Build to Manage - Node.js Observability labs

During this lab we will instrument a simple Node.js application for logging in order to use with log analytics tools like Elastic stack and Humio as well as with metrics for monitoring with Prometheus and Grafana.

Instrumentation of the application code with metrics, logging and tracing is part of the general concept we call **Build to Manage**. It specifies the practice of activities developers can do in order to provide manageability aspects as part of an application release.

Objectives

- Lab 1: Node.js logging with Winston and ELK stack
- Lab 2: Node.js logging with Winston and Humio
- Lab 3: Node.js metrics instrumentation and monitoring with Prometheus and Grafana
- Lab 4: Configure application monitoring with Prometheus on Openshift
- Distributed tracing labs

Prerequisites

Install the following software on your workstation. You may use your laptop for all the labs, but probably a better idea is to use a clean Linux VM.
If you have access and would like to use Fyre, I'd recommend to deploy Ubuntu 20.04 ember with 8 core CPU and 16 GB RAM (for this setup, installation of all prerequisites is as easy as `apt install docker-compose`).

- Docker for Desktop
- Docker Compose
- Openshift `oc` CLI (optional)
- `curl`

Clone the following repository from GitHub.

```
git clone https://github.com/rafal-szypulka/b2m-nodejs-v2
```

Most of the commands should be executed from the `b2m-nodejs-v2/lab-x` directory:

> The solution to the lab is located in the directory b2m-nodejs-v2/lab-x/solution

> Login to Docker Hub using `docker login` in order to avoid the problems with recently introduced limits.

Logging

A production service should have both logging and monitoring. Monitoring provides a real-time and historical view on the system and application state, and alerts you in case a situation is met. In most cases, a monitoring alert is simply a trigger for you to start an investigation. Monitoring shows the symptoms of problems. Logs provide details and state on individual transactions, so you can fully understand the cause of problems.

Logs provide visibility into the behavior of a running app, they are one of the most fundamental tools for debugging and finding issues within your application. If structured correctly, logs can contain a wealth of information about a specific event. Logs can tell us not only when the event took place, but also provide us with details as to the root cause. Therefore, it is important that the log entries are readable to humans and machines.

According to the 12-factor application guidelines, logs are the stream of aggregated, time-ordered events. A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage log files. Instead, each running process writes its event stream, unbuffered, to stdout. If you deviate from these guidelines, make sure that you address the operational needs for log files, such as logging to local files and applying log rotation policies.

Lab 1 - Node.js app logging with Elastic stack

Deploy a local Elastic stack with Docker Compose

During this lab we will run the Elastic Stack (Elasticsearch, Logstash, Kibana) in docker-compose.
The ELK configuration in this lab is based on https://github.com/deviantony/docker-elk (6.8.x branch).

Briefly review the simple Logstash configuration we will use for this lab: `lab-1/logstash/pipeline/logstash.conf`:

```
input {
    gelf { port => 5000 }
}

filter {
    json { source => "message" }
    #we need level field in a numeric format
    mutate {
     gsub => [
      "level", "info", 6,
      "level", "error", 3
     ]
    }
    mutate {
     convert => { "level" => "integer" }
    }
}
```

```
output {
    elasticsearch {
        hosts => "elasticsearch:9200"
        user => "elastic"
        password => "changeme"
    }
    stdout { codec => rubydebug }
}
```

The above will configure Logstash input to use `gelf` (Graylog Extended Log Format) protocol supported by Docker log driver, so we can directly stream application logs from the app running in Docker container to Logstash using `gelf` protocol. JSON formatted app log message is extracted from the field `message` and parsed to named fields. After parsing and conversion the log steam is sent to elasticsearch.

1). Start the Elastic stack:

```
cd b2m-nodejs-v2/lab-1
docker-compose build
docker-compose up -d
```

> Note, it may take a while for the first time, because it will download the ELK images from DockerHub and build an image for our Node.js application.

2). While waiting for containers, review the configuration of our logging lab.

- `b2m-nodejs-v2/lab-1/docker-compose.yaml` - this is the main config file for docker-compose stack which specifies all options for all containers in the stack.
- `b2m-nodejs-v2/lab-1/app/server.js` - the source code of our sample Node.js application.
- `b2m-nodejs-v2/lab-1/app/Dockerfile` - this file is used to build your app docker image.

3). After the `docker-compose` completed the startup, verify you can access Kibana on `http://<your-hostname>:5601`.

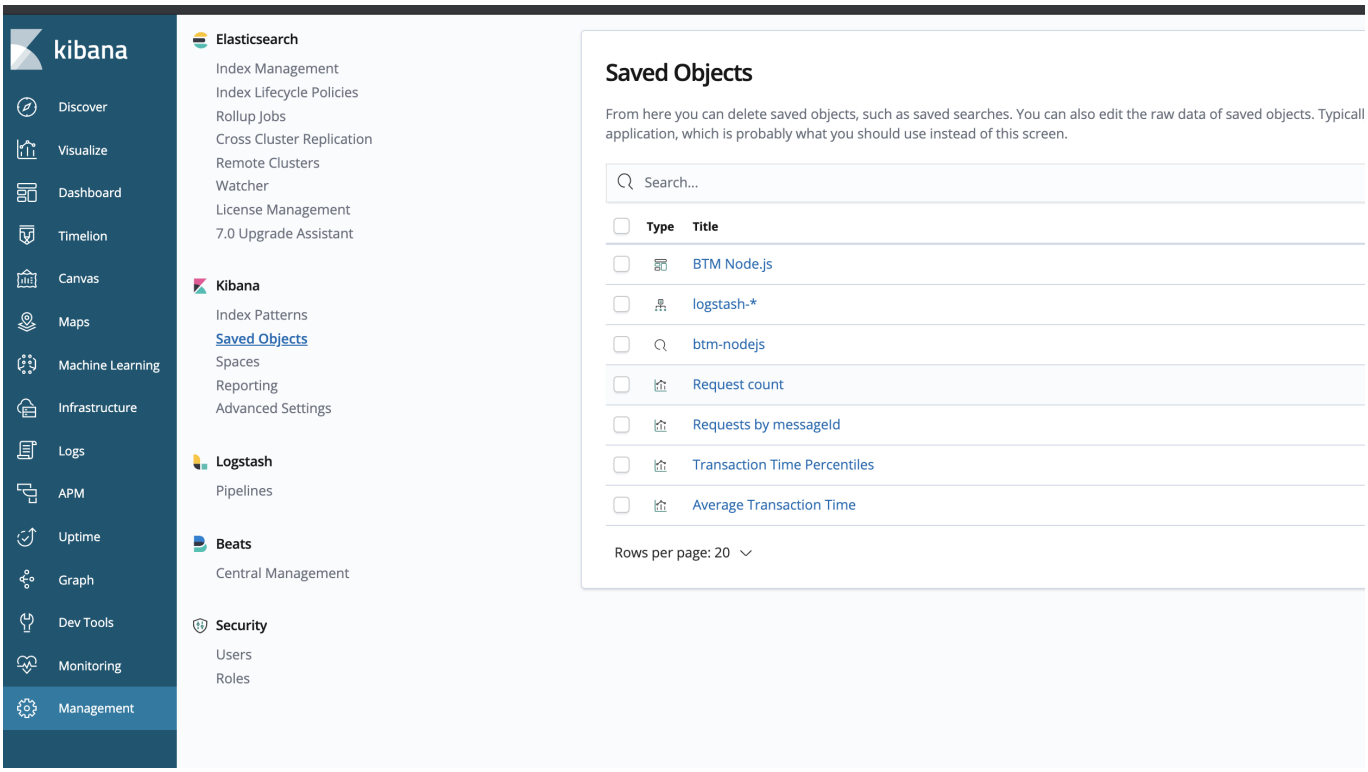
Logon using user: `elastic`, password: `changeme`

> If you run the lab locally, you can use the `localhost`

4). Import the index pattern, saved search, visualizations and dashboard from the provided `kibana.json` file. We will use them in the next part of the lab:

- Go to Management -> Kibana/Saved Objects -> Import and select the `b2m-nodejs-v2/lab-1/kibana.json` file.

You should see the following Saved Objects imported:



Instrument the Node.js app with logging

Now we will configure a logging library for our Node.js app and add some log statements that will be easy to consume by the ELK stack.

1). Go to the directory `b2m-nodejs-v2/lab-1/app` where the `server.js` file is located and add the following dependency to the `package.json`:

```
    "winston": "^3.2.1"
```

This will add winston logging library for node.js.

2). Add/uncomment the following line at the beginning of `server.js` to load the `winston` module:

```
const { createLogger, format, transports } = require('winston')
```

then create/uncomment the `logger` object:

```
const logger = createLogger({
  level: 'debug',
  format: format.combine(
    format.timestamp({
      format: "YYYY-MM-DD'T'HH:mm:ss.SSSZ"
    }),
    format.json()
  ),
```

```
  transports: [new transports.Console()]
});
```

The configuration above specifies timestamp field format and enables sending logs in `json` format to STDOUT.
Timestamp should include the time zone information and be precise down to milliseconds.

Whenever you want to generate a log entry, just use the `logger` object with level specified methods: **error**, **warn**, **info**, **verbose**, **debug**

```
msg = 'RSAP0010E: Severe problem detected'
logger.error(msg)
msg = 'RSAP0001I: Transaction OK'
logger.info(msg)
```

You can add also additional metadata like `errorCode` or `transactionTime` that can be useful in log analytics.

3). Add some logging statements as described below. Additional metadata will be used later in our log analytics dashboard.

Look for commented lines starting with `logger` and uncomment them.

```
msg = 'RSAP0001I: Transaction OK'
logger.info(msg, {"errCode": "RSAP0001I", "transactionTime": delay})

msg = 'RSAP0010E: Severe problem detected'
logger.error(msg, {"errorCode": "RSAP0010E", "transactionTime": delay})
```

After these changes the expected application STDOUT is:

```
{"errCode":"RSAP0001I","transactionTime":81,"level":"info","message":"RSAP
0001I: Transaction OK","timestamp":"2019-02-27T07:34:49.625Z"}
{"errCode":"RSAP0010E","transactionTime":76,"level":"error","message":"RSA
P0010E: Severe problem detected","timestamp":"2019-02-27T07:34:50.008Z"}
{"errCode":"RSAP0001I","transactionTime":22,"level":"info","message":"RSAP
0001I: Transaction OK","timestamp":"2019-02-27T07:34:50.325Z"}
{"errCode":"RSAP0001I","transactionTime":1,"level":"info","message":"RSAP0
001I: Transaction OK","timestamp":"2019-02-27T07:34:50.620Z"}
{"errCode":"RSAP0001I","transactionTime":96,"level":"info","message":"RSAP
0001I: Transaction OK","timestamp":"2019-02-27T07:34:50.871Z"}
{"errCode":"RSAP0001I","transactionTime":62,"level":"info","message":"RSAP
0001I: Transaction OK","timestamp":"2019-02-27T07:34:51.156Z"}
```

3). Rebuild the Node.js app container:

```
cd b2m-nodejs-v2/lab-1
docker-compose down
docker-compose build
docker-compose up -d
```

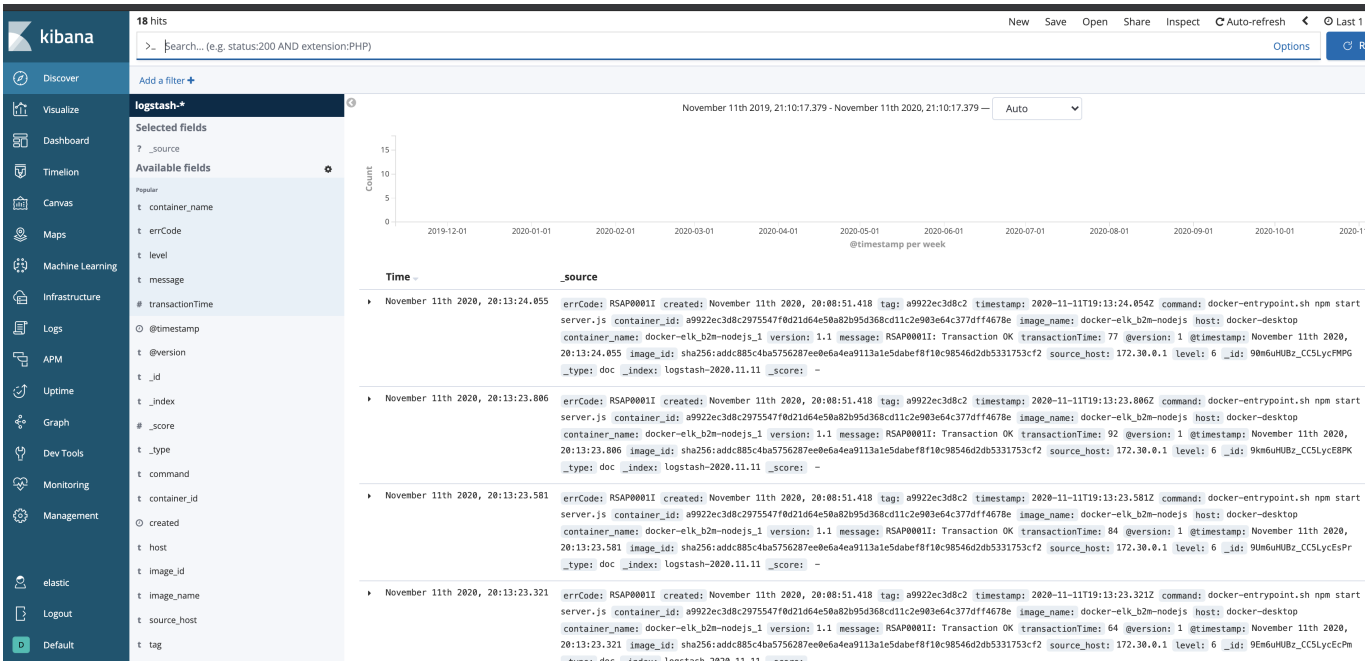
4). Simulate a couple of transactions using your web browser or `curl` by accessing `http://<your-hostname>:3001/checkout`:

```
for i in {1..10000}; do curl -w "\n" http://<your-hostname>:3001/checkout;
done
```

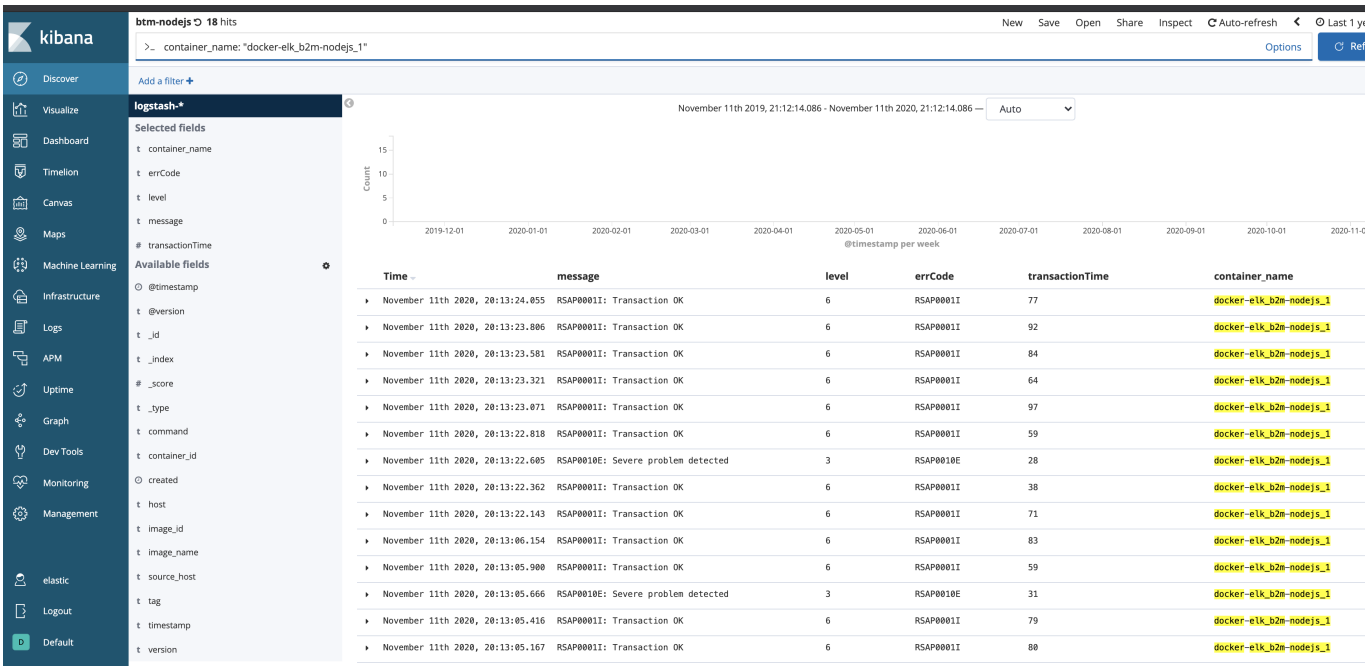
> If you run the lab locally, you can use the `localhost`

and check out the Kibana (`http://<your-hostname>:5601` `elastic/changeme`)

