

Impact Factor:

ISRA (India) = 3.117
ISI (Dubai, UAE) = 0.829
GIF (Australia) = 0.564
JIF = 1.500

SIS (USA) = 0.912
PIHHI (Russia) = 0.156
ESJI (KZ) = 8.716
SJIF (Morocco) = 5.667

ICV (Poland) = 6.630
PIF (India) = 1.940
IBI (India) = 4.260
OAJI (USA) = 0.350

SOI: [1.1/TAS](#) DOI: [10.15863/TAS](#)

International Scientific Journal Theoretical & Applied Science

p-ISSN: 2308-4944 (print) e-ISSN: 2409-0085 (online)

Year: 2019 Issue: 06 Volume: 74

Published: 17.06.2019 <http://T-Science.org>

QR – Issue



QR – Article



SECTION 4. Computer science, computer engineering and automation.

Oleg Dmitrievich Romanov

Peter the Great St. Petersburg Polytechnic University
Bachelor
oleromd@gmail.com

Oleg Yurievich Sabinin

Peter the Great St. Petersburg Polytechnic University
Candidate of technical sciences, Docent,
Department of Intellectual Sciences and Technology
olegsabinin@mail.ru

BUILDING A CONTAINER BASED APPLICATION AND SHIPPING IT TO GOOGLE CLOUD PLATFORM

Abstract: This article describes the process of building a container based application (concert ticket search service), deploying it into Google Kubernetes Engine, creating Cloud SQL instance and setting up a Virtual Private Cloud. We will cover in detail the steps how to build a small size docker image and push it to a docker registry. Also, we will compare image sizes with different build approaches on our application. After that we will focus on setting up a Cloud SQL for PostgreSQL and we will not forget to mention some limitations. And finally, we will describe what is VPC, how it can be basically configured and how to work with it.

Key words: google cloud platform, docker, kubernetes, cloud sql.

Language: English

Citation: Romanov, O. D., & Sabinin, O. Y. (2019). Building a container based application and shipping it to google cloud platform. *ISJ Theoretical & Applied Science*, 06 (74), 257-262.

Soi: <http://s-o-i.org/1.1/TAS-06-74-31> **Doi:**  <https://dx.doi.org/10.15863/TAS.2019.06.74.31>

Introduction

The cloud services receive more and more attention every day. And that's explainable:

- it provides different computing resources on-demand and self-service, requiring users to use simple interface to get the processing power, storage, and network they need;
- it is geographically wide, meaning that user can access the resources from any place;
- provider keeps huge pool of these resources and just gives some to the users, giving the win-win offer for customer and themselves;
- these resources are elastic;
- payment system is dedicated only for charging resources that are used;

It was also the definition of cloud.

In this article we are going to build an application and ship it to the GCP. Also, we will

mention the advantages and disadvantages of different steps of developing such application.

Concert ticket search service

We will use an application written in Go as an example. It's purpose is to find concert ticket which are stored in database. From bird's-eye it does the following:

- it has a TCP connection with other service which sends messages;
- it parses the message and tries to select some records from database;
- it completes some logic on extracted data and sends back information to another service.

We have to keep in mind that this particular application does not expose any ports. Instead, it establishes the connection with another service and works with.

Impact Factor:

ISRA (India) = 3.117	SIS (USA) = 0.912	ICV (Poland) = 6.630
ISI (Dubai, UAE) = 0.829	PIHHI (Russia) = 0.156	PIF (India) = 1.940
GIF (Australia) = 0.564	ESJI (KZ) = 8.716	IBI (India) = 4.260
JIF = 1.500	SJIF (Morocco) = 5.667	OAJI (USA) = 0.350

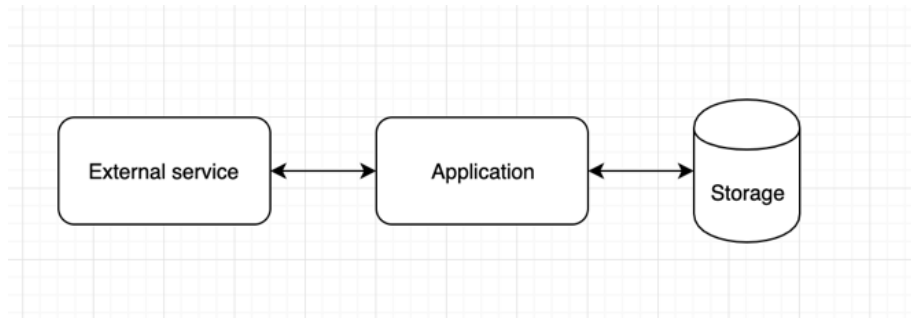


Figure 1 - Example service interactions.

As a conclusion, our application example:

- does not have any ports to be exposed;
- requires ability to connect to database;
- requires ability to connect to external service.

Building and pushing docker image

First of all, we have to put our application into Docker container. To do this we have to write a Dockerfile with required steps to do during build stage.

It is very important to keep built images compact. There is more efficient space usage on docker registry if image has small size. That also

means that machine (which runs the container) will not have so much space usage. The way to minimize image size is to keep only required files inside that image.

For example, we can copy all the directory with source code and build it then. At first glance everything is ok: we can run our application and it works. But what if we keep only binary file with application? We don't need keep our source files and some source files which were fetched as dependencies.

Let's get deep into the details of the Dockerfile mentioned on figure 2.

```
1 >> FROM golang:latest as builder
2
3 WORKDIR /application
4 ADD . .
5 RUN go build -o server ./cmd/server/server.go
6
7 FROM golang:latest
8 WORKDIR /application
9 COPY --from=builder /application/server ./server
10
11 CMD ["/server"]
```

Figure 2 – Dockerfile.

This technique is called multi-stage builds [1]. In this example we are defining two stages of the build. During docker image building each stage get processed. Each stage can get some files from previous stages.

On the first stage we are copying all the source files of our application and building it. On the 5th line of our Dockerfile we assume that resulting image would have all the source files and resulting binary. But since we are using multi-stage builds we can do

elegant move: we are copying only application binary file from the previous step.

Let's now compare the sizes of both images. First of all let's build them with the following tags: fat-application (for image with source files and binary) and small-application (for image only with binary file).

Image size inspection is mentioned on the figure 3.

```
(base) MacBookPro-ORomanov:~ ORomanov$ docker image inspect fat-application --format='{{.Size}}'
798374840
(base) MacBookPro-ORomanov:~ ORomanov$ docker image inspect small-application --format='{{.Size}}'
782278454
```

Figure 3 – Image size inspection.

As a result we received the following numbers:

Impact Factor:

ISRA (India) = 3.117	SIS (USA) = 0.912	ICV (Poland) = 6.630
ISI (Dubai, UAE) = 0.829	PIHHI (Russia) = 0.156	PIF (India) = 1.940
GIF (Australia) = 0.564	ESJI (KZ) = 8.716	IBI (India) = 4.260
JIF = 1.500	SJIF (Morocco) = 5.667	OAJI (USA) = 0.350

Table 1. Image size comparison

Image	Size, bytes
fat-application	798374840
small-application	782278454

The difference is almost 16 megabytes! That is very important especially if you expect that your image will be used as a base for someone else.

Our image can be pushed to any desired docker registry now via docker push command.

Creating Cloud SQL instance

We need to have a storage for our example application as it was mentioned before. This article is

dedicated to the usage of Google Cloud Platform and there is fully-managed database service, Cloud SQL [2]. Currently Cloud SQL can be used with MySQL or PostgreSQL. We will use PostgreSQL one in this article.

First of all we have to create one instance, it can be done with the help of Cloud Platform Console, a web user interface as shown on figure 4.

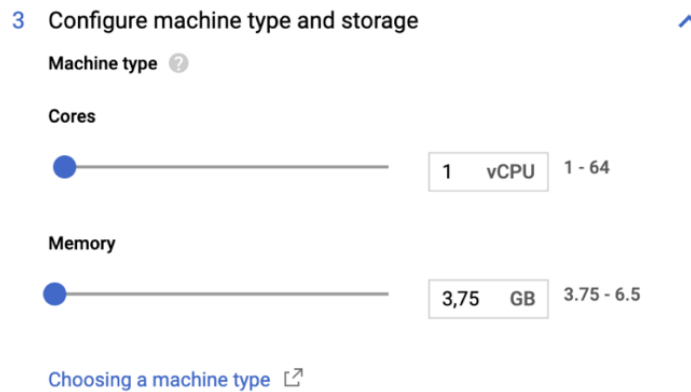


Figure 4 – Example of Cloud SQL instance configuration.

There are some parameters for creation, such as number of vCPU, number of memory, permanent storage type, its capacity and etc. We will create an instance with the smallest possible configuration.

Now we have instance with Public IP address. We can connect from anywhere we need. But firstly

we have to set up a whitelist (add addresses for establishing connection with them). It can be done on editing page of instance. Also Cloud SQL for PostgreSQL gives an ability to set different database flags.

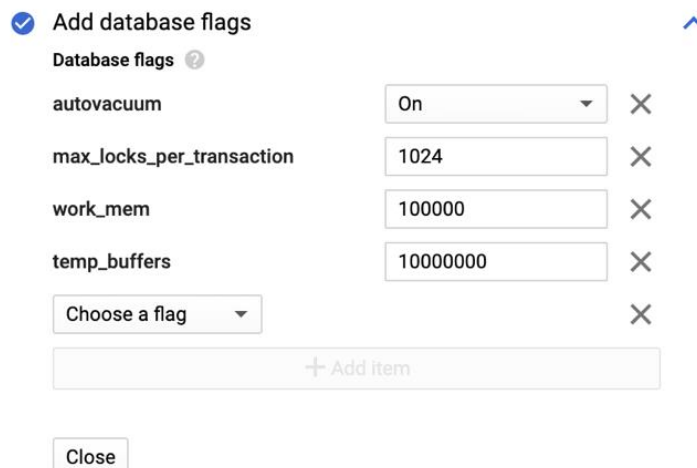


Figure 5 – PostgreSQL flags

Impact Factor:

ISRA (India) = 3.117	SIS (USA) = 0.912	ICV (Poland) = 6.630
ISI (Dubai, UAE) = 0.829	PIHHI (Russia) = 0.156	PIF (India) = 1.940
GIF (Australia) = 0.564	ESJI (KZ) = 8.716	IBI (India) = 4.260
JIF = 1.500	SJIF (Morocco) = 5.667	OAJI (USA) = 0.350

But Cloud SQL for PostgreSQL is different from typical PostgreSQL. Here are some limitations, which may cause troubles: not all of the database flags are supported, also there are only some of PostgreSQL extensions [3].

Now we can execute DDL script to create some objects which are required for our example application. And later we can connect to it.

Deploying the application

In our GCP project we can create a Google Kubernetes Engine cluster. We are going to do it via Cloud Platform Console as shown on figure 6.

Name ^	Location	Cluster size	Total cores	Total memory	Notifications	Labels
standard-cluster-1	us-central1-a			0.00 GB		

Figure 6 – Just creating instance

We can execute different kubectl commands to work with Kubernetes cluster manager [4].

First of all we have to create a namespace to place our application resources in there. To do so we have to apply the configuration shown on figure 7.

```
1  apiVersion: v1
2  kind: Namespace
3  metadata:
4    name: example-namespace
```

Figure 7 – Just creating instance

Since now we can work with Kubernetes cluster and place resources in example-namespace namespace

Let's now create a deployment configuration shown on figure 8 and examine its lines.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: application
5    namespace: example-namespace
6    labels:
7      project: article
8  spec:
9    replicas: 1
10   selector:
11     matchLabels:
12       project: article
13   template:
14     metadata:
15       labels:
16         project: article
17     spec:
18       containers:
19         - image: olerom/bot:latest
20           name: bot
21           resources:
22             requests:
23               memory: 100Mi
24               cpu: 1
25             limits:
26               memory: 100Mi
27               cpu: 1
```

Figure 8 – Deployment configuration

Deployment represents a desired state of a pod (which is a container or a group of containers). That

means that we have to describe this state in configuration and apply it. After that the deployment

Impact Factor:

ISRA (India) = 3.117	SIS (USA) = 0.912	ICV (Poland) = 6.630
ISI (Dubai, UAE) = 0.829	PIHHI (Russia) = 0.156	PIF (India) = 1.940
GIF (Australia) = 0.564	ESJI (KZ) = 8.716	IBI (India) = 4.260
JIF = 1.500	SJIF (Morocco) = 5.667	OAJI (USA) = 0.350

controller will provide declarative updates for this pod [5].

Let's get deeper in the provided configuration. The first line is the entrance point and definition of Kubernetes API version. After that we are defining object type and in this case it is deployment. It's time to define metadata of the deployment: its name, namespace and also some labels (key - value pairs). The next thing to do is to describe deployment specification:

- only one pod will be ran since replicas field is set to 1;
- we are defining labels in selector field to let deployment to know which pods it should control;
- finally, pod templates are defined.

The Pod specification determines how each Pod should look like: what applications should run inside its containers, which volumes the Pods should mount, its labels, and more [6].

Also we have defined the requested and limited resources. In our example we have requested 1 vCPU and 100Mi of memory [7]. And our requested values are equal to the limit ones.

After applying with the help of kubectl command this configuration our example application

will be started [8]. It will do all the business logic including interaction with external service and a database that is placed in Cloud SQL.

Virtual Private Cloud

As we have mentioned before, our Cloud SQL instance has Public IP. And the way our application can connect to is the following:

- add application IP address to the instance's whitelist;
- connect to the instance via internet.

But let's get into details and image what can be bad here:

- since we are connecting through the internet there may be some unwanted network latencies;
- public IP of the instance is exposed to the public internet, which may be a potential vulnerability.

GCP allows us to use Virtual Private Cloud (VPC) which can solve mentioned disadvantages [9].

VPC creation is shown on figure 9. We are doing it via Cloud Platform Console. We have to setup a subnet by mentioning some information such as used GCP region, address range for a subnet, enabling or disabling logger and a route mode.

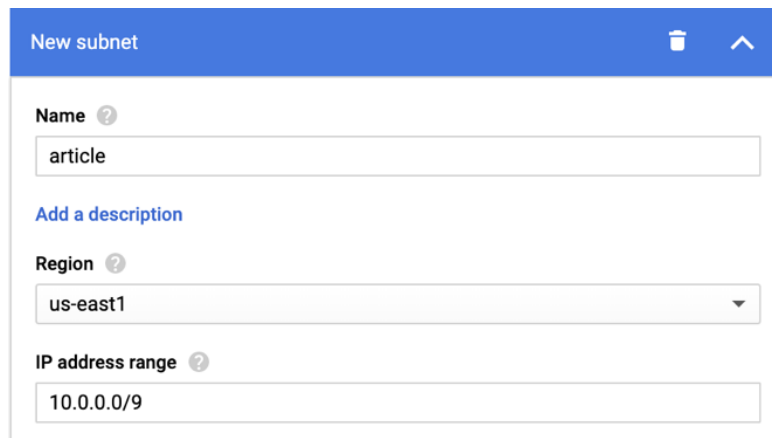


Figure 9 – VPC creation

By doing these steps we have created a VPC network. Now we can enable its support on our kubernetes cluster and Cloud SQL instance.

But it is also important to mention the noticeable limitation: once Cloud SQL instance enables private IP usage with VPC it cannot be disabled.

Let's get a summary over VPC usage:

- both GKE and Cloud SQL supports just created VPC network;
- our example application connects to the PostgreSQL via VPC;
- the connection from our application to Cloud SQL instance is secured;
- nothing is exposed to the public internet from Cloud SQL instance;

- there is a variety of abilities to setup different rules to be used later [10].

Conclusion

We have shown the way to build a small size docker image of our example application. Also we have compared the actual size of both images: fat-application and small-application. We've made a conclusion that difference is important and may be essential in some cases.

After that we have setup a PostgreSQL instance in Cloud SQL, showing that it is fully managed and ready-to-use. But we have also mentioned some limitations and disadvantages of this solution.

Next thing to do was kubernetes cluster creation in GKE. We have created the cluster and the

Impact Factor:

ISRA (India) = 3.117	SIS (USA) = 0.912	ICV (Poland) = 6.630
ISI (Dubai, UAE) = 0.829	PIHHI (Russia) = 0.156	PIF (India) = 1.940
GIF (Australia) = 0.564	ESJI (KZ) = 8.716	IBI (India) = 4.260
JIF = 1.500	SJIF (Morocco) = 5.667	OAJI (USA) = 0.350

namespace. After that we have provided the deployment which controls the required pod with some labels. We have also mentioned the resource limits and requests and did not forget to use them in configuration.

Finally, we have described how VPC can be created and used in our application. Although, we have shown a limitation of its usage with Cloud SQL.

References:

1. (n.d.). Use multi-stage builds. Retrieved June 13, 2019, from <https://docs.docker.com/develop/develop-images/multistage-build/>
2. (n.d.). Cloud SQL documentation. Retrieved June 13, 2019, from <https://cloud.google.com/sql/docs/>
3. (n.d.). PostgreSQL extensions. Retrieved June 13, 2019, from <https://cloud.google.com/sql/docs/postgres/extensions>
4. (n.d.). Kubernetes References. Retrieved June 13, 2019, from <https://kubernetes.io/docs/reference/>
5. (n.d.). Deployments documentation. Retrieved June 13, 2019, from <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>
6. (n.d.). Kubernetes Engine: Deployment. Retrieved June 13, 2019, from <https://cloud.google.com/kubernetes-engine/docs/concepts/deployment>
7. (n.d.). Managing Compute Resources for Containers. Retrieved June 13, 2019, from <https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/>
8. (n.d.). Kubectl Reference Docs. Retrieved June 13, 2019, from <https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands>
9. (n.d.). Cloud SQL: Private IP. Retrieved June 13, 2019, from <https://cloud.google.com/sql/docs/mysql/private-ip>
10. (n.d.). Virtual Private Cloud (VPC) Network Overview. Retrieved June 13, 2019, from <https://cloud.google.com/vpc/docs/vpc>