content in OW's documentation).

Make a new Zip archive (see listing 3) from your code and the virtualenv.

Coding task 7

Create the new version of the action under the name "pdfmagic/v1".

This time, the action *must be declared as a web action* when calling `wsk create`, to make OpenWhisk treat its input and output as HTTP query and response so it can be used behind an API endpoint.

Coding task 8

Create an API endpoint for the web action, under the base path "/convert" and the API path "/pdf".

Note that you must specify that your action returns an HTTP response (because your action specifies that the response's body is a PDF content).

Question 3

Justify your choice of API verb when creating the endpoint.

Make a note of the URL produced by OW in response.

Information

Specifying the API endpoint as two components (the base path and the API path) allows to have an application made of multiple API paths, each of them handling different aspects of it through different HTTP methods.

In addition, it is possible to create multiple API endpoints under the same API path, provided they use different API verbs.

Now to test: use `curl` to send the image to the URL responded by OW, with the command in listing 7. In the command, replace `URL` with the URL of the action's API endpoint, and `METHOD` with the API verb you selected when creating the API endpoint.

Listing 7: Test PDFMagic as a web action.

```
# --insecure tells curl not to validate the certificate because OpenWhisk uses a
# self-signed one.
curl --insecure --request METHOD --data-binary @image.jpg\
    --header "Content-type: image/jpeg" URL --output image.pdf
```

As previously, check the activations and the received PDF to verify that the service worked.

Congratulations! You implemented version 1 of your service PDFMagic, as a REST API endpoint.

This version is already representative of a real-world application. Nonetheless, in the next part, you will implement version 2, where the image is fetched from cloud storage in preparation to automation.

Part V.
Version 2: manual invocation through REST API, image in cloud storage

In this part, PDFMagic remains an endpoint to a REST API, however it fetches the image from a Google Cloud Storage bucket specified in its arguments.

Information

Google Cloud Storage is simply cloud storage, the most well known cloud service to the mass consumers. We will use it here as more professional users, setting up a bucket (like a storage volume) for the inputs of our application, and another bucket for the outputs. It is a paid product, but our usage will be so small that it will cost virtually nothing.

5. Setting up Google Cloud Storage buckets

As explained above, you must create two buckets in Google Cloud Storage for PDFMagic, as shown in listing 8.

Warning

In Google Cloud Storage (GCS), buckets are not namespaced, i.e., all buckets across GCS must have unique names. In your case, it is best to include a unique identifier of yours, such as your login name, etc.

Listing 8: Create buckets for PDFMagic.

```
# Replace NAME with your name, pseudonym or anything else unique and rememberable.
gsutil mb gs://NAME-convert-in
gsutil mb gs://NAME-convert-out
```

Then, you must request an API key to access the buckets. While you can access them manually with the command `gsutil` as above, your PDFMagic action cannot, and requires authentication information. You will see in section 6 how this authentication information is managed. For now, run the commands in listing 9. They will write the key to the JSON file "pdfmagic-key.json".

Listing 9: Create key for PDFMagic to access buckets.

```
# Find your service account name (the e-mail address).
gcloud iam service-accounts list
# Replace SERVICE_ACCOUNT with the e-mail from the output of the previous command.
gcloud iam service-accounts keys create pdfmagic-key.json \
    --iam-account=SERVICE_ACCOUNT
```

6. Building the action

Move to the folder "pdfmagic_bucket", where you will find the same files as before; however, the code was updated (and left with holes for you to fill) to accept parameters to fetch the input image from Google Cloud Storage. A new dependency is used: the Google Cloud SDK, to authenticate to the platform and use the cloud storage.

Coding task 9

Complete the holes in the code of "___main___.py".

To help you, here is the reference for the Google Cloud Storage SDK: `https://cloud.google.com/python/docs/reference/storage/lates`

`t` (and more precisely, the reference for the Google Cloud Storage client: `https://cloud.google.com/python/docs/reference/storage/latest/google.cloud.storage.client.Client`).

Question 4

What are the implications of including cloud provider-specific code in your application?

You must package the action as before, in a Zip archive. This time, new Python dependencies are required, to interact with Google services.

Coding task 10

As in the first coding task, create a virtualenv that includes two new dependencies: google-cloud-storage and google-auth.

Make a Zip archive (see listing 3) that includes your code and the virtualenv.

Coding task 11

Create the new version of the action under the name "pdfmagic/v2".
For this version, set default values to two parameters:

- `out-bucket` is always `NAME-convert-out` (from listing 8);
- `gcs-key` is always the content of the file "pdfmagic-key.json" that you created in listing 9.

Question 5

What is the purpose of setting those two parameters with default values?

Coding task 12

Create an API endpoint for the web action, under the base path "/convert" and the API path "/pdf".

Again, make a note of the URL produced by OW in response.

Question 6

- Justify your choice of API verb when creating the endpoint.
- What is the response type of this version?

7. Testing the action

To test version 2 of PDFMagic, first upload the test image to the input bucket, with the command in listing 10.

Listing 10: Upload test image to input bucket.

```
# Replace IN_BUCKET with the name of the input bucket chosen before.
gsutil cp image.jpg gs://IN_BUCKET/image.jpg
```

Now to test: use `curl` to send the required arguments to the URL responded by OW, with the command in listing 11, replacing the placeholders with their actual values.

Listing 11: Test PDFMagic as a web action with buckets.

```
curl --insecure --request GEMETHOD --data 'bucket=IN_BUCKET&name=image.jpg' URL
```

You should have received the names of the output bucket and file as a response. You can check the result with the command in listing 12.

Listing 12: Check the result of PDFMagic version 2.

```
# Replace OUT_BUCKET with the name of the output bucket in the response from above.
# Replace OUT_PDF with the name of the output file in the reponse from above.
gsutil cp gs://OUT_BUCKET/OUT_PDF image.pdf
# And then display or verify the file image.pdf.
```

Congratulations! You deployed your service with a link to cloud storage.

The execution of PDFMagic, although via a programmable REST API endpoint, remains manual: there is no automatic link between uploading a file, and execution the action on it.

Part VI.
Version 3: semi-automatic invocation with a trigger, image in cloud storage

In this part, you will setup a trigger and a rule to automatically invoke PDFMagic when the trigger is fired. In a fully automated system, the trigger would be fired by a webhook from Google Cloud Storage's, closing the final link of an automated conversion pipeline.[4]

Version 3 of PDFMagic actually reuses version 2 as-is.

Coding task 13

Create an OpenWhisk trigger, representing events of creation of new files in the input bucket.

Coding task 14

Create an OpenWhisk rule that ties the action "pdfmagic/v2" to the trigger.

Test your setup by manually firing the trigger, and then downloading the resulting image from the output bucket.

Question 7

What would you do to execute another action when the trigger is fired, in addition to running "pdfmagic/v2"?

Coding task 15

Bonus question: Write an action in the language of your choice[a], that displays its arguments to the standard output, and returns them; and tie

[4]Technical limitations currently make this impossible in the context of this lab.

it to the same trigger as PDFMagic from above.

[a] In a language that OpenWhisk supports; preferably NodeJS, Python or Java, see `https://github.com/apache/openwhisk/blob/master/docs/actions.md`.

Congratulations! You implemented PDFMagic as a semi-automatic pipeline to convert images, completely in the cloud, and serverless.

Part VII. Conclusion

The Google Kubernetes Engine cluster to which you deployed OpenWhisk, being deployed on Compute Engine VMs, will continue to consume credits.

Warning

If you don't need to use the cluster anymore, i.e., if you will not follow another lab on serverless using it, destroy the cluster! Also, destroy the input and output buckets to clean up, with `gsutil rb gs://BUCKET_NAME`.

In this lab, you iteratively implemented several versions of your image conversion service, PDFMagic. From a very manual and unpractical interface, you set it up as a semi-automatic pipeline that is easily invoked.

To go further, you may try to implement the same pipeline in the real Function-as-a-Service offering of Google Cloud, called Google Cloud Run. While the backend is different, based on another Function-as-a-Service platform called Knative[5], the concepts and features are similar; furthermore, the integration with other GCP features such as Storage will allow you to implement a completely automated pipeline, as suggested in the last part. The communication part of the pipeline, linking the storage tier to the compute tier to propagate events, is implemented by the Pub/Sub service of GCP, i.e., by a message-passing middleware where publishers can produce messages in topics that consumers subscribe to.

[5] *Home - Knative*. 2018. URL: `https://knative.dev/docs/` (visited on 12/04/2021).