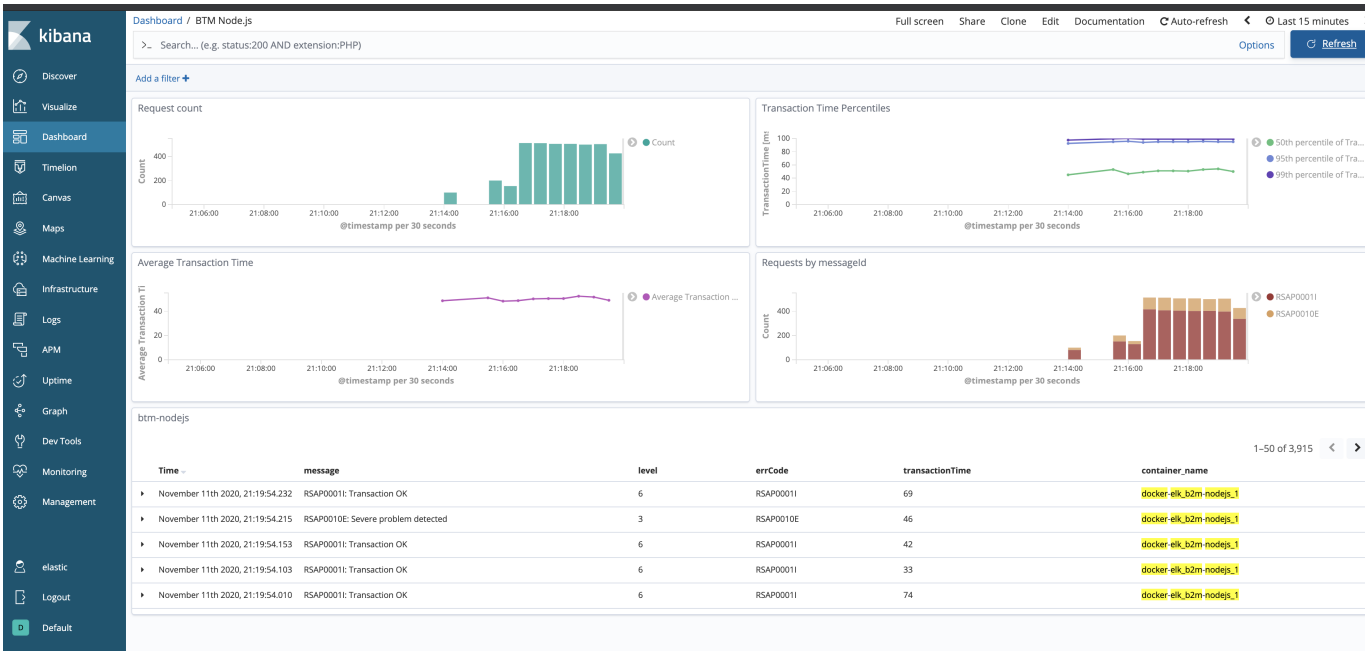
- The **Discover** view should be similar to:



- From the upper menu select `Open` and select the preconfigured saved search `btm-nodejs`:



- Access Dashboards -> `BTM Node.js`:



Kibana Dashboard is a collection of **Visualizations**. If you are interested in how these charts are configured, select the **Visualize** on the left menu, then select one of the Visualizations and check its configuration.

> **Stop the docker-compose stack before starting the next exercise:**

```
cd b2m-nodejs-v2/lab-1
docker-compose down -v
```

Lab 2 - Node.js app logging with Humio

Humio is purpose-built to help any organization achieve the benefits of large-scale logging and analysis. It has virtually no latency even at massive ingest volumes. Humio aggregates, alerts, and visualizes streaming

data in real time, so no matter what volume of data you send to Humio, data is processed instantly. This gives organizations live observability into the operations and health of their systems.
Indexing can be a very computationally-expensive activity, causing latency between data entering a system and then being included in search results and visualizations. Humio does no indexing, so it remains lightning fast with no compromise on performance. Free-text search lets you search anything, in any field, without relying on pre-parsed fields. Schema on read allows you to extract data, define new fields, and use them to filter and aggregate as you search.
Humio uses high data compression, so you can cut hardware costs and store more data. High compression also makes it cost-effective to retain more data for longer, enabling both more detailed analysis and traceability over longer time periods.

More on https://www.humio.com/log-management#features

During this lab we will run Humio in docker-compose and configure it in order to collect and analyse the log stream from a sample Node.js microservice.

Instrument the Node.js app with logging

If you did the app instrumentation part of the Lab 1, just copy and replace the `b2m-nodejs-v2/lab-1/app/server.js` to `b2m-nodejs-v2/lab-2/app/server.js` and change the line:

```
const port = process.env.PORT || 3001
```

to

```
const port = process.env.PORT || 3002
```

If you start with Lab 2, do the instrumentation steps with `winston` logging library as described in chapter Instrument the Node.js app with logging.

Deploy Humio with Docker Compose

1). Start Humio with our Node.js app in docker-compose:

```
cd b2m-nodejs-v2/lab-2
./start-lab2.sh
```

> Note, it may take a while for the first time, because it will download the Humio image from DockerHub.
> Ignore the error: `"No such service: b2m-nodejs"`, we will add it later.

While waiting for containers, review the configuration of our logging lab.

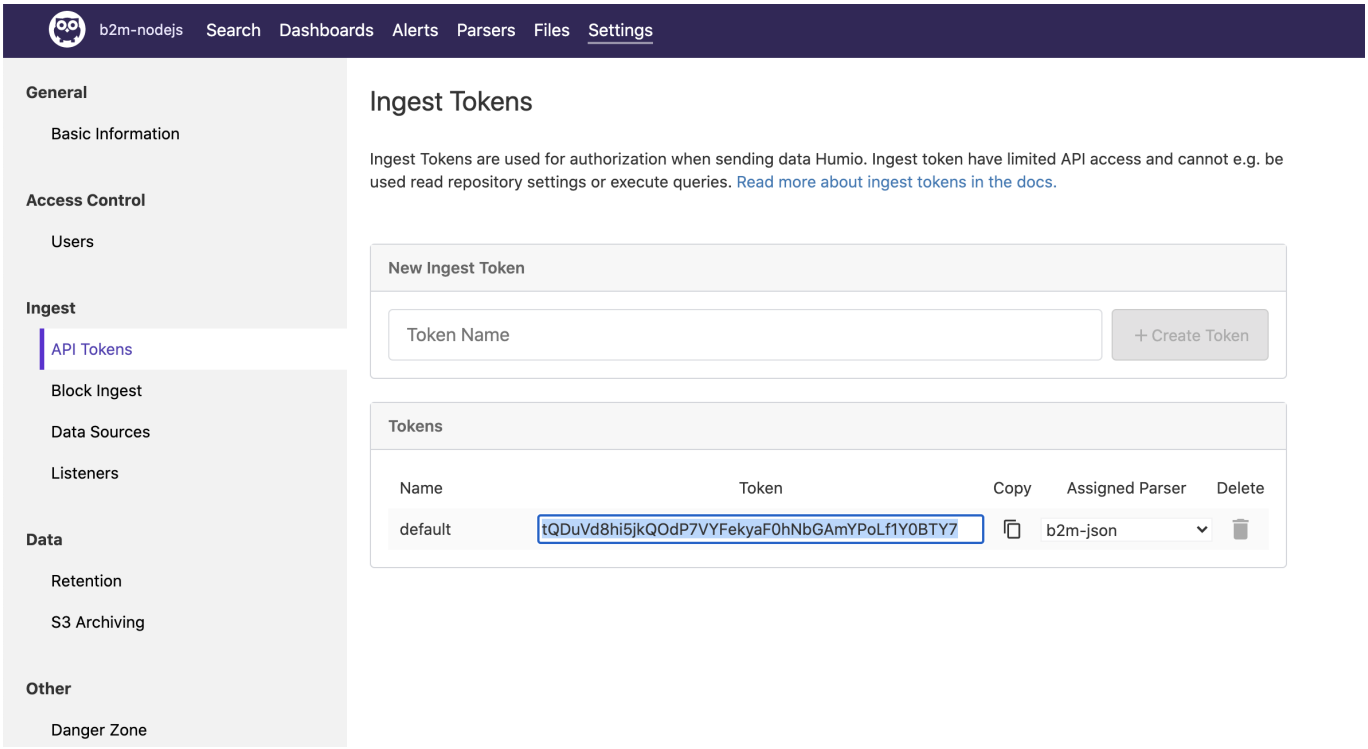
- `b2m-nodejs-v2/lab-2/docker-compose.yaml` - this is the main config file for docker-compose stack which specifies all options for all containers in the stack.

- `b2m-nodejs-v2/lab-2/app/server.js` - the source code of our sample Node.js application instrumented using `winston` logging library.
- `b2m-nodejs-v2/lab-2/app/Dockerfile` - this file is used to build your application docker image.

2). Access the Humio UI using web browser on `http://<your-hostname>:8080` (If you run the lab locally, you can use the `localhost`)

3). Click `Add item` and create `b2m-nodejs` repository.

4). Inside new repository, go to `Settings -> API Tokens`. Copy the default token.



5). Stop the Humio stack:

```
cd b2m-nodejs-v2/lab-2
docker-compose down
```

6). Edit the `b2m-nodejs-v2/lab-2/docker-compose.yaml` and paste the token as value of `splunk-token` (remember to put the token in quotes).

7). Uncomment the whole `b2m-nodejs` section (together with all its options). Make sure the YAML indentation is correct.

8). Start the Humio and Node.js stack:

```
cd b2m-nodejs-v2/lab-2
./start-lab2.sh
```

9). Access the Humio UI using internet browser on `http://<your-hostname>:8080` (If you run the lab locally, you can use the `localhost`).

10). Go to `b2m-nodejs` repository and select `Parsers`. Click on `+ New Parser`. Name it `b2m-json`.

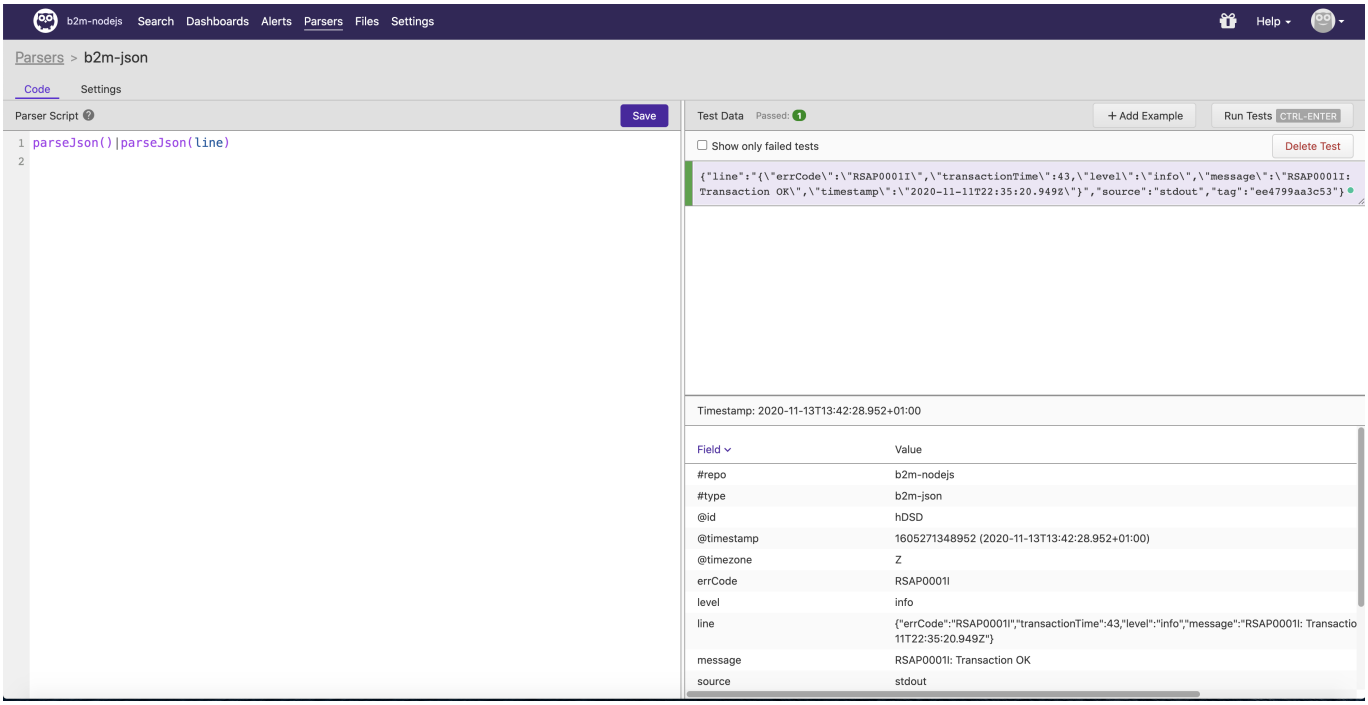
11). Replace the default content on the left side with the following parser script:

```
parseJson()|parseJson(line)
```

On the right side of the `Parsers` editor you can test your parser. Delete the default tests, click `+ Add Example` and paste this line (which looks similar to the log line emitted by our app docker container):

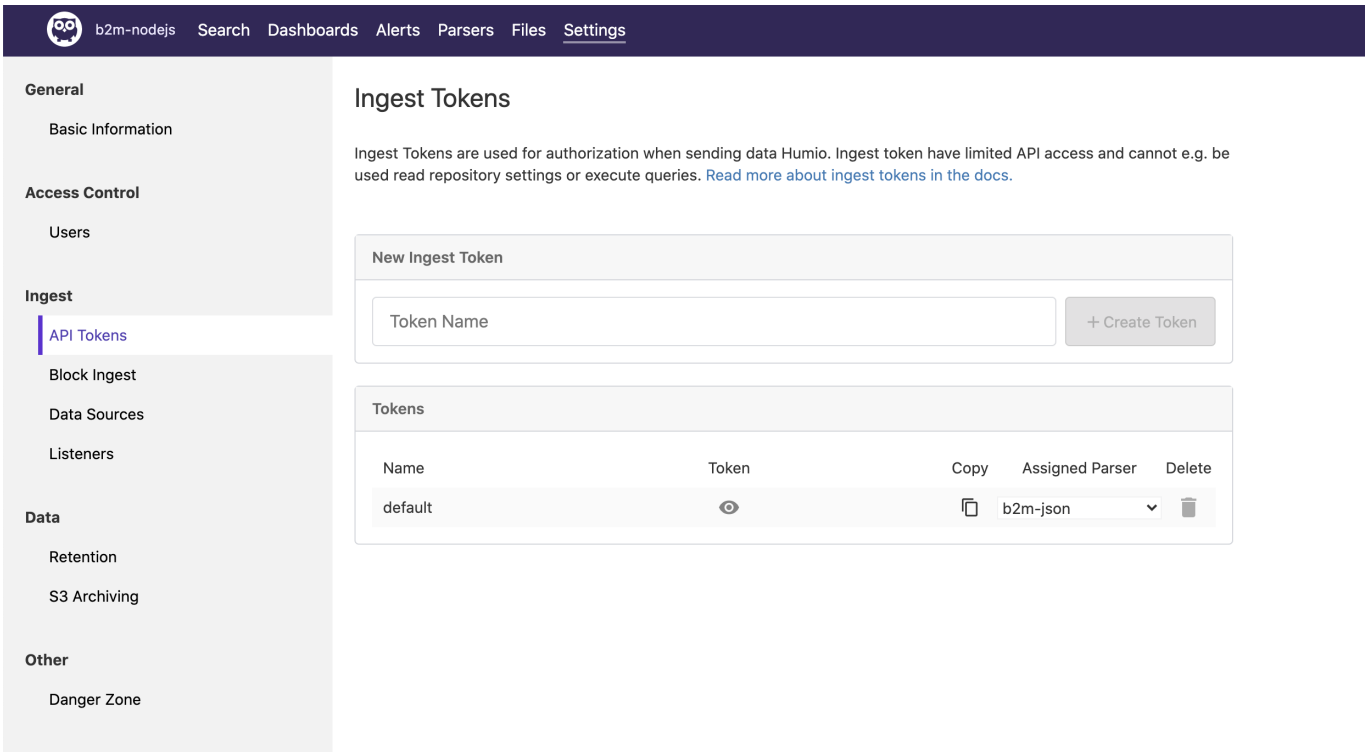
```
{"line":"
{\"errCode\":\"RSAP0001I\",\"transactionTime\":43,\"level\":\"info\",\"mes
sage\":\"RSAP0001I: Transaction OK\",\"timestamp\":\"2020-11-
11T22:35:20.949Z\"}","source":"stdout","tag":"ee4799aa3c53"}
```

and verify extracted fields. Save your new Parser.



> Why we parse JSON twice?
> Docker wraps the application log (which the app emits as JSON) in its own JSON envelope, so first we parse docker JSON and then parse JSON contents of already extracted field `line`. Note the `|` character (which works exactly the same way as in Linux or Unix shell!).

12). Go to `Settings->API` tokens and select your newly defined parser in the `Assigned Parser` option of the `default` API token.



13). Simulate a couple of transactions using your web browser or `curl` by accessing `http://<your-hostname>:3002/checkout`:

```
for i in {1..10000}; do curl -w "\n" http://localhost:3002/checkout; done
```

> If you run the lab locally, you can use the `localhost`

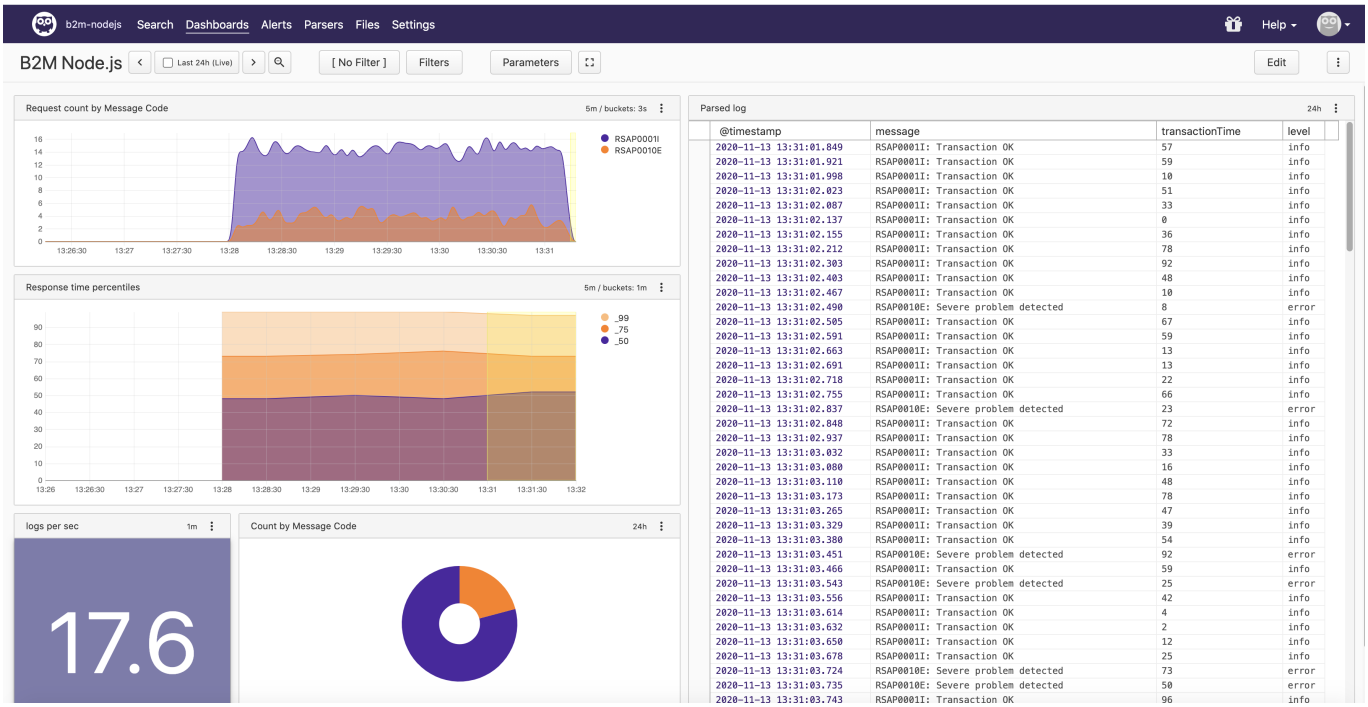
Verify results in the `Search` view.

14). Import the Humio dashboard provided with this lab `humio-dashboard.yaml`:

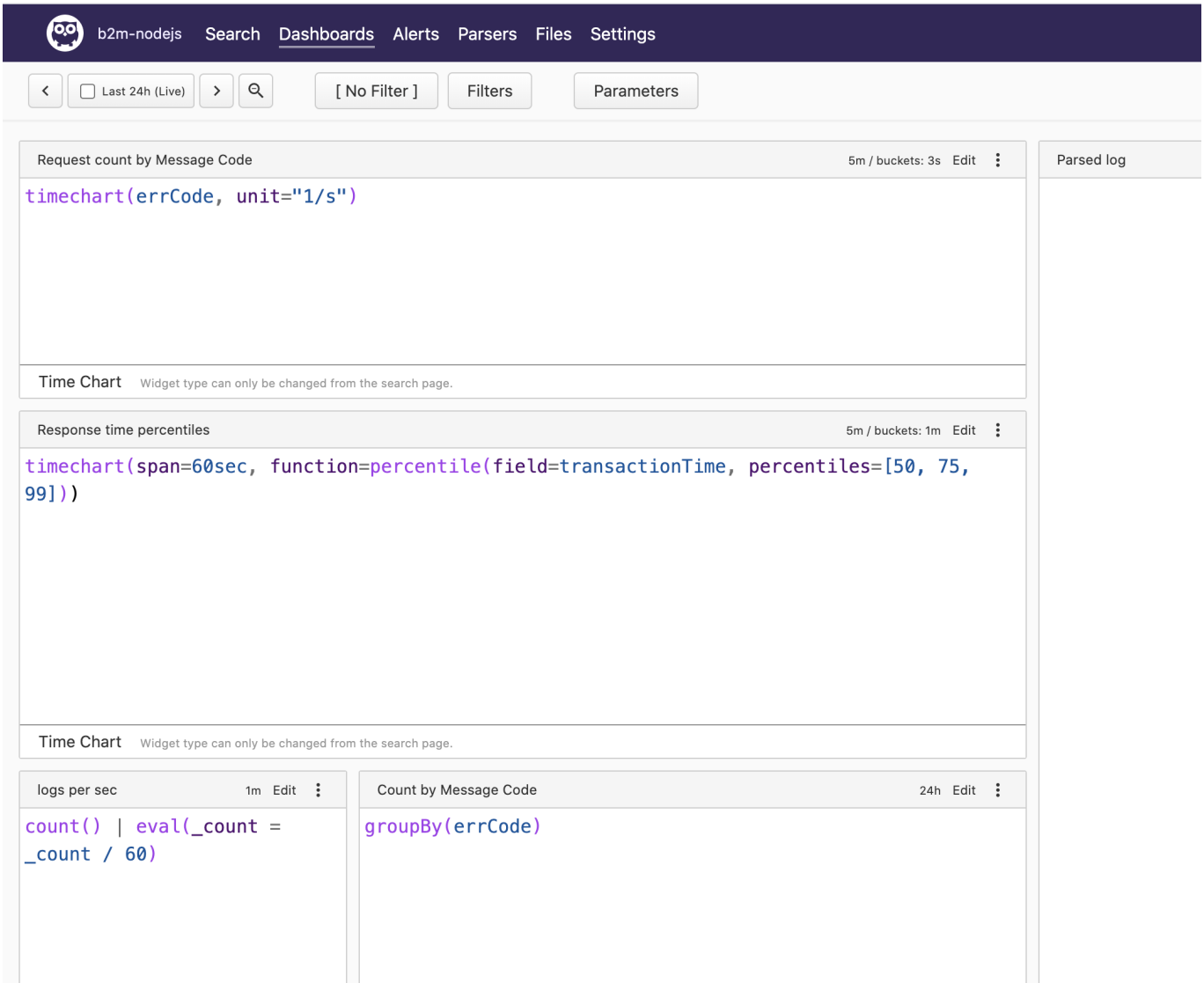
- Go to `Dashboards`, click `+ New Dashboard`
- Name your dashboard `B2M Node.js` and select `Template file` option.
- Click `Upload Template` and select `b2m-nodejs-v2/lab-2/humio-dashboard.yaml`
- Click `Create Dashboard`

15). Access `B2M Node.js` dashboard. Generate more application requests with:

```
for i in {1..10000}; do curl -w "\n" http://localhost:3002/checkout; done
```



16). Click the `Edit` button and then `Show Queries`. You will see the Humio queries used to produce data for every chart.



Stop the docker-compose stack before starting the next exercise:

```
cd b2m-nodejs-v2/lab-2
docker-compose down -v
```

Monitoring

One of the most important decisions to make when setting up web application monitoring is deciding on the type of metrics you need to collect about your app. The metrics you choose simplifies troubleshooting when a problem occurs and also enables you to stay on top of the stability of your services and infrastructure.

The RED method follows on the principles outlined in the Four Golden Signals, which focuses on measuring things that end-users care about when using your web services. With the RED method, three key metrics are instrumented that monitor every microservice in your architecture:

- (Request) Rate - the number of requests, per second, your services are serving.
- (Request) Errors - the number of failed requests per second.
- (Request) Duration - The amount of time each request takes expressed as a time interval.

Rate, Errors and Duration attempt to cover the most obvious web service issues. These metrics also capture an error rate that is expressed as a proportion of request rate.

Of course, this is just a good starting point for metrics instrumentation. Generally, the more metrics we collect from an application the better.

> **Instrument first, ask questions later**
> During development you will never know what questions you need to ask later. Software needs good instrumentation, it's not optional. Metrics are cheap. Use them generously. The First and the most important rule, if you have to remember only one thing remember this one. Instrument all the things!
>
> The Zen of Prometheus

Deploy the monitoring stack in Docker Compose

1). Start the monitoring stack:

```
cd b2m-nodejs-v2/lab-3
docker-compose build
docker-compose up -d
```

> Note, it may take a while for the first time, because it will download the monitoring stack images from DockerHub and build an image for our Node.js application.

2). While waiting for containers, review the configuration of our monitoring lab.

- `b2m-nodejs-v2/lab-3/docker-compose.yaml` - this is the main config file for docker-compose stack which specifies all options for all containers in the stack.
- `b2m-nodejs-v2/lab-3/app/server.js` - the source code of our sample Node.js application.
- `b2m-nodejs-v2/lab-3/app/Dockerfile` - this file is used to build your app docker image.

3). Verify that you can access the monitoring stack UIs:

- Prometheus: `http://<your-hostname>:9090`
- Grafana: `http://<your-hostname>:3000` (user/pw: admin/foobar)

4). Verify that your Node.js app works:

```
curl http://<your-hostname>:3003
```

Instrument application code with Node.js client library for Prometheus

Expose default Node.js runtime metrics

1). Go to the directory `b2m-nodejs-v2/lab-3/app` where the `server.js` file is located and add the following dependency to the `package.json`:

```
    "prom-client": "^11.2.1"
```

There are some default metrics recommended by Prometheus itself.

To collect these, call `collectDefaultMetrics`

> Some of the metrics, concerning File Descriptors and Memory, are only available on Linux.

In addition, some Node-specific metrics are included, such as event loop lag, active handles and Node.js version. See what metrics there are in https://github.com/siimon/prom-client/lib/metrics.

`collectDefaultMetrics` takes 1 options object with 3 entries, a timeout for how often the probe should be fired, an optional prefix for metric names and a registry to which metrics should be registered. By default probes are launched every 10 seconds, but this can be modified like this:

```
const client = require('prom-client');
const collectDefaultMetrics = client.collectDefaultMetrics;
// Probe every 5th second.
collectDefaultMetrics({ timeout: 5000 });
```

2). Edit `server.js` and *uncomment* the following lines to enable exposure of default set of Node.js metrics on standard Prometheus route `/metrics`

```
const Prometheus = require('prom-client')
const metricsInterval = Prometheus.collectDefaultMetrics()
```

and

```
app.get('/metrics', (req, res) => {
  res.set('Content-Type', Prometheus.register.contentType)
  res.end(Prometheus.register.metrics())
})
```

3). Rebuild you app container

```
cd ~/b2m-nodejs-v2/lab-3
docker-compose down
docker-compose build
docker-compose up -d
```

Run a couple of transactions by refreshing the URL: `http://<your-hostname>:3003/checkout`

Use browser or `curl` to access `http://<your-hostname>:3003/metrics` in order to verify exposed metrics. Output should be similar to:

```
# HELP process_cpu_user_seconds_total Total user CPU time spent in
seconds.
# TYPE process_cpu_user_seconds_total counter
process_cpu_user_seconds_total 0.028084 1546452963611

# HELP process_cpu_system_seconds_total Total system CPU time spent in
seconds.
# TYPE process_cpu_system_seconds_total counter
process_cpu_system_seconds_total 0.0038780000000000004 1546452963611

# HELP process_cpu_seconds_total Total user and system CPU time spent in
seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 0.031962 1546452963611

# HELP process_start_time_seconds Start time of the process since unix
epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1546452953

# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 29188096 1546452963611

# HELP nodejs_eventloop_lag_seconds Lag of event loop in seconds.
# TYPE nodejs_eventloop_lag_seconds gauge
nodejs_eventloop_lag_seconds 0.000393303 1546452963612

# HELP nodejs_active_handles_total Number of active handles.
# TYPE nodejs_active_handles_total gauge
nodejs_active_handles_total 3 1546452963611
```

```
# HELP nodejs_active_requests_total Number of active requests.
# TYPE nodejs_active_requests_total gauge
nodejs_active_requests_total 0 1546452963611

# HELP nodejs_heap_size_total_bytes Process heap size from node.js in
bytes.
# TYPE nodejs_heap_size_total_bytes gauge
nodejs_heap_size_total_bytes 20217856 1546452963611

# HELP nodejs_heap_size_used_bytes Process heap size used from node.js in
bytes.
# TYPE nodejs_heap_size_used_bytes gauge
nodejs_heap_size_used_bytes 8464704 1546452963611

# HELP nodejs_external_memory_bytes Nodejs external memory size in bytes.
# TYPE nodejs_external_memory_bytes gauge
nodejs_external_memory_bytes 24656 1546452963611

# HELP nodejs_heap_space_size_total_bytes Process heap space size total
from node.js in bytes.
# TYPE nodejs_heap_space_size_total_bytes gauge
nodejs_heap_space_size_total_bytes{space="read_only"} 0 1546452963612
nodejs_heap_space_size_total_bytes{space="new"} 8388608 1546452963612
nodejs_heap_space_size_total_bytes{space="old"} 8134656 1546452963612
nodejs_heap_space_size_total_bytes{space="code"} 1048576 1546452963612
nodejs_heap_space_size_total_bytes{space="map"} 1073152 1546452963612
nodejs_heap_space_size_total_bytes{space="large_object"} 1572864
1546452963612

# HELP nodejs_heap_space_size_used_bytes Process heap space size used from
node.js in bytes.
# TYPE nodejs_heap_space_size_used_bytes gauge
nodejs_heap_space_size_used_bytes{space="read_only"} 0 1546452963612
nodejs_heap_space_size_used_bytes{space="new"} 829768 1546452963612
nodejs_heap_space_size_used_bytes{space="old"} 6008448 1546452963612
nodejs_heap_space_size_used_bytes{space="code"} 847136 1546452963612
nodejs_heap_space_size_used_bytes{space="map"} 533016 1546452963612
nodejs_heap_space_size_used_bytes{space="large_object"} 249024
1546452963612

# HELP nodejs_heap_space_size_available_bytes Process heap space size
available from node.js in bytes.
# TYPE nodejs_heap_space_size_available_bytes gauge
nodejs_heap_space_size_available_bytes{space="read_only"} 0 1546452963612
nodejs_heap_space_size_available_bytes{space="new"} 3294904 1546452963612
nodejs_heap_space_size_available_bytes{space="old"} 1656536 1546452963612
nodejs_heap_space_size_available_bytes{space="code"} 0 1546452963612
nodejs_heap_space_size_available_bytes{space="map"} 80 1546452963612
nodejs_heap_space_size_available_bytes{space="large_object"} 1506500096
1546452963612

# HELP nodejs_version_info Node.js version info.
```

```
# TYPE nodejs_version_info gauge
nodejs_version_info{version="v10.7.0",major="10",minor="7",patch="0"} 1
```

Define custom metric

Node.js Prometheus client library allows to define various types of Prometheus metrics like histograms, summaries, gauges and counters. More detailed description of metric types can be found in Prometheus documentation.

In this lab we will define two custom metrics:

- counter `checkouts_total` which will store a total number of `checkout` requests
- histogram `http_request_duration_ms` which will store percentiles of application requests response time

Uncomment the rest of commented lines in `server.js`.

checkouts_total

Declaration of `checkouts_total` counter.

```
const checkoutsTotal = new Prometheus.Counter({
   name: 'checkouts_total',
   help: 'Total number of checkouts',
   labelNames: ['payment_method']
 })
```

This counter will be incremented for every `checkout` request

```
checkoutsTotal.inc({
     payment_method: paymentMethod
})
```

http_request_duration_ms

Declaration of `http_request_duration_ms` histogram:

```
const httpRequestDurationMicroseconds = new Prometheus.Histogram({
   name: 'http_request_duration_ms',
   help: 'Duration of HTTP requests in ms',
   labelNames: ['method', 'route', 'code'],
   buckets: [0.10, 5, 15, 50, 100, 200, 300, 400, 500]  // buckets for
response time from 0.1ms to 500ms
 })
```

The current time is recorded before each request:

```
app.use((req, res, next) => {
  res.locals.startEpoch = Date.now()
  next()
})
```

We record the current time also after each request and update our `http_request_duration_ms` histogram accordingly:

```
app.use((req, res, next) => {
  const responseTimeInMs = Date.now() - res.locals.startEpoch

  httpRequestDurationMicroseconds
    .labels(req.method, req.route.path, res.statusCode)
    .observe(responseTimeInMs)
  next()
})
```

After you complete code changes, rebuild your app container:

```
cd ~/b2m-nodejs-v2/lab-3
docker-compose down
docker-compose build
docker-compose up -d
```

Run a couple of transactions by refreshing the URL: `http://<your-hostname>:3003/checkout`

Use browser to access `http://<your-hostname>:3003/metrics` to verify exposed metrics. The output should be similar to:

```
(...)
# HELP checkouts_total Total number of checkouts
# TYPE checkouts_total counter
checkouts_total{payment_method="paypal"} 7
checkouts_total{payment_method="stripe"} 5

# HELP http_request_duration_ms Duration of HTTP requests in ms
# TYPE http_request_duration_ms histogram
http_request_duration_ms_bucket{le="0.1",code="304",route="/",method="GET"
} 0
http_request_duration_ms_bucket{le="5",code="304",route="/",method="GET"}
0
http_request_duration_ms_bucket{le="15",code="304",route="/",method="GET"}
0
http_request_duration_ms_bucket{le="50",code="304",route="/",method="GET"}
```

```
0
http_request_duration_ms_bucket{le="100",code="304",route="/",method="GET"
} 0
http_request_duration_ms_bucket{le="200",code="304",route="/",method="GET"
} 3
http_request_duration_ms_bucket{le="300",code="304",route="/",method="GET"
} 3
http_request_duration_ms_bucket{le="400",code="304",route="/",method="GET"
} 3
http_request_duration_ms_bucket{le="500",code="304",route="/",method="GET"
} 3
http_request_duration_ms_bucket{le="+Inf",code="304",route="/",method="GET
"} 3
http_request_duration_ms_sum{method="GET",route="/",code="304"} 415
http_request_duration_ms_count{method="GET",route="/",code="304"} 3
http_request_duration_ms_bucket{le="0.1",code="500",route="/bad",method="G
ET"} 0
http_request_duration_ms_bucket{le="5",code="500",route="/bad",method="GET
"} 1
http_request_duration_ms_bucket{le="15",code="500",route="/bad",method="GE
T"} 1
http_request_duration_ms_bucket{le="50",code="500",route="/bad",method="GE
T"} 1
http_request_duration_ms_bucket{le="100",code="500",route="/bad",method="G
ET"} 1
http_request_duration_ms_bucket{le="200",code="500",route="/bad",method="G
ET"} 1
http_request_duration_ms_bucket{le="300",code="500",route="/bad",method="G
ET"} 1
http_request_duration_ms_bucket{le="400",code="500",route="/bad",method="G
ET"} 1
http_request_duration_ms_bucket{le="500",code="500",route="/bad",method="G
ET"} 1
http_request_duration_ms_bucket{le="+Inf",code="500",route="/bad",method="
GET"} 1
http_request_duration_ms_sum{method="GET",route="/bad",code="500"} 1
http_request_duration_ms_count{method="GET",route="/bad",code="500"} 1
http_request_duration_ms_bucket{le="0.1",code="304",route="/checkout",meth
od="GET"} 8
http_request_duration_ms_bucket{le="5",code="304",route="/checkout",method
="GET"} 12
http_request_duration_ms_bucket{le="15",code="304",route="/checkout",metho
d="GET"} 12
http_request_duration_ms_bucket{le="50",code="304",route="/checkout",metho
d="GET"} 12
http_request_duration_ms_bucket{le="100",code="304",route="/checkout",meth
od="GET"} 12
http_request_duration_ms_bucket{le="200",code="304",route="/checkout",meth
od="GET"} 12
http_request_duration_ms_bucket{le="300",code="304",route="/checkout",meth
od="GET"} 12
http_request_duration_ms_bucket{le="400",code="304",route="/checkout",meth
od="GET"} 12
http_request_duration_ms_bucket{le="500",code="304",route="/checkout",meth
```

```
od="GET"} 12
http_request_duration_ms_bucket{le="+Inf",code="304",route="/checkout",met
hod="GET"} 12
http_request_duration_ms_sum{method="GET",route="/checkout",code="304"} 4
http_request_duration_ms_count{method="GET",route="/checkout",code="304"}
12
```

Besides the default set of metrics related to resource utilization by the application process, we can see the additional metrics:

- `checkouts_total`
- `http_request_duration_ms_bucket`

Metrics collection

Prometheus in this lab has been pre-configured to collect metrics from your Node.js app. Check the `b2m-nodejs/lab-3/prometheus/prometheus.yml` file for this config:

```
- job_name: 'b2m-nodejs'
  scrape_interval: 20s
  static_configs:
  - targets: ['b2m-nodejs:3003']
    labels:
      service: 'b2m-nodejs'
```

Verify that Prometheus server was started via: http://:9090
Check the status of scraping targets in Prometheus UI -> Status -> Targets

Run example PromQL queries

Generate some application load before running the queries:

```
for i in {1..10000}; do curl -w "\n" http://<your-hostname>:3003/checkout;
done
```

> If you run the lab locally, you can use the `localhost`

Run the following example PromQL queries using the Prometheus UI.

Throughput

Error rate

Range[0,1]: number of 5xx requests / total number of requests

```

```

```
sum(increase(http_request_duration_ms_count{code=~"^5..$"}[1m])) /
sum(increase(http_request_duration_ms_count[1m]))
```

Expected value `~0.2` because our application should return 500 for about 20% of transactions.

Request Per Minute

```
sum(rate(http_request_duration_ms_count[1m])) by (service, route, method,
code)  * 60
```

Check the graph.

Response Time

Apdex

Apdex score approximation: `100ms` target and `300ms` tolerated response time

```
(sum(rate(http_request_duration_ms_bucket{le="100"}[1m])) by (service) +
sum(rate(http_request_duration_ms_bucket{le="300"}[1m])) by (service)
) / 2 / sum(rate(http_request_duration_ms_count[1m])) by (service)
```

> Note that we divide the sum of both buckets. The reason is that the histogram buckets are cumulative. The le="100" bucket is also contained in the le="300" bucket; dividing it by 2 corrects for that. - Prometheus docs

95th Response Time

```
histogram_quantile(0.95, sum(rate(http_request_duration_ms_bucket[1m])) by
(le, service, route, method))
```

Median Response Time

```
histogram_quantile(0.5, sum(rate(http_request_duration_ms_bucket[1m])) by
(le, service, route, method))
```

Average Response Time

```
avg(rate(http_request_duration_ms_sum[1m]) /
rate(http_request_duration_ms_count[1m])) by (service, route, method,
```

```
code)
```

Memory Usage

Average Memory Usage

In Megabytes.

```
avg(nodejs_external_memory_bytes / 1024 ) by (service)
```

Configure Prometheus alert

Alerting rules allows to define alert conditions based on Prometheus expression language expressions and to send notifications about firing alerts to an external service. In this lab we will configure one alerting rule for median response time higher than 100ms.

Lab instruction:

Add the following alert rule to the `alert.rules` file. In the lab VM it is located in `/root/prometheus/prometheus/alert.rules`

```
  - alert: APIHighMedianResponseTime
    expr: histogram_quantile(0.5, sum by(le, service, route, method)
(rate(http_request_duration_ms_bucket[1m])))
      > 30
    for: 1m
    annotations:
      description: '{{ $labels.service }}, {{ $labels.method }} {{
$labels.route }}
        has a median response time above 100ms (current value: {{ $value
}}ms)'
      summary: High median response time on {{ $labels.service }} and {{
```

```
  $labels.method
          }} {{ $labels.route }}
```

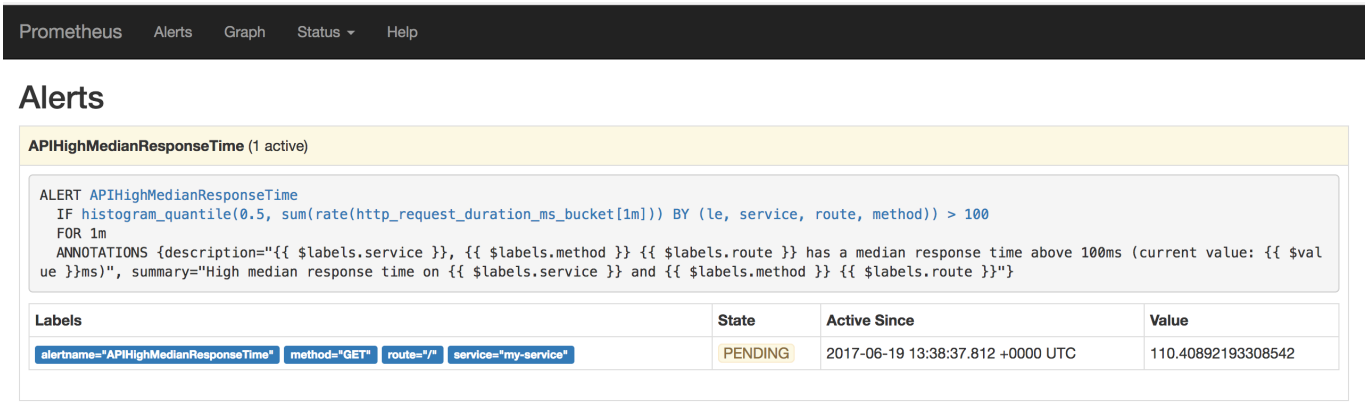
Restart the Prometheus stack:

```
cd ~/prometheus
docker-compose down
docker-compose up -d
```

Alerts can be listed via Prometheus UI: `http://<your-hostname>:9090/alerts`

States of active alerts:

- `pending`:



- `firing`:

Set the Prometheus datasource in Grafana

Logon to Grafana via `http://<your-hostname>:3000`

- user: admin
- password: foobar

Verify the prometheus datasource configuration in Grafana. If it was not already configured, create a Grafana datasource with these settings:

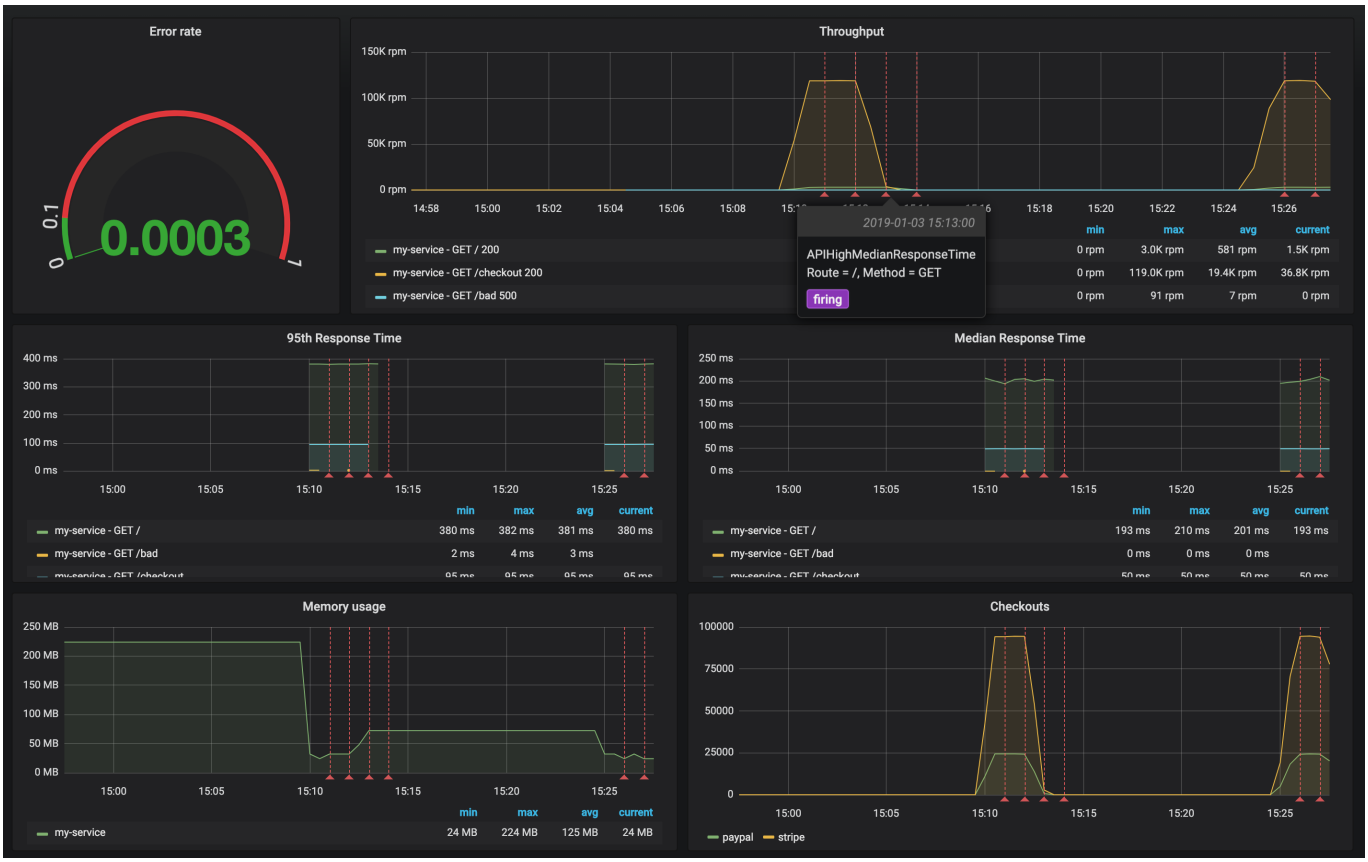
- name: Prometheus
- type: prometheus
- url: http://prometheus:9090
- Access: Server

Configure dashboard

Grafana Dashboard to import: `~/b2m-nodejs-v2/lab-3/btm-nodejs-grafana.json`

Monitoring dashboard was created according to the RED Method principles:

- Rate (`Thoughput` and `Checkouts` panels)
- Errors (`Error rate` panel)
- Duration (`95th Response Time` and `Median Response Time` panels)



Review the configuration of each dashboard panel. Check the annotation settings.

Define the Apdex score chart using the following query:

```
(sum(rate(http_request_duration_ms_bucket{le="100"}[1m])) by (service) +
sum(rate(http_request_duration_ms_bucket{le="300"}[1m])) by (service)
) / 2 / sum(rate(http_request_duration_ms_count[1m])) by (service)
```

You can add it to the existing dashboard:

- Click on the icon `Add panel` and select `Graph` panel type.
- Click on the panel title and select edit.
- Select `Prometheus` datasource in the `Metrics` tab of the panel query editor
- Copy PromQL to the free form field
- Verify the results on the panel preview
- Explore other Graph panel options

How to monitor applications on OpenShift 4.x with Prometheus Operator

OpenShift Container Platform includes a pre-configured, pre-installed, and self-updating monitoring stack that is based on the Prometheus open source project and its wider eco-system. It provides monitoring of cluster components and includes a set of alerts to immediately notify the cluster administrator about any occurring problems and a set of Grafana dashboards. The cluster monitoring stack is only supported for monitoring OpenShift Container Platform clusters and adding additional monitoring targets is not supported.

In this lab we will configure application monitoring stack on Openshift 4.x using Prometheus Operator for a sample Node.js microservice instrumented with Prometheus client library (instrumentation was covered in Lab-3).

Deploy an instrumented application

Use the following command and provided yaml file, to deploy sample Node.js microservice instrumented with Prometheus client library.

```
oc new-project b2m-nodejs
oc create -f b2m-nodejs.yml
```

In case of problems with pulling the app image from Docker Hub, you can build the application image by yourself using:

```
oc new-app https://github.com/rafal-szypulka/b2m-nodejs-v2 \
--context-dir=lab-4/app --name b2m-nodejs \
--labels='name=b2m-nodejs' --insecure-registry=true
```

The monitor expects the service's port name to be `web`, so edit the service and change `spec.ports.name` to `web`:

```
  spec:
    clusterIP: xxx.xxx.xxx.xxx
    ports:
      - name: web
```

Create route to expose this application externally:

```
oc expose svc b2m-nodejs
```

Collect the app URL:

```
$ oc get routes -n default
NAME         HOST/PORT                                          PATH
SERVICES     PORT    TERMINATION   WILDCARD
b2m-nodejs   b2m-nodejs-b2m-nodejs.apps.rsocp.os.fyre.ibm.com
b2m-nodejs   <all>   edge          None
```

and make sure it works:

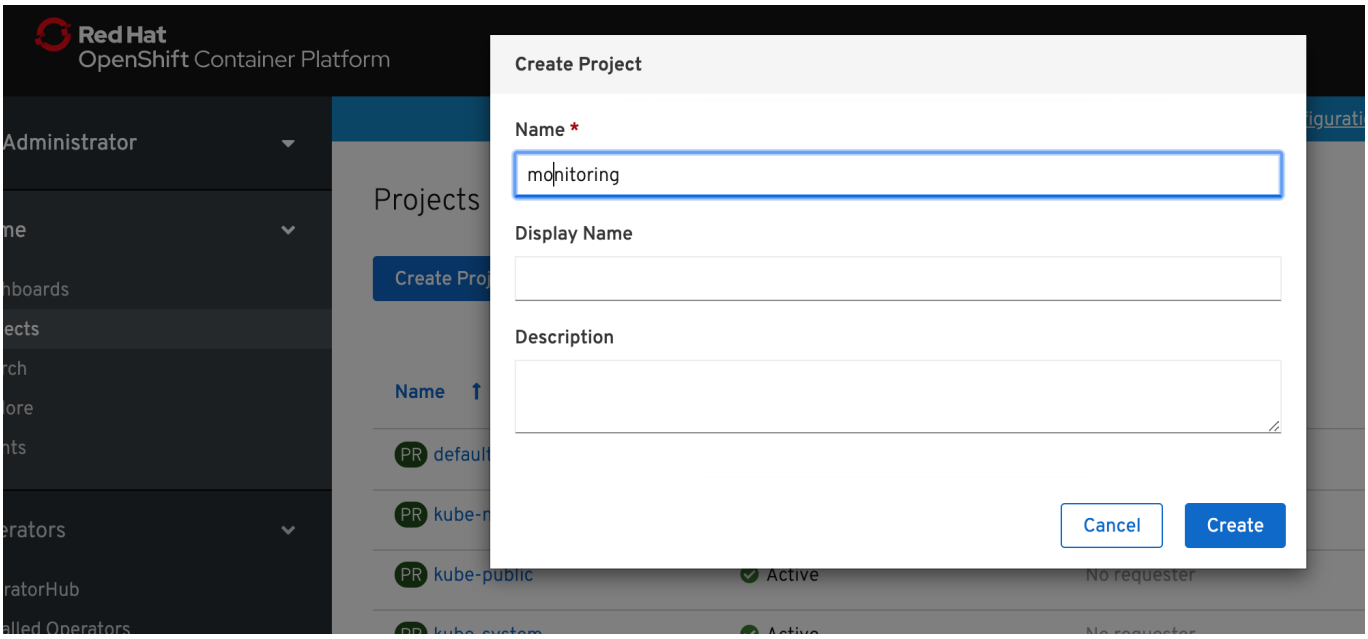
```
$ curl -k https://b2m-nodejs-b2m-nodejs.apps.rsocp.os.fyre.ibm.com

{"status":"ok","transactionTime":"353ms"}
```

Verify that it properly exposes metrics in Prometheus format:

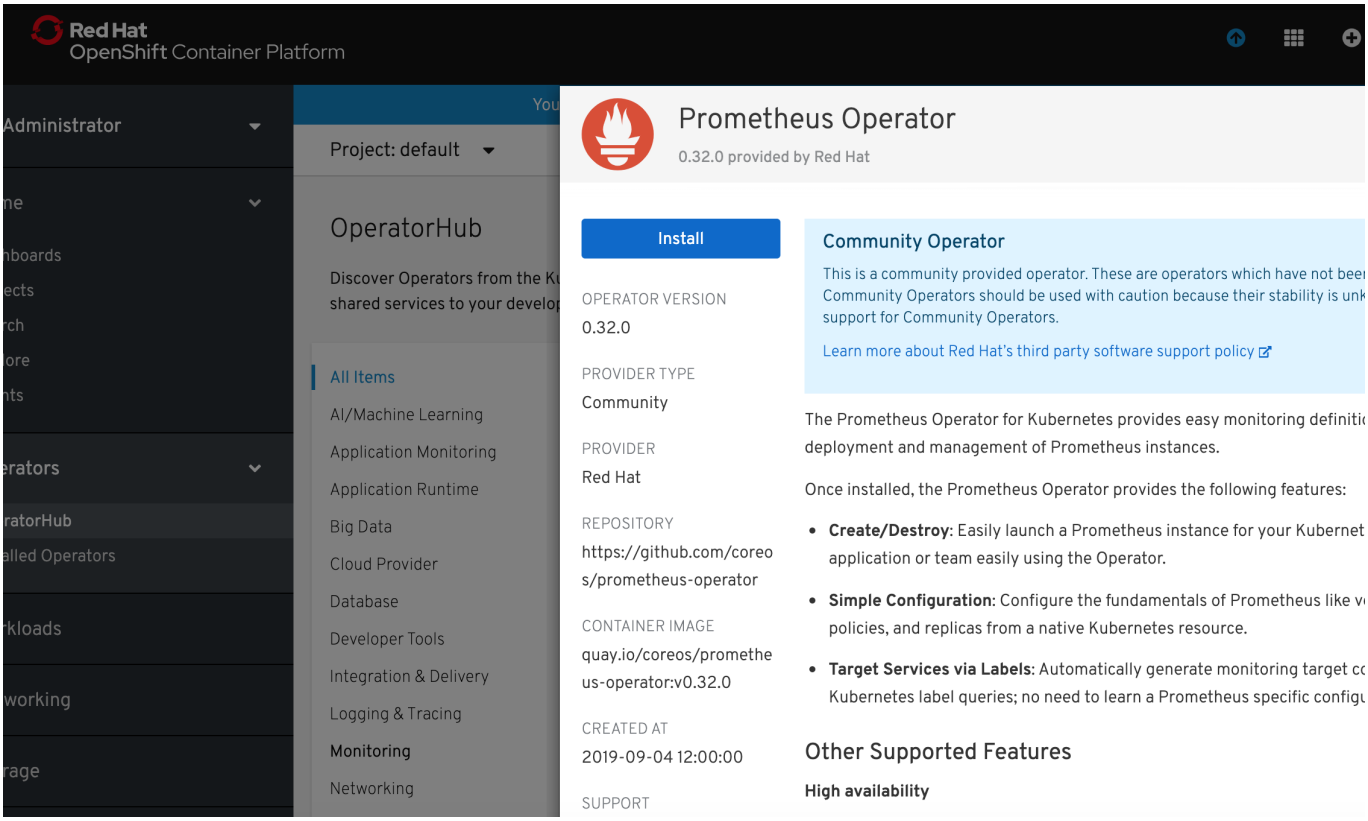
```
$ curl -k https://b2m-nodejs-b2m-nodejs.apps.rsocp.os.fyre.ibm.com/metrics

# HELP process_cpu_user_seconds_total Total user CPU time spent in
seconds.
# TYPE process_cpu_user_seconds_total counter
process_cpu_user_seconds_total 0.23436700000000005 1573764470969

# HELP process_cpu_system_seconds_total Total system CPU time spent in
seconds.
# TYPE process_cpu_system_seconds_total counter
process_cpu_system_seconds_total 0.069524 1573764470969

# HELP process_cpu_seconds_total Total user and system CPU time spent in
seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 0.3038910000000002 1573764470969
(...)
```

Deploy Prometheus monitoring stack for applications.

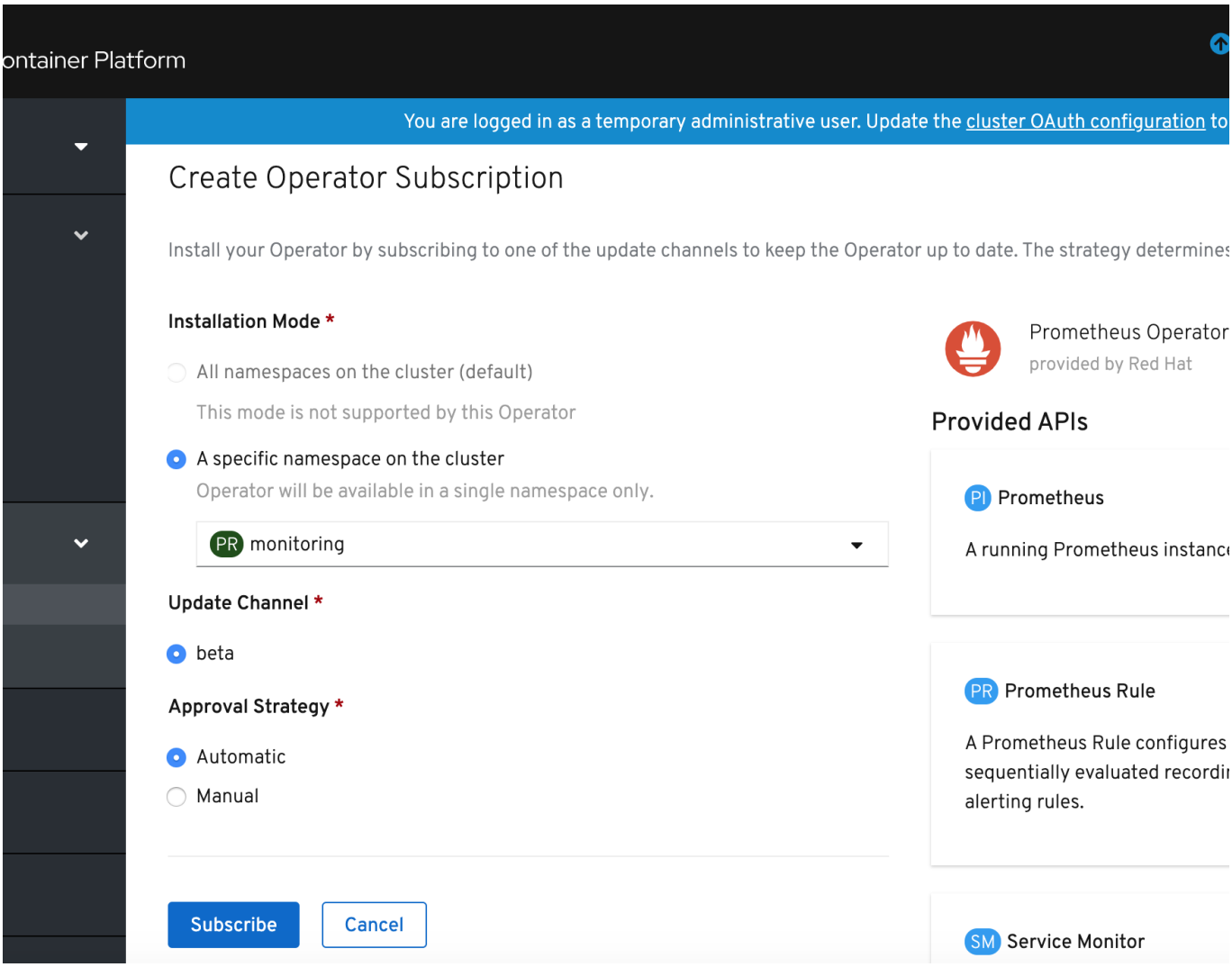
1). Create a new project for the Prometheus monitoring stack for applications.



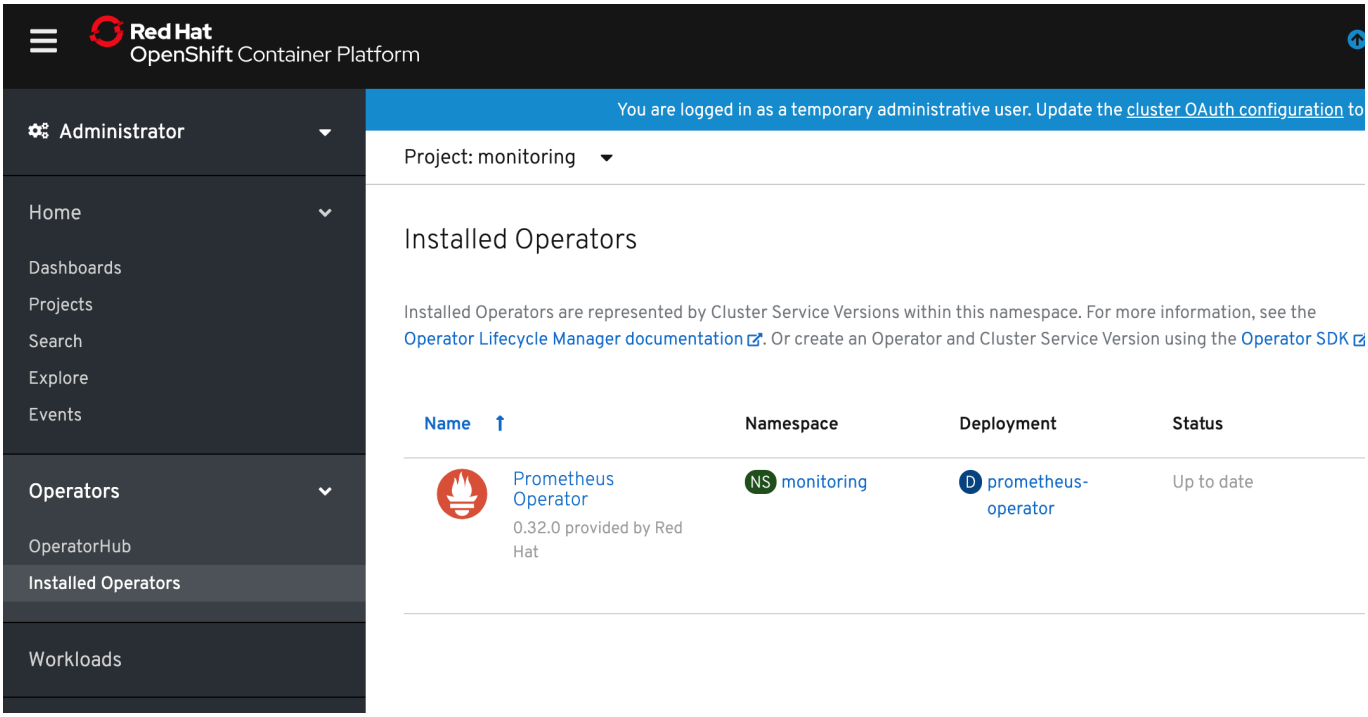
2). Select Operators -> Operator Hub and select `Prometheus Operator`. Click `Install`.



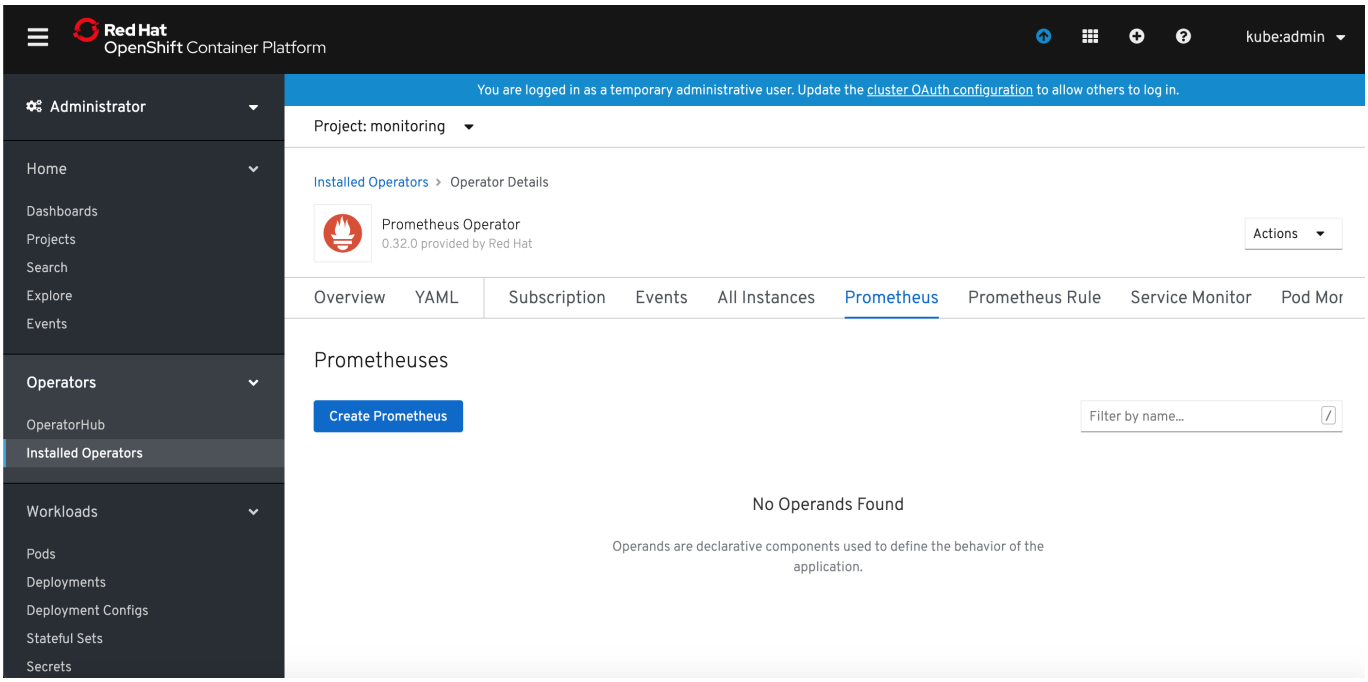
3). In the `Create Operator Subscription` window click `Subscribe`.



4). Wait until Prometheus Operator is deployed and click `Prometheus Operator` link.



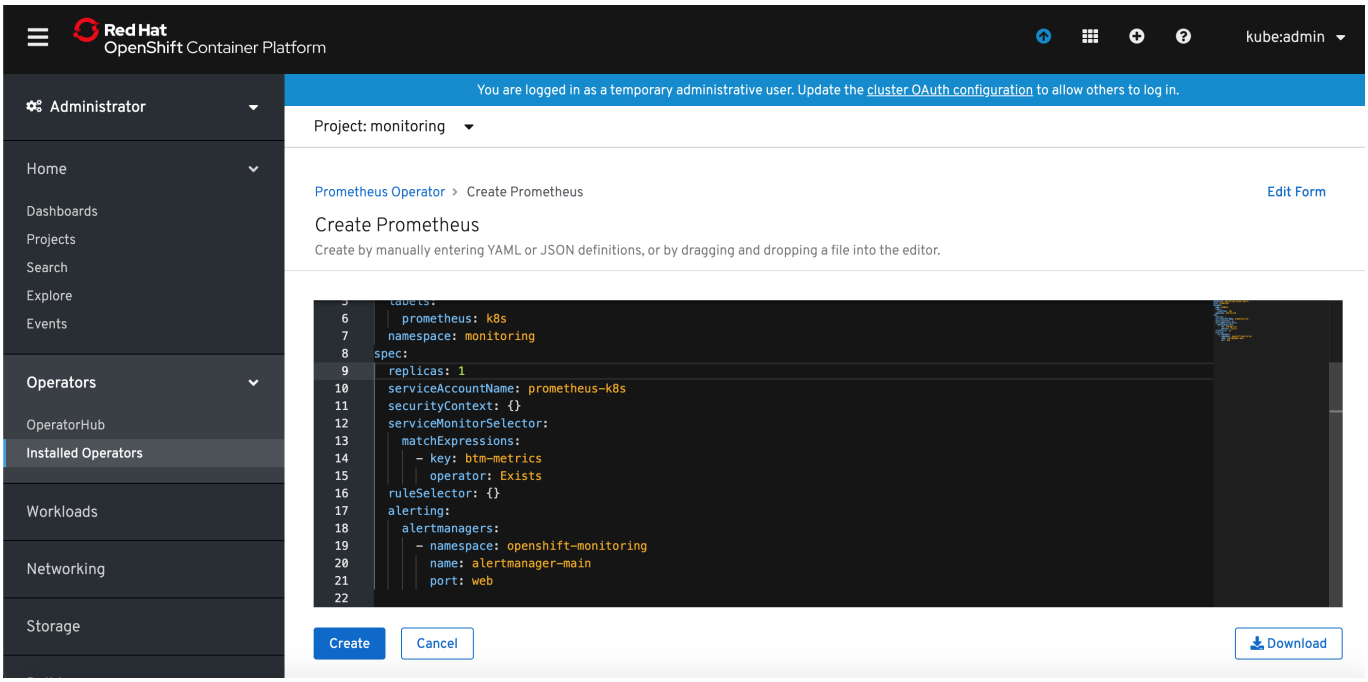
5). Select `Prometheus` tab and click `Create Prometheus` button.



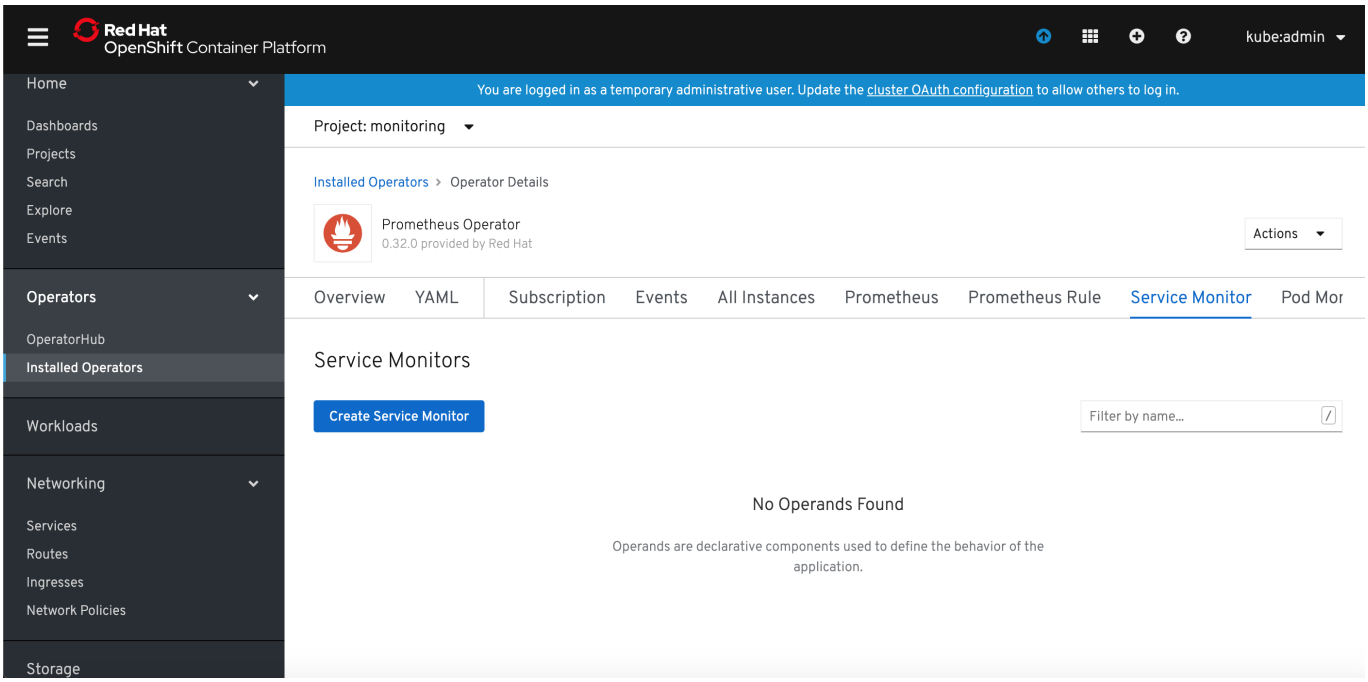
6). Modify default YAML template for Prometheus. I added `serviceMonitorSelector` definition which will instruct defined Prometheus instance to match `ServiceMonitors` with label `key=btm-metrics`. I also changed the Prometheus instance name to `app-monitor`.

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
    name: app-monitor
  labels:
    prometheus: k8s
  namespace: monitoring
spec:
  replicas: 1
  serviceAccountName: prometheus-k8s
  securityContext: {}
  serviceMonitorSelector:
    matchExpressions:
      - key: btm-metrics
        operator: Exists
  ruleSelector:
    matchLabels:
      prometheus: app-monitor
      role: alert-rules
  alerting:
    alertmanagers: {}
```

Click `Create` button.



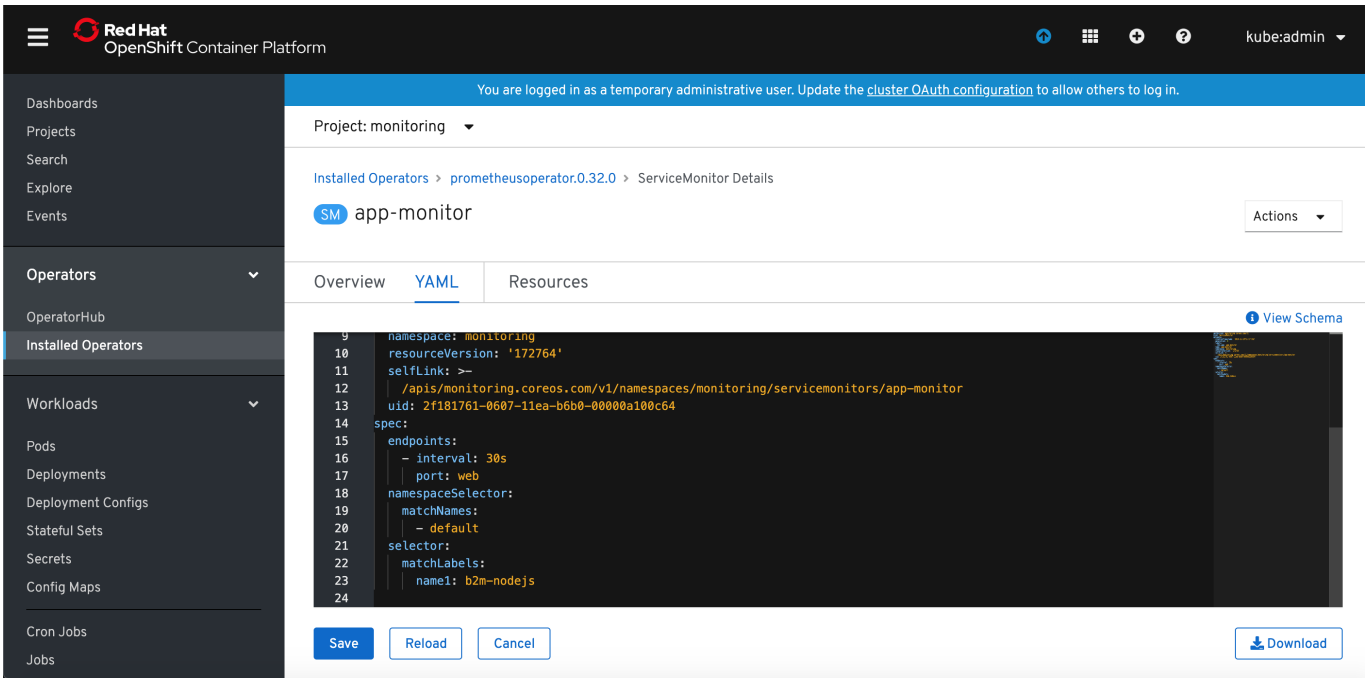
7). Select `Service Monitor` tab and click `Create Service Monitor`.



8). Modify default YAML template for ServiceMonitor. I added `namespaceSelector` definition to limit the scope to naespace `default` where my app has been deployed and modified `selector` that to look for services with label `name=b2m-nodejs`. I also changed the Service monitor name to `app-monitor`.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    btm-metrics: b2m-nodejs
  name: app-monitor
  namespace: monitoring
spec:
  endpoints:
```

```
    - interval: 30s
      port: web
  namespaceSelector:
    matchNames:
      - b2m-nodejs
  selector:
    matchLabels:
      name: b2m-nodejs
```



9). Grant `view` cluster role to the Service Account created by the operator and used by Prometheus.

```
oc adm policy add-cluster-role-to-user view
system:serviceaccount:monitoring:prometheus-k8s
```

or, if you want to limit it to the application namespace, add `view` role only to the app namespace:

```
oc adm policy add-role-to-user view
system:serviceaccount:monitoring:prometheus-k8s -n default
```

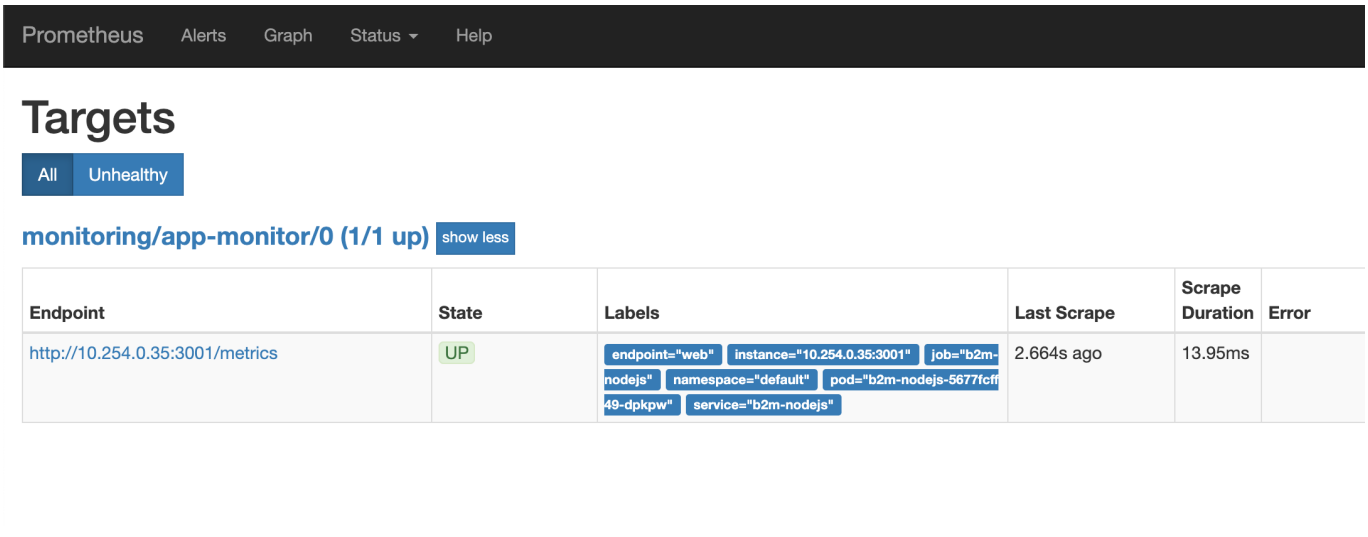
10). Expose app monitoring Prometheus route:

```
oc expose svc/prometheus-operated -n monitoring
```

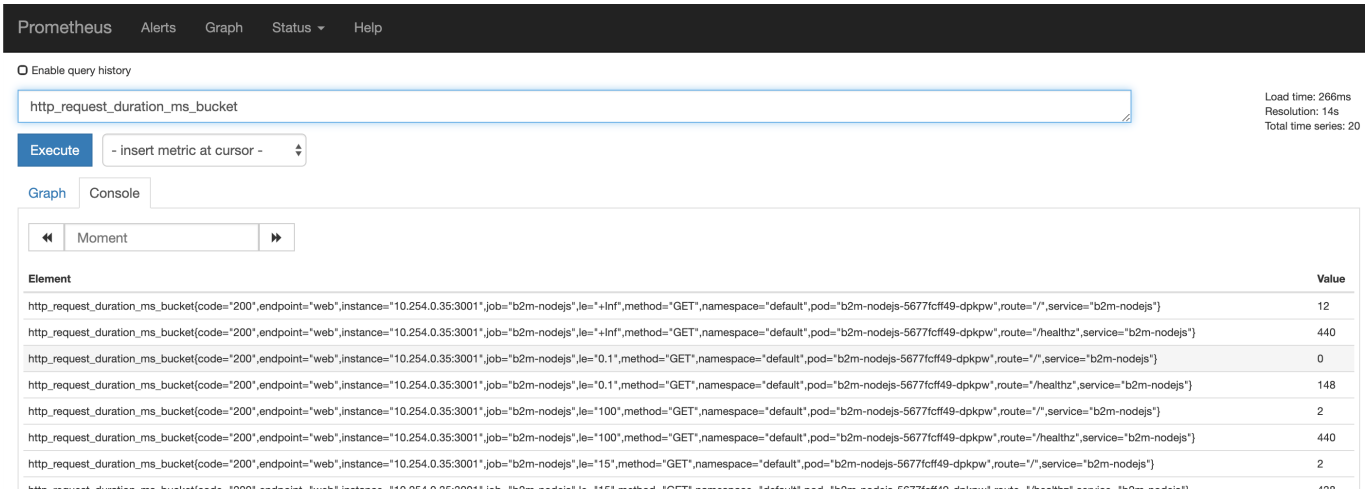
11). Collect the app monitoring Prometheus URL:

```
$ oc get routes -n monitoring
NAME                  HOST/PORT
PATH   SERVICES              PORT   TERMINATION   WILDCARD
prometheus-operated   prometheus-operated-
monitoring.apps.rsocp.os.fyre.ibm.com          prometheus-operated   web
None
```

12). Verify that app monitoring Prometheus can scrape `b2m-nodejs` app. Access the Prometheus URL via browser and select Status -> Targets.

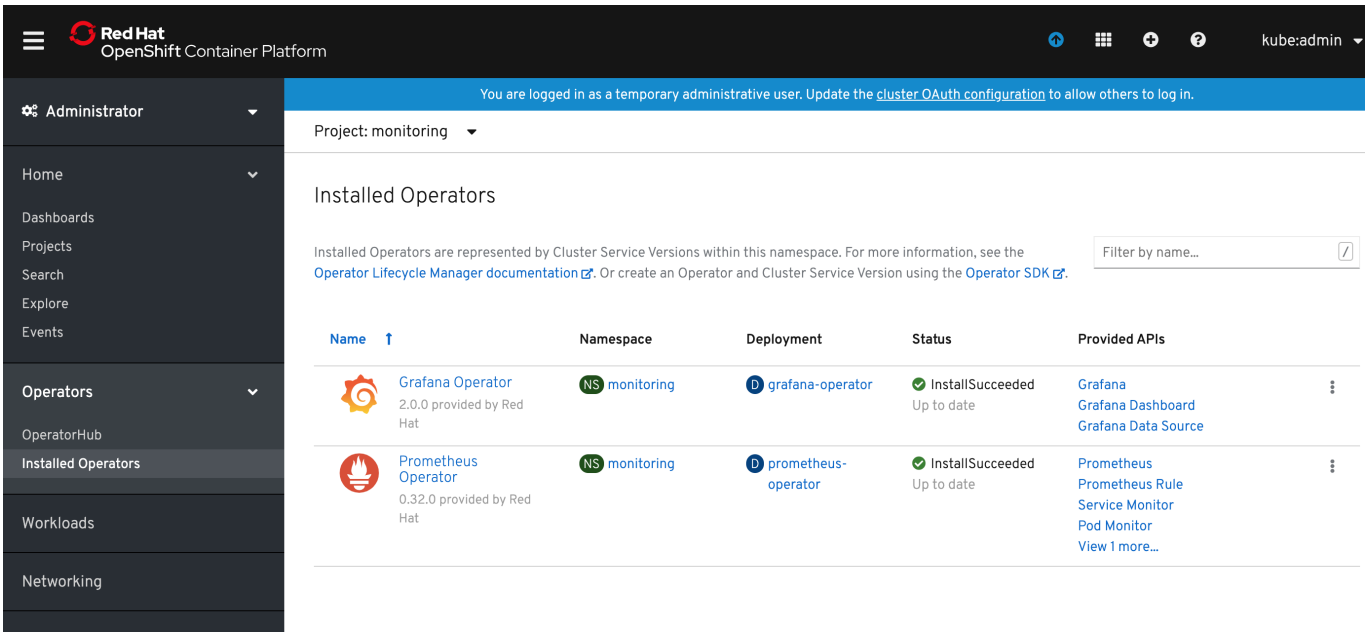


13). Verify that instrumented metrics are collected:

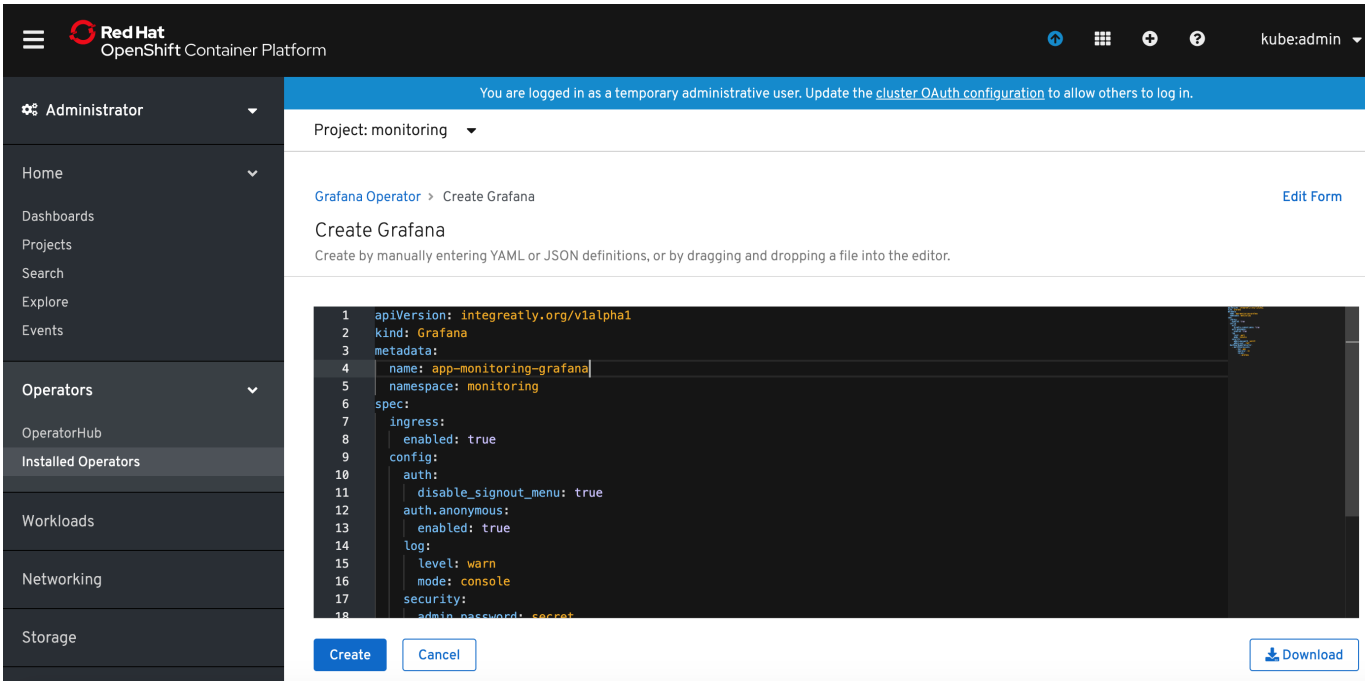


Deploy the Grafana Operator

1). Deploy the Grafana Operator from OperatorHub using the same steps as for Prometheus Operator. Now you should see it in `Operators -> Installed Operators`.

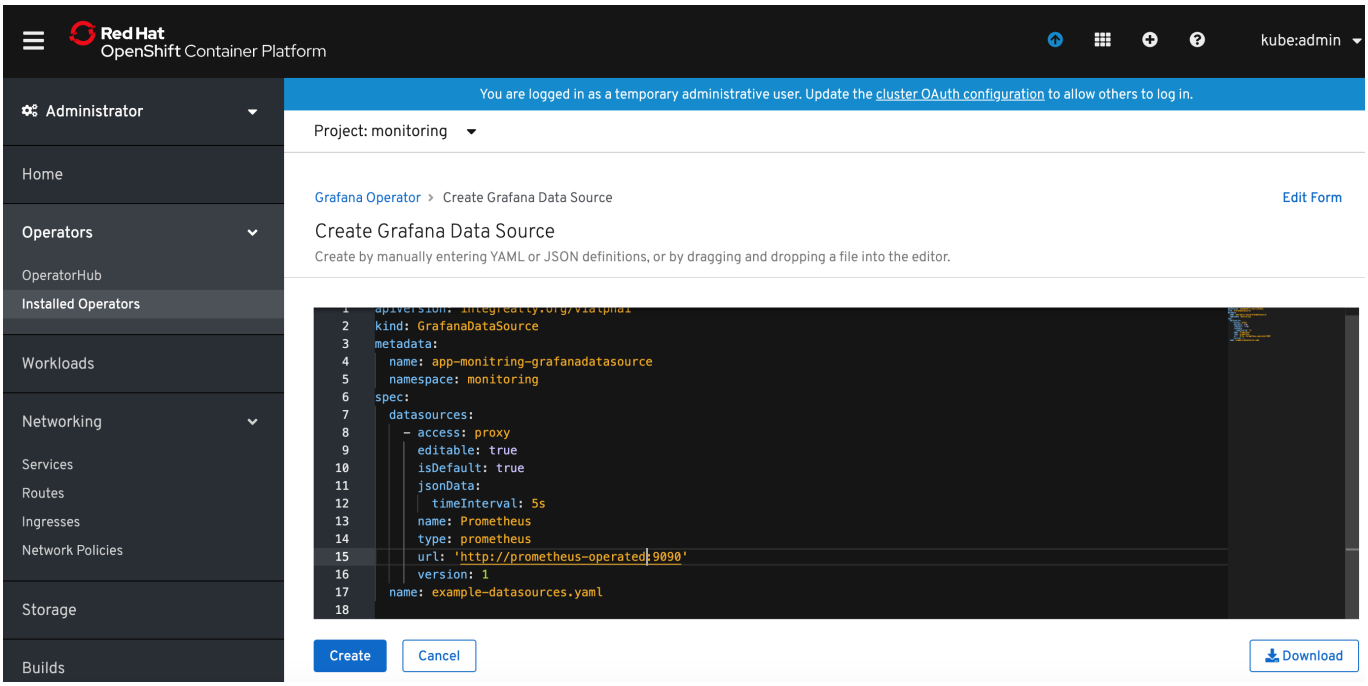


2). Click on the Grafana Operator link, select `Grafana` tab and click `Create Grafana`.



3). Modify the name of the Grafana instance to something meaningful. I named it `app-monitoring-grafana`. Click `Create` button. Modify also the admin user name and password.

4). Return to the Grafana Operator details, select `Grafana Data Source` and click `Create Grafana Data Source` button. Rename the `name:` to something meaningful (I named it `app-monitoring-grafana-datasource`) and modify spec.datasources.url to your app monitoring prometheus instance. In my case it was `http://prometheus-operated:9090`.



The prometheus hostname is the same as the app monitoring prometheus service name. You can find it in `Networking->Services` (filtered by the project where app monitoring prometheus has been deployed).

5). Make the route for Grafana has been created in `Networking->Routes` (project `monitoring`). If it is not listed, create it with command:

```
oc create route edge  --service=grafana-service -n monitoring
```

Access the Grafana console URL and logon to Grafana.
Verify the Prometheus datasource has been created and can connect to app monitoring Prometheus.

6). Import provided grafana dashboard: `b2m-nodejs-v2/lab-4/app-monitoring-dashboard.json`.

7). Verify that Grafana dashboard has been provisioned:

Deploy Alertmanager

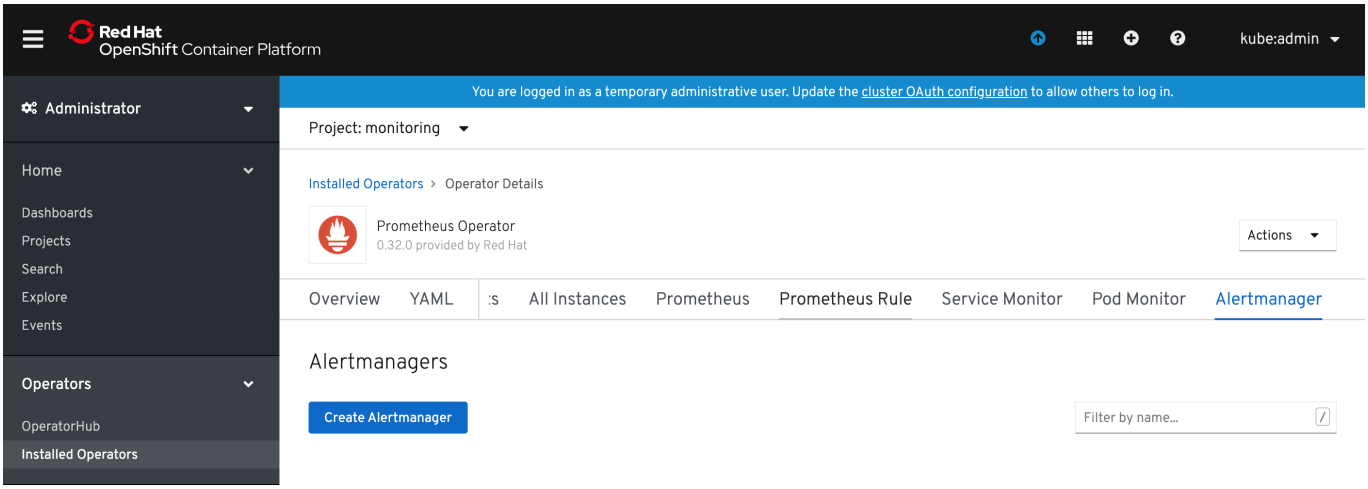
The Prometheus Operator introduces an `Alertmanager` resource, which allows users to declaratively describe an Alertmanager cluster. To successfully deploy an Alertmanager cluster, it is important to understand the contract between Prometheus and Alertmanager.

The Alertmanager may be used to:

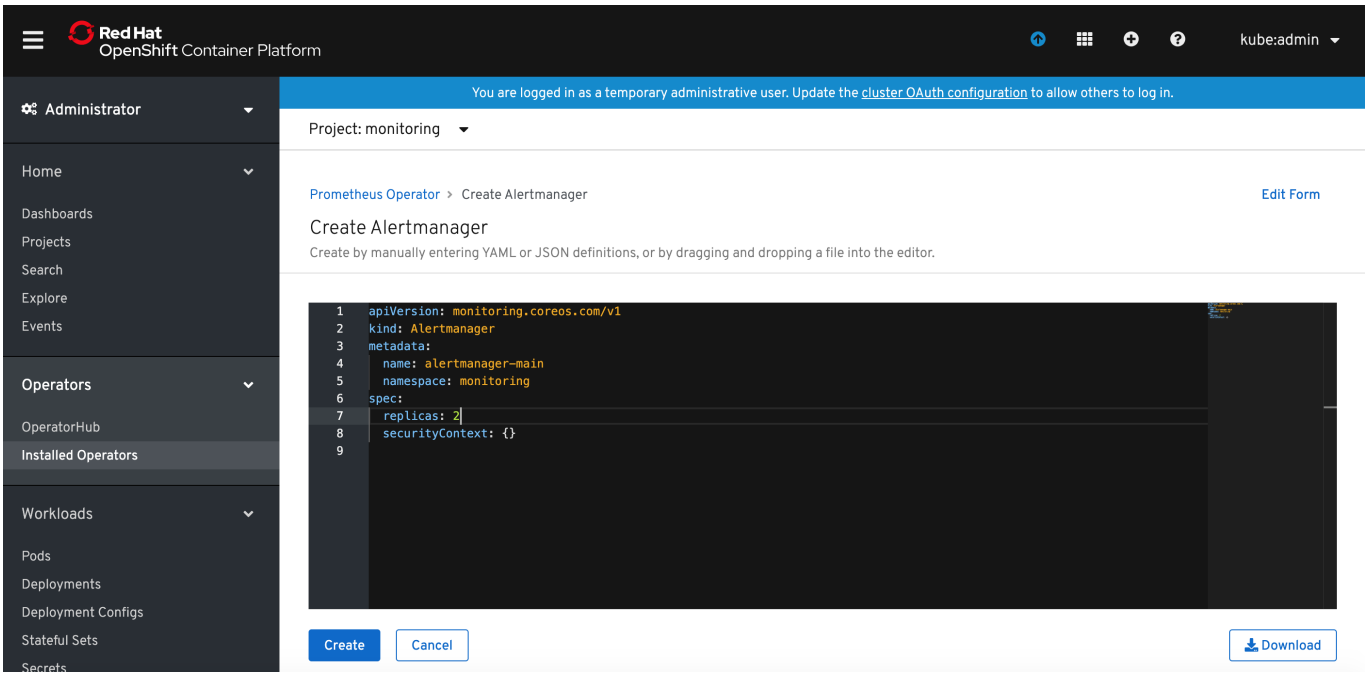
- Deduplicate alerts fired by Prometheus
- Silence alerts
- Route and send grouped notifications via providers (PagerDuty, OpsGenie, Slack, Netcool Message Bus Probe, etc.)

Prometheus' configuration also includes "rule files", which contain the alerting rules. When an alerting rule triggers, it fires that alert against all Alertmanager instances, on every rule evaluation interval. The Alertmanager instances communicate to each other which notifications have already been sent out.

In OpenShift console go to Installed Operators, click on Prometheus Operator instance, scroll tabs to Alertmanager tab.
Click `Create Alertmanager` button.



Specify the desired number of replicas and click `Create` button.



Now you can list the resources of Alertmanager and you should see Alertmanager pods in `Pending` state. This is because Alertmanager can't run wthout a configuration file.

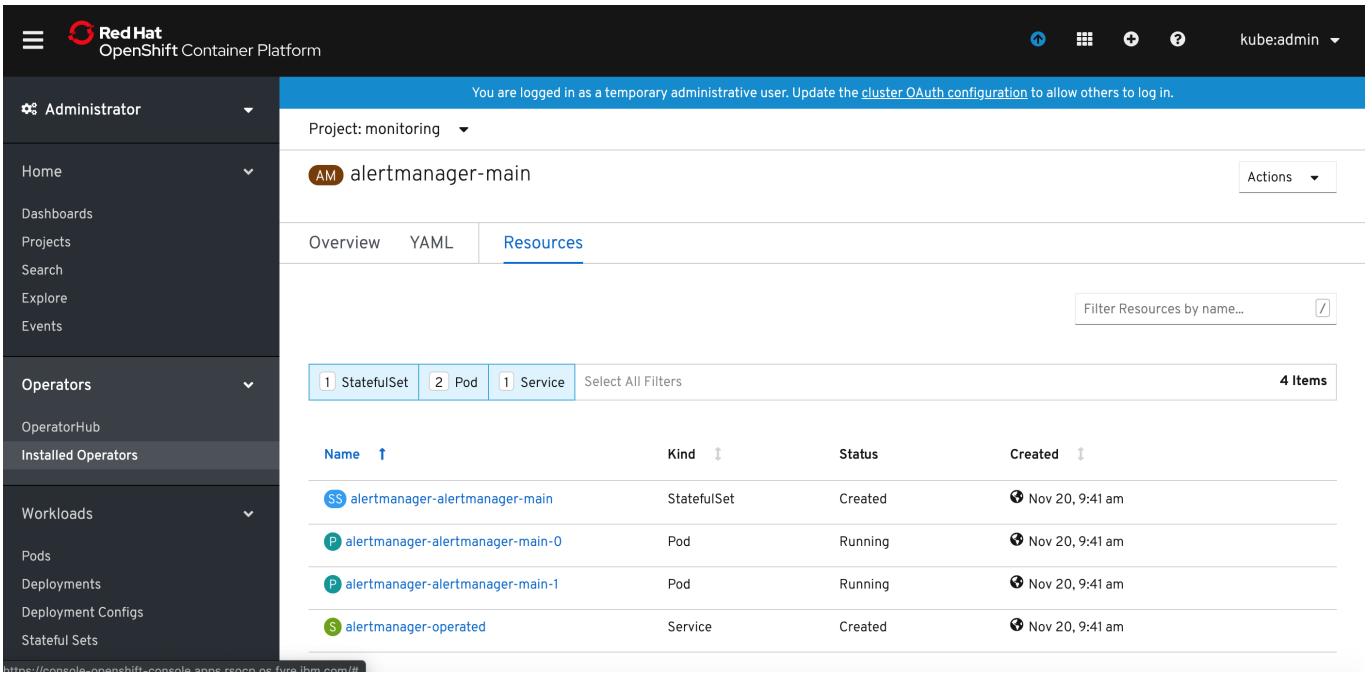
The Alertmanager instances will not be able to start up, unless a valid configuration is given. The following example configuration sends notifications against a non-existent webhook, allowing the Alertmanager to start up, without issuing any notifications. For more information on configuring Alertmanager, see the Prometheus Alerting Configuration document.

```
global:
  resolve_timeout: 5m
route:
  group_by: ['job']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 12h
  receiver: 'webhook'
receivers:
- name: 'webhook'
  webhook_configs:
  - url: 'http://alertmanagerwh:30500/'
```

Save the above Alertmanager config in a file called alertmanager.yaml and create a secret from it using `oc`.

```
oc create secret generic alertmanager-alertmanager-main --from-
file=alertmanager.yaml
```

Alertmanager pods should change the status to `Running`.



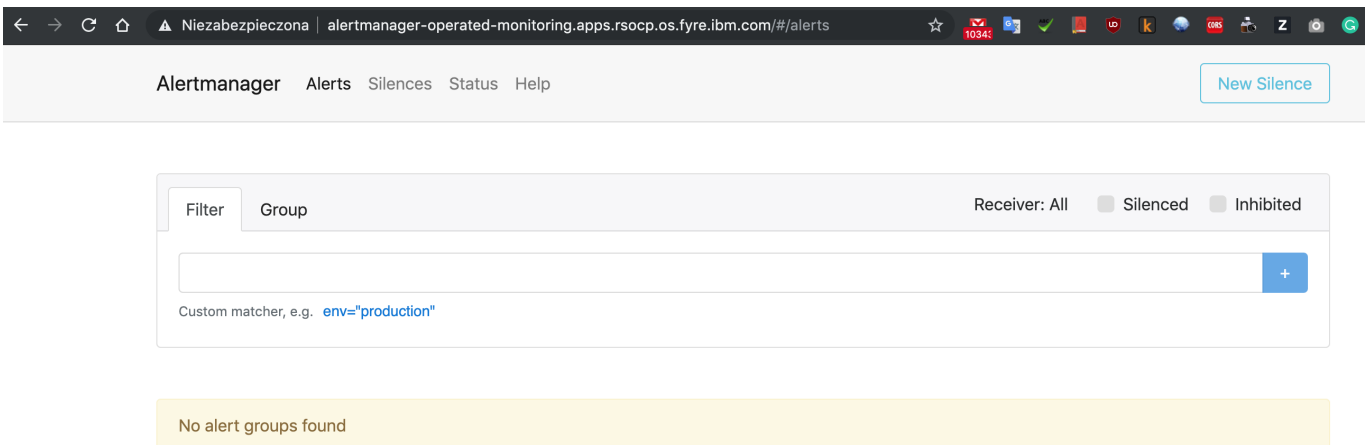
The service `alertmanager-operated` has been created automatically and if you want to externally expose Alertmanaget UI, create the route using the followng command:

```
$ oc create route edge  --service=alertmanager-operated -n monitoring
```

Collect the Alertmanager URL:

```
 oc get routes alertmanager-operated
NAME                    HOST/PORT
PATH   SERVICES                 PORT   TERMINATION   WILDCARD
alertmanager-operated   alertmanager-operated-
monitoring.apps.rsocp.os.fyre.ibm.com          alertmanager-operated   web
edge          None
```

and verify using web browser:



This Alertmanager cluster is now fully functional and highly available, but no alerts are fired against it. Configure Prometheus resource to fire alerts to our Alertmanager cluster.

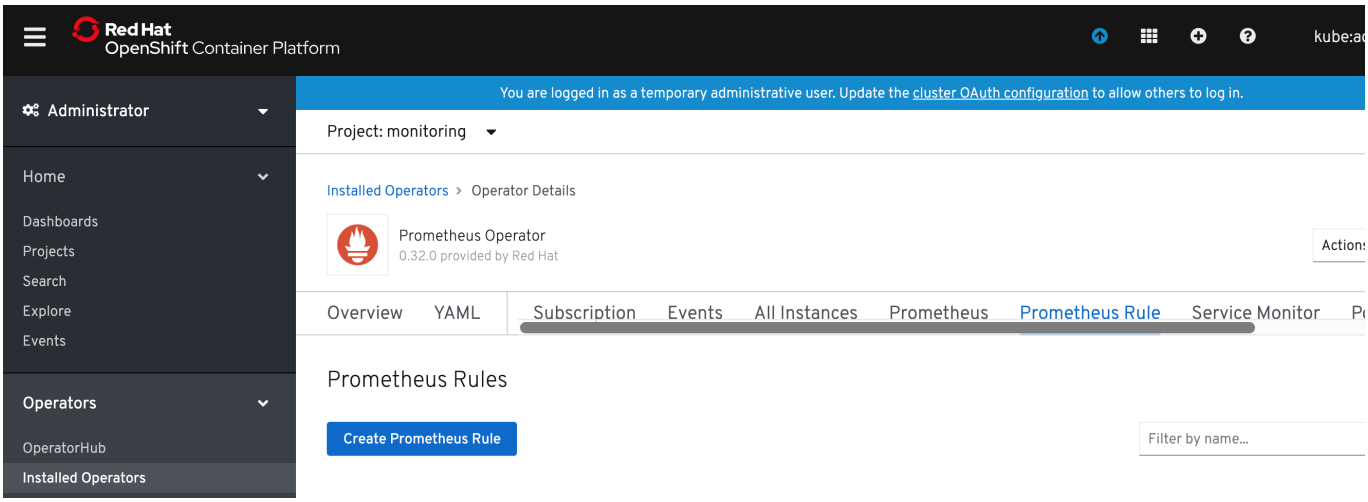
Edit Prometheus resource `spec.alerting` section:

```
spec:
  alerting:
    alertmanagers:
      - name: alertmanager-operated
        namespace: monitoring
        port: web
```

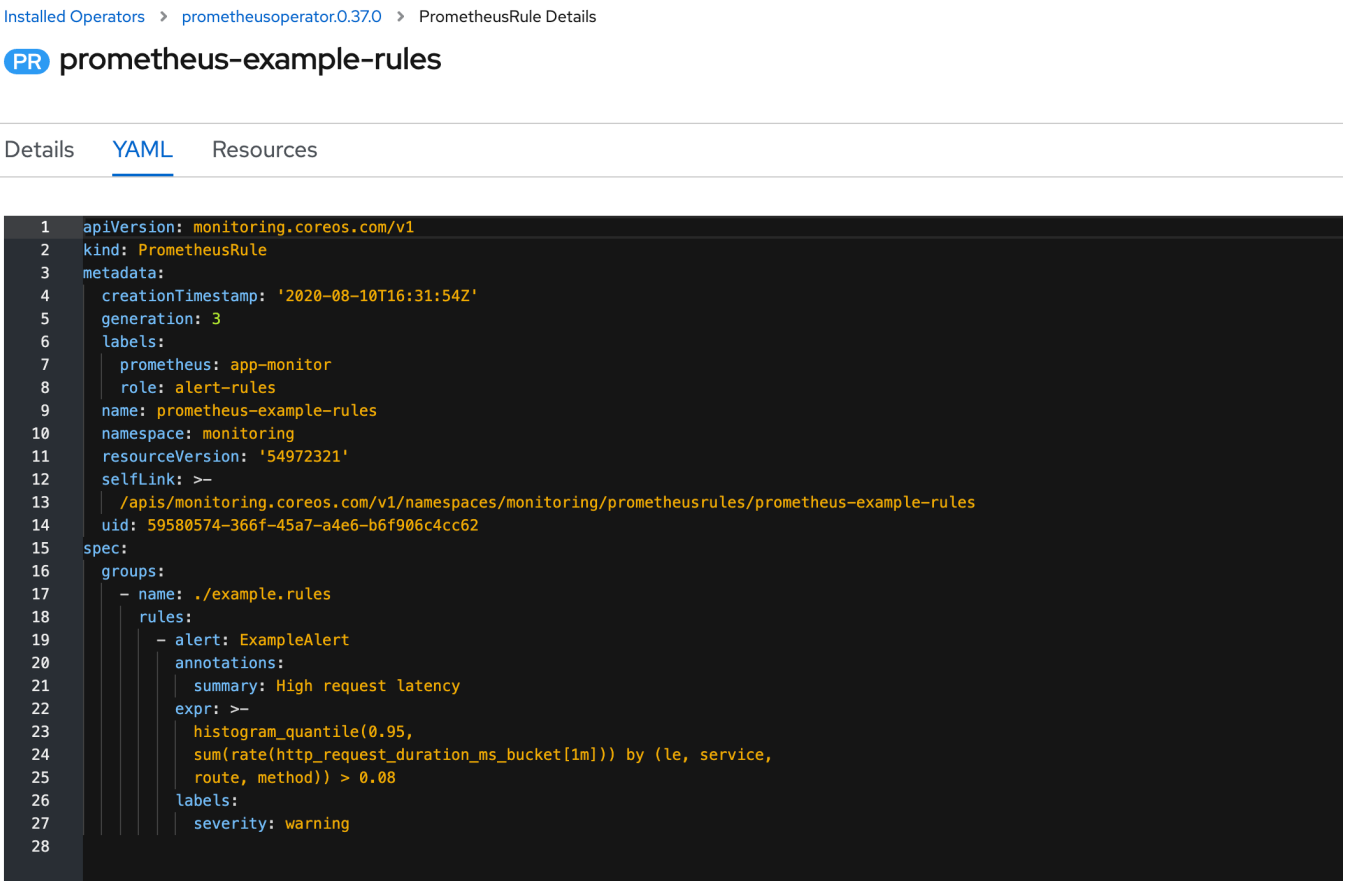
and click `Save`.

Configure Alerting Rules

Alerting Rules for application monitoring can be created from the `Operator Details` view of our Prometheus Operator instance. Click on the `Prometheus Rule` tab and then on `Create Prometheus Rule` button.



Specify alert rule definition in the YAML file. You can use provided ExampleAlert.yaml as an example.



After short time verify that your alert(s) have been activated using Prometheus UI: