

Devops & Continuous Deployment

A project presented by the SuperCool team: Nicolas Blanchard, Faustin Dewas,
Guillaume Theret, Benjamin Lesieux, Martin Gayet and Julien Hassoun

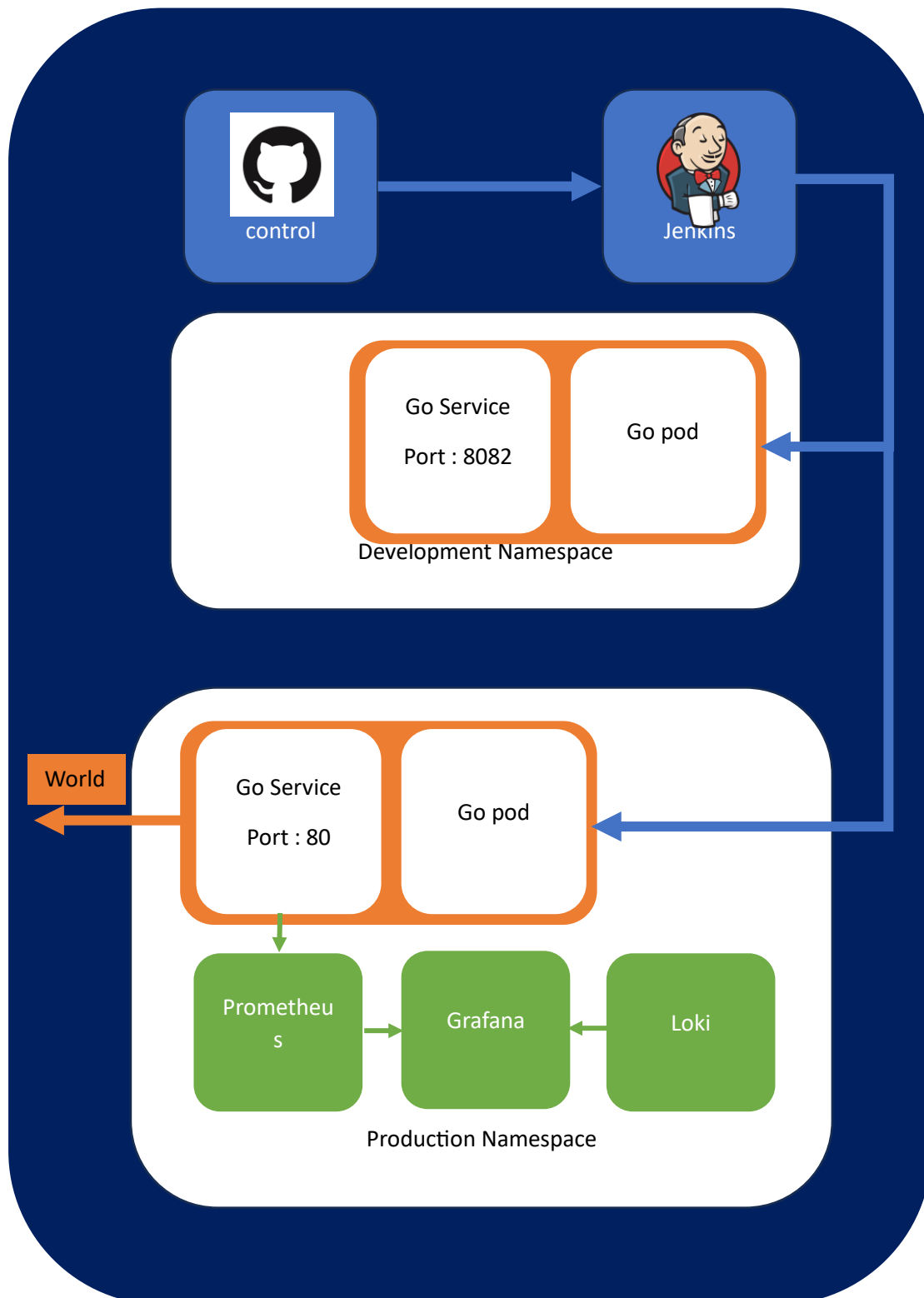
Contents

I.	Diagram of the solution	2
II.	Modification of the application	3
III.	Docker Image	3
IV.	Kubernetes and Helm	3
A.	Configuration of the application.....	3
B.	Helm repository.....	4
C.	Creation of the Kubernetes user “Jenkins”.....	4
V.	Jenkins	4
A.	Jenkins configuration.....	4
B.	Pipeline Configuration	5
C.	Basics stages	5
D.	Overall structure of the pipeline	6
E.	Results	6
VI.	Prometheus	7
A.	Target.....	7
	7
B.	Rules	8
VII.	Grafana	9
A.	Prometheus datasource	9
B.	Loki Datasource	10
C.	Creation of a Dashboard.....	11

Note: If you want to deploy locally everything as we did, please take a look at the README file located in the root of the project.

Github project URL: <https://github.com/Faust1-2/ST2DCE-PRJ>

I. Diagram of the solution

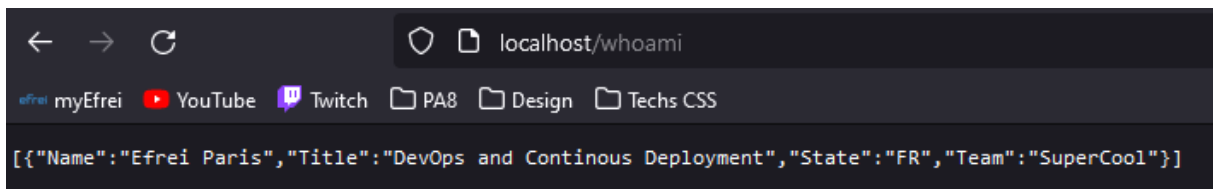


II. Modification of the application

The first thing to do in this project was to update the application endpoint “whoami” in order to have the name of our team, “Supercool”, displayed.

We did it via the following code:

```
func whoAmI(response http.ResponseWriter, r *http.Request) {
    who := []whoami{
        whoami{Name: "Efrei Paris",
            Title: "DevOps and Continous Deployment",
            State: "FR",
            Team: "SuperCool",
        },
    }
    json.NewEncoder(response).Encode(who)
    fmt.Println("Endpoint Hit", who)
}
```



Now, let's follow the work of the Supercool team in this devops project.

III. Docker Image

We were provided a go application, which we had to dockerize and modify in order to have a easily deployable application.

It simply expose the application on the port 8080 and run the main go file. We will have the occasion to take a look at the application in the **Prometheus part**.

```
FROM golang:latest
WORKDIR /app
COPY ./webapi ./
RUN go mod download
EXPOSE 8080
CMD ["go", "run", "main.go"]
```

IV. Kubernetes and Helm

A. Configuration of the application

For our application to work and be exposed on one of our desired port, we will need two Kubernetes resources:

- A **deployment**: It will deploy x containers, known as “pods”, in our Kubernetes cluster. The x is the number of replicas we want.
- A **service** : It will be the tool which will permit us to access the application. The type of service we will use is a “LoadBalancer”.

These resources are created via two YAML files, **supercool.yml** for our deployment and **supercool-service.yml** for our service. But now, we have to simplify the way to change the configuration values at each deployment for our Jenkins deployments. For this, we will use Helm.

B. Helm repository

Helm is a tool able to manage Kubernetes applications. It can either download some preconfigured Kubernetes repositories, or create custom charts.

With a simple command, **helm create**, we create a deployment folder in which we can now put our previously created configuration files. Then we just had a bit of work to do in order to make the desired configuration values editable and gave them some default values with a **values.yml** file.

Now, with the command **helm install deployment/**, we have now an easily editable Kubernetes deployment for our application. We can easily modify a value with the **--set** flag and the correct field.

C. Creation of the Kubernetes user "Jenkins"

Since this project is meant to be deployed via Jenkins, we need to have a Kubernetes account with the needed authorizations to do what we want. So we created with the **serviceAccount.yml** file a service account named **Jenkins**, that can do every operation on the development and production namespaces. With it, we also created a "secret", named **Jenkins-secret**, that will permit Jenkins to log in to the Kubernetes cluster when it needs.

V. Jenkins

We used Jenkins as an application to run deployment tests and if they were successful, to deploy our application to the namespace production. You can find it on our github repository, so we won't give the whole code in this report.

A. Jenkins configuration

Kubernetes API Plugin

Since we are using Jenkins and Kubernetes on the same machine, and want Jenkins to deploy the application on the local machine, we simply have to ensure that it understands Kubernetes commands and have access to the Kubernetes cluster.

In order to do this, we installed the Kubernetes API plugin. We will show how it works in the stages part of this report.

But we need one more thing in order to make the Kubernetes API plugin work.

Kubernetes credentials

We created a service account named Jenkins with various access to the development and production namespaces. Now, we need its authentication token in order to let Jenkins use properly our Kubernetes environment.

For that, we will retrieve the token account (command is located in the README file), and then create a Jenkins credential as a **Secret text**. This secret text will be used by Kubernetes API to access the Kubernetes cluster.

B. Pipeline Configuration

We decided to create a multi-branch pipeline. It has various advantages, which include having different properties on the branches of our project. It also comes with a default GitHub project configuration.

C. Basics stages

We started by creating our docker image, and tagged it as the build_id of Jenkins. It is a id that is created and autoincremented by Jenkins, so we have nothing to do with it !

```
stage('build docker image') {  
    sh 'docker build -t supercool-docker:${BUILD_ID} .'}
```

Now, let's take a look at how we will deploy our application with Kubernetes. Since we created a helm repository for the project, it is quite simple to modify.

```
stage('Deploy to development') {  
    withKubeConfig([  
        credentialsId: 'kubernetes-credentials',  
        serverUrl: 'https://localhost:6443'  
    ]) {  
        sh 'helm install development-${BUILD_ID} deployment/ --set  
image.tag=${BUILD_ID} --set metadata.namespace=development --set  
service.port=8082 -n development'  
    }  
}
```

This stage is a bit more complex to understand. Here, we are using the Kubernetes API plugin we just downloaded with the **withKubeConfig** function. It takes the Kubernetes credentials we created and the url of the server. Since we are working with Kubernetes and Jenkins on the same machine, in this case, the server URL is **https://localhost:6443**.

Then, we are simply using a helm command to deploy to the development namespace our application. We are providing the newly created image, and the port of the tested application is 8082, because we don't want it to take the current port of the production application. We also give the deployment name "development-\${BUILD_ID}" in order to have a basic way to find it in case of pipeline errors.

The next stage of our pipeline is the deployment check. This is a basic test that will simply curl the application and see the HTML status code.

```
final String status = sh(script: 'curl --silent --output /dev/null --  
write-out "%{http_code}" localhost:8082', returnStdout: true)
```

If the status code is 200, we will deploy the application in production by using the **helm upgrade – install** command. It is a command that update the deployment or create it if it does not exists yet.

More precisely, we use the command:

```
helm upgrade --install production deployment/ --set image.tag=${BUILD_ID} --set
metadata.namespace=production -n production
```

D. Overall structure of the pipeline

Our pipeline is structured as a try-catch function, that throws an error if the development application fails to start. It also contains a **finally** enclosure to delete the deployment of the development deployment.

```
node {
    // checkout
    try {
        // build docker image
        // deploy development application
        // test development application
        If (status == 200) {
            // deploy to production
        } else {
            // throw an error
        }
    } catch {
        // notify that the deployment failed
    } finally {
        // clean development production
    }
}
```

Now, we have an easily deployable application thanks to helm, that is deployed automatically by Jenkins at each repository scan.

E. Results

✓ main

Full project name: ST2DCE-PRJ/main

Stage View

	checkout	build docker image	Deploy to development	Check deployment status	Deploy to production	cleanup Development
Average stage times: (Average full run time: ~16s)	1s	1s	433ms	10s	437ms	732ms
Feb. 29 01:21 No Changes	1s	1s	432ms	10s	442ms	585ms
Feb. 29 00:50 No Changes	984ms	1s	435ms	10s	432ms	880ms

We have a fast Jenkins pipeline that build an docker image, deploy a test Kubernetes deployment in the development namespace, and then deploy it in the production namespace on success, and finally clean the tests.

VI. Prometheus

Prometheus is a monitoring tool. We will use it to get metrics from our application during its runtime, and check that everything goes as expected. We will simply deploy it with a YAML file using Kubernetes, which will load the Prometheus image and expose it to the cluster with a service.

A. Target

The first thing to do when defining a target for Prometheus is to create a `/metrics` endpoint on our application. This endpoint will be exploited by Prometheus to retrieve the different metrics. Many programming languages are supported by Prometheus, but it requires the developer to add some dependencies to the application.

In our case, we had to add a Prometheus package to our go application, and exposed the required functions in the `/metrics` endpoint.

```
func request1() {  
    ...  
    http.Handle("/metrics", promhttp.Handler())  
    ...  
}
```

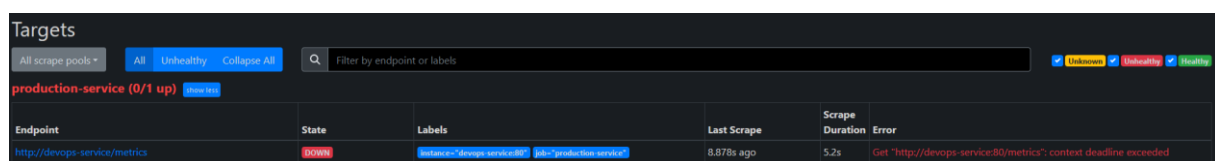
Then, we have to add a configuration to our Prometheus application. Since we are deploying Prometheus with Kubernetes, we will have to create a Kubernetes ConfigMap and volumes to provide the necessary configuration to Prometheus.

```
scrape_configs:  
- job_name: 'production-service'  
  scrape_interval: 5s  
  static_configs:  
  - targets: ['devops-service:80']  
rule_files:  
- /etc/prometheus/alerts/alert_rules.yml
```

This configuration here will first define our `scrape_configs`: they are the services observed by Prometheus. In our case, this is our deployed application in the namespace `production`.

We defined our target by providing an URL to Prometheus. In our case, we want it to be **devops-service:80**, because **devops-service** is the name of the service exposing our application and **80** is the port the service is exposing our application to. We are not using localhost this time because Kubernetes has an intern DNS. It permits every service in the namespace to access each other with their names and ports. It also means that this configuration will work in every case, if Prometheus is deployed on the same namespace as the watched application.

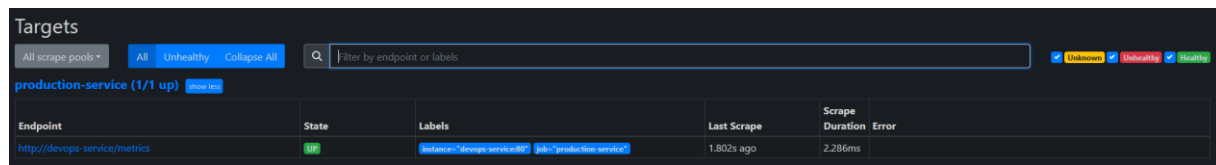
Let's now check if the configuration is working properly. When no deployment has been released, we have the following view:



The screenshot shows the Prometheus Targets page. At the top, there's a header with 'All scrape pools', 'All', 'Unhealthy', and 'Collapse All'. Below that is a search bar 'Filter by endpoint or labels' and three status filters: 'Unknown', 'Unhealthy', and 'Healthy'. The main content area shows a single target 'production-service (0/1 up)' with a 'down' status. Below this is a table with columns: Endpoint, State, Labels, Last Scrape, Scrape Duration, and Error. The table contains one row for 'http://devops-service/metrics' with state 'down', labels 'instance=devops-service' and 'job=production-service', last scrape '8.878s ago', duration '5.2s', and an error message 'Get "http://devops-service:80/metrics": context deadline exceeded'.

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://devops-service/metrics	down	instance=devops-service job=production-service	8.878s ago	5.2s	Get "http://devops-service:80/metrics": context deadline exceeded

Let's deploy our application and see what happens next.



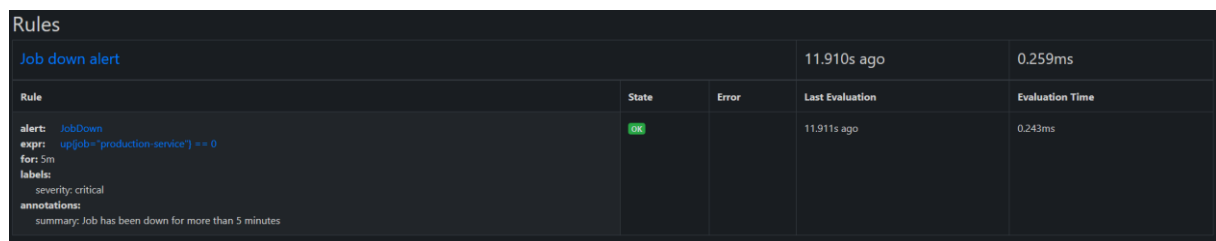
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://devops-service/metrics	UP	instance="devops-service-001" job="production-service"	1.802s ago	2.286ms	

So, it seems that Prometheus is correctly detecting and retrieving metrics from the application.

B. Rules

In our configuration file, we also provided the path to a rule file. In this project, we created one to check if the application is down for more than 5 minutes. We used the exam same ConfigMap to create it, and you can find it in the **Prometheus.yml** file of our project.

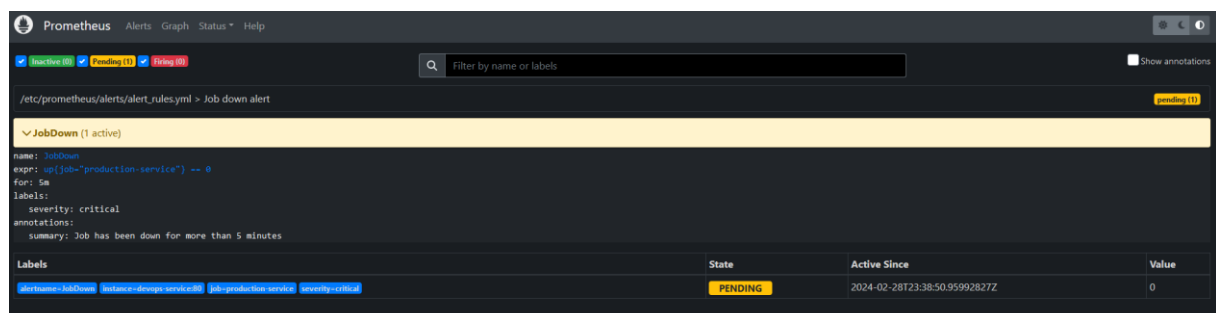
Let's head to localhost:9090 to check our configuration. If we go to status > Rules, we have the following result:



Rule	State	Error	Last Evaluation	Evaluation Time
Rule alert: JobDown expr: up{job="production-service"} == 0 for: 5m labels: severity: critical annotations: summary: Job has been down for more than 5 minutes	OK		11.911s ago	0.243ms

Meaning that our rule has been correctly added. It will test each minute if the application is working correctly. Let's now try it. We will simply remove the deployment from the cluster with the command **helm uninstall production -n production**. It will not create any problem for the rest of the project.

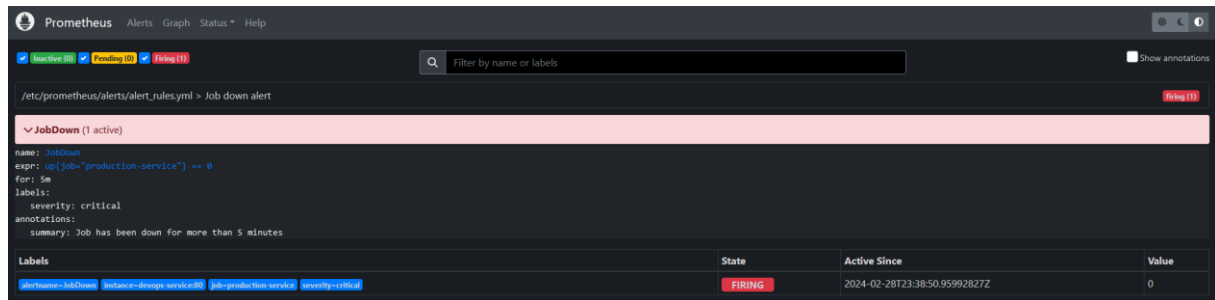
First of all, we shall go to the "Alert" section of Prometheus. As soon as we shut down our application, we get the following result:



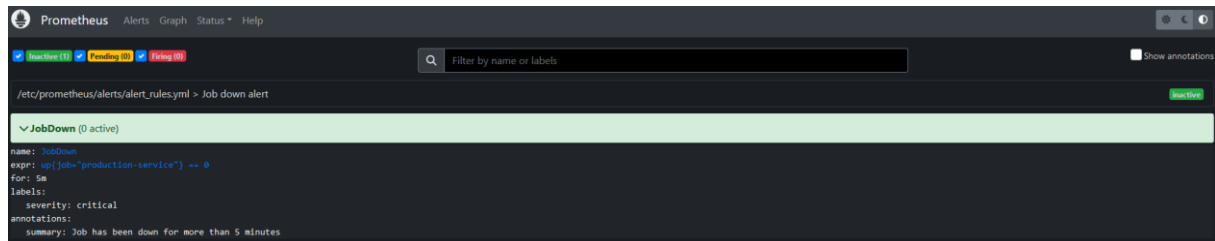
Labels	State	Active Since	Value
alertname: JobDown instance: devops-service-001 job: production-service severity: critical	PENDING	2024-02-28T23:38:50.95992827Z	0

Which means that Prometheus detected the application was down, but because it has not been 5 minutes, the alert is "pending".

Once the 5 minutes have passed, we get the following screen:



Meaning that our alert is perfectly working. When deploying the app back, the alert is gone:



VII. Grafana

Prometheus is a good tool to retrieve metrics for the application, but it is not enough. We might need more data, from different metrics at different moments, and even from our cluster. In order to group all of this data in the same place, we will use Grafana.

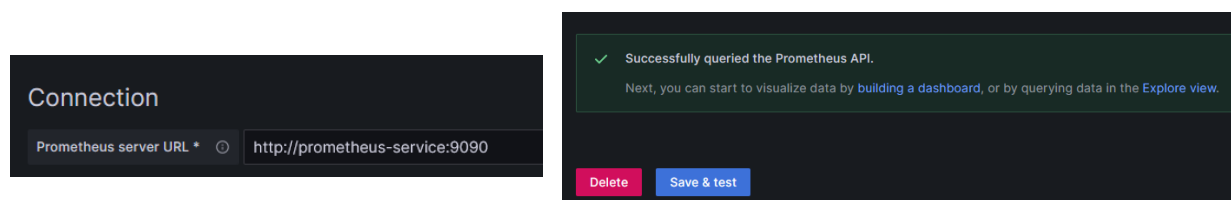
In this case, we will also deploy Grafana as a Kubernetes application, using the helm repo Grafana/Grafana. It will be located in the production namespace, and exposed with a LoadBalancer service.

When deploying Grafana for the first time, it says that we have to retrieve the password of the admin user. Do not forget to retrieve it !

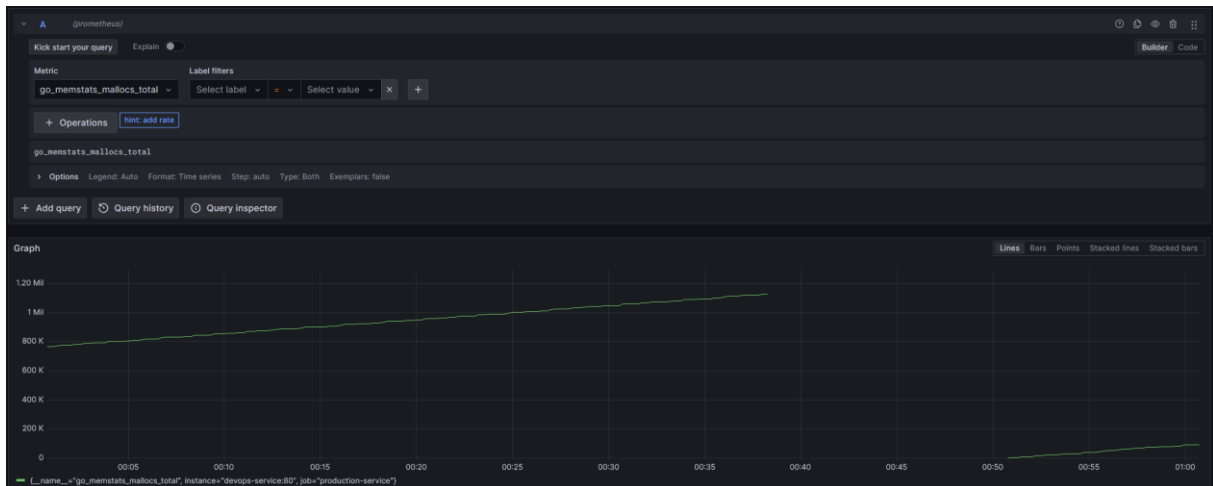
A. Prometheus datasource

Grafana does not retrieve data by itself: it needs datasources. A lot of datasources are available for Grafana, but for now, the one we need is Prometheus.

When adding a datasource, we select Prometheus, and since Grafana and Prometheus are both deployed within the same namespace, we simply have change one line in order to configure the datasource.



With this simple configuration, grafana is ready to fetch metrics from our applications using the prometheus application we created before. Let's try it in the explore view of the application:



It is working perfectly !

B. Loki Datasource

We also want to have a loki datasource set up in our Grafana. Let's create one with the basic command of :

helm install loki grafana/loki-stack --set grafana.enabled=false --namespace=production

It will use the loki-stack of the helm Grafana repository and disable the Grafana deployment given with it by default.

Now, let's add a new datasource, of type Loki this time:

The image shows the 'Connection' form in Grafana for adding a new datasource. The 'URL' field is set to `http://loki:3100`.

And test it in the explorer view:

The image shows the Grafana Explorer view with a new query added. The query is `{namespace="production"} |= `level=error``. The label filters are `namespace=production` and `level=error`. The query type is set to 'Range' and the line limit is 1000.

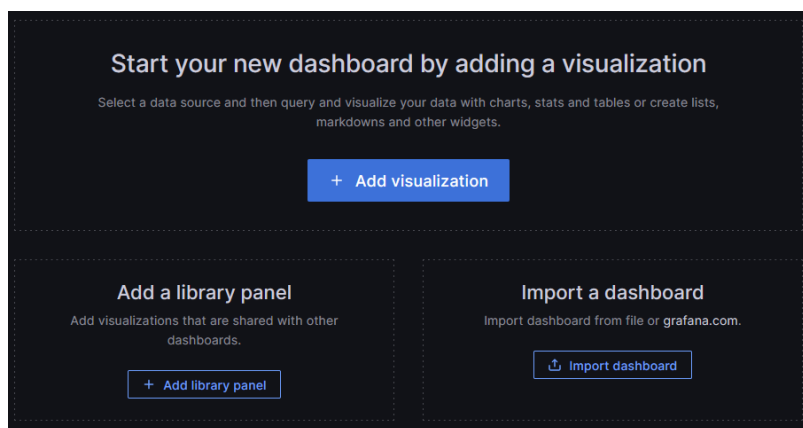
This query will help us get all the errors from the namespace "production".

```
> 2024-02-29 01:05:10.856 ("log" "level:error" ts=2024-02-29T00:05:10.82643064Z caller:stats.go:57 component=frontend msg="failed to record query metrics" err="expected one of the *LokiRequest, *LokiInstantRequest, *LokiSeriesRequest, *Loki...
  subNameRequest, got "\n", stream="stderr", "time":"2024-02-29T00:05:10.82666282Z")
> 2024-02-29 01:05:10.855 ("log" "level:error" ts=2024-02-29T00:05:10.826410514Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826661475Z")
> 2024-02-29 01:05:10.854 ("log" "level:error" ts=2024-02-29T00:05:10.826412332Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826660104Z")
> 2024-02-29 01:05:10.853 ("log" "level:error" ts=2024-02-29T00:05:10.826403971Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826658832Z")
> 2024-02-29 01:05:10.853 ("log" "level:error" ts=2024-02-29T00:05:10.826405966Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826657566Z")
> 2024-02-29 01:05:10.853 ("log" "level:error" ts=2024-02-29T00:05:10.826404991Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826656185Z")
> 2024-02-29 01:05:10.852 ("log" "level:error" ts=2024-02-29T00:05:10.826403692Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826654802Z")
> 2024-02-29 01:05:10.852 ("log" "level:error" ts=2024-02-29T00:05:10.826402637Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826653558Z")
> 2024-02-29 01:05:10.852 ("log" "level:error" ts=2024-02-29T00:05:10.826401596Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826652141Z")
> 2024-02-29 01:05:10.852 ("log" "level:error" ts=2024-02-29T00:05:10.826400722Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826650837Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.8263995148Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826649532Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826400472Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826648166Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826397018Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826646838Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826396535Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826645608Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826393802Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826643766Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826392252Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826642392Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826391022Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826640942Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826393732Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826639683Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826391530Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826629388Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826387074Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826627994Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826373329Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826626589Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826368604Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826624782Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826313686Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826613922Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826302301Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826551245Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826370436Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826549911Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826377133Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826548585Z")
> 2024-02-29 01:05:10.851 ("log" "level:error" ts=2024-02-29T00:05:10.826373852Z caller:retry.go:73 org_id=fake msg="error processing request" try=0 err="context canceled"\n", stream="stderr", "time":"2024-02-29T00:05:10.826547251Z")
```

C. Creation of a Dashboard

Now that we have set up our datasources, we can now create a dashboard. We can see it as an aggregation of explorer views.

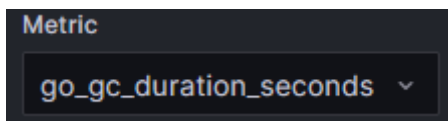
When creating a new dashboard, we will get the following view:



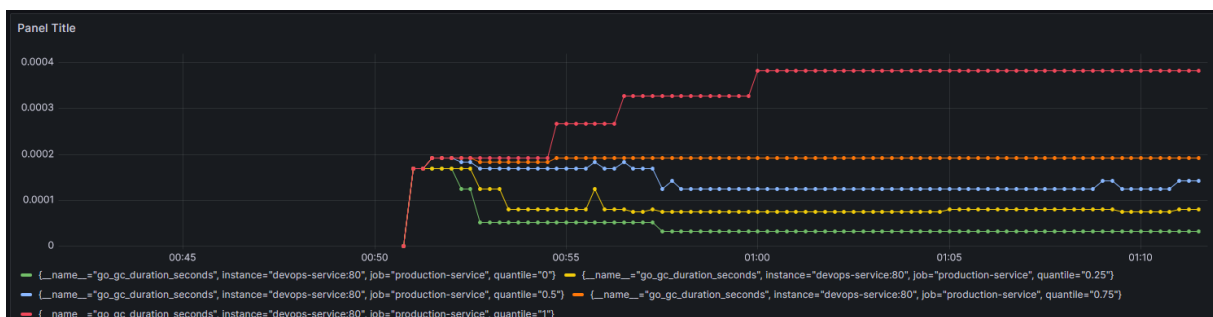
Let's start by adding a visualization.

We will start with a Prometheus visualization.

We want to get the duration of the go application. We can do that by selecting the metric:

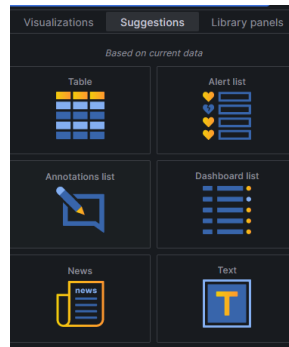


And we will have the following result. We can modify how the data is shown, but since it is a duration, a time series representation is adapted.



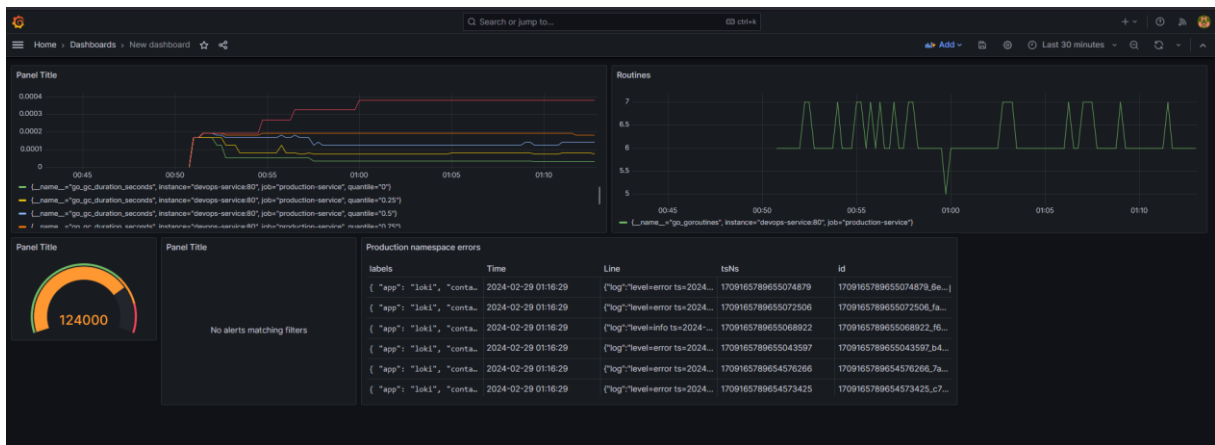
Let's save it and keep adding more data visualizations to our application.

We can also create an alert visualization from the Prometheus datasource, by simply selecting “alert list”:



We can then do the same thing with loki, by adding a error fetching table. To do this, we will just add, just like with the explorer view, the selector “error”.

Finally, we end up with the following dashboard:



And when the deployment is down:

