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`.

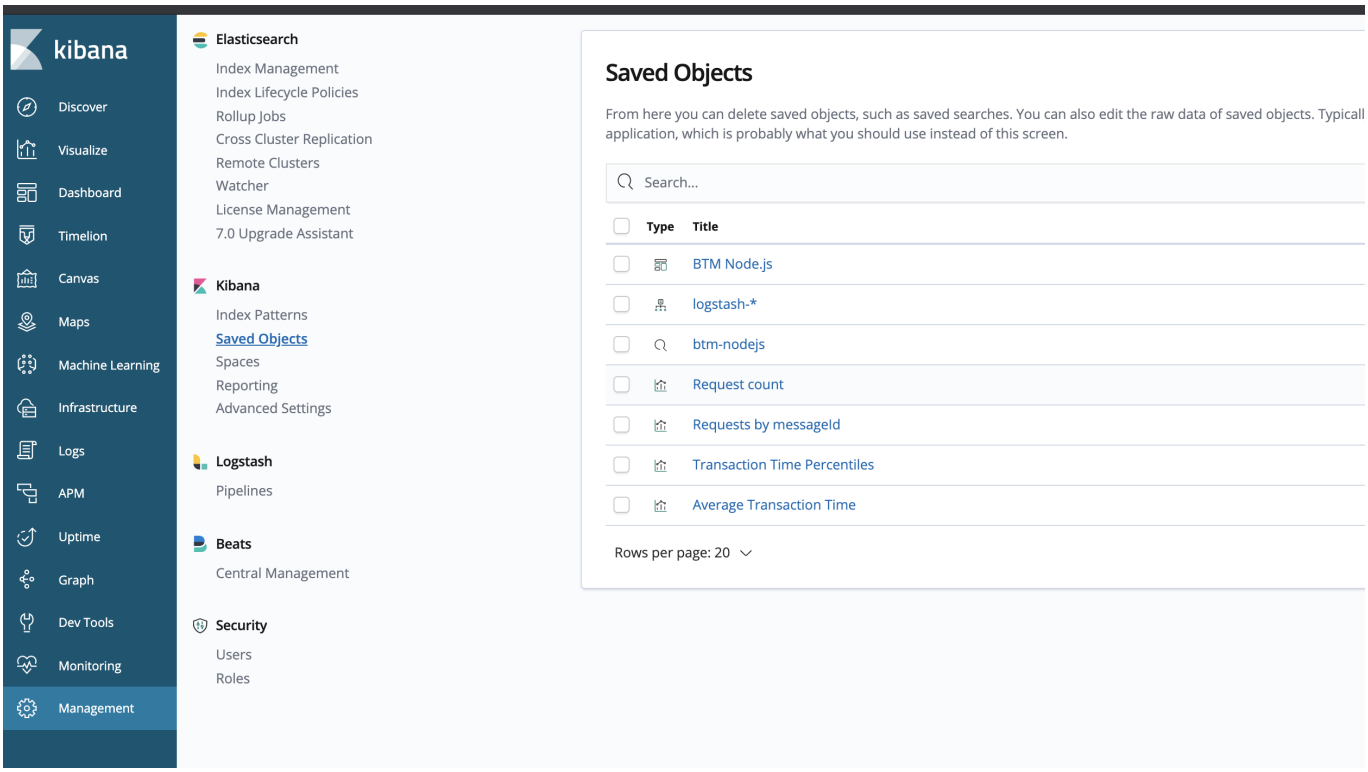
Logon using user: `elastic`, password: `changeme`

> 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
```

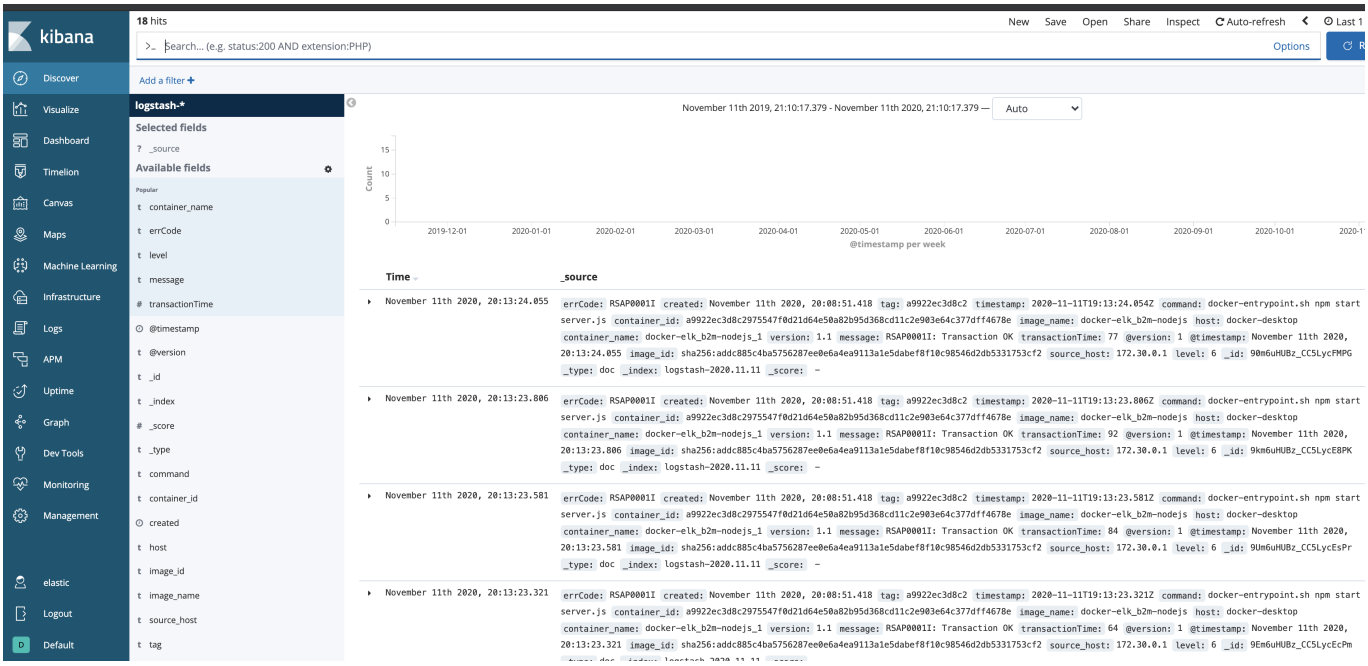
4). Simulate a couple of transactions using your web browser or `curl` by accessing `http://<your-hostname>:3001/checkout`:

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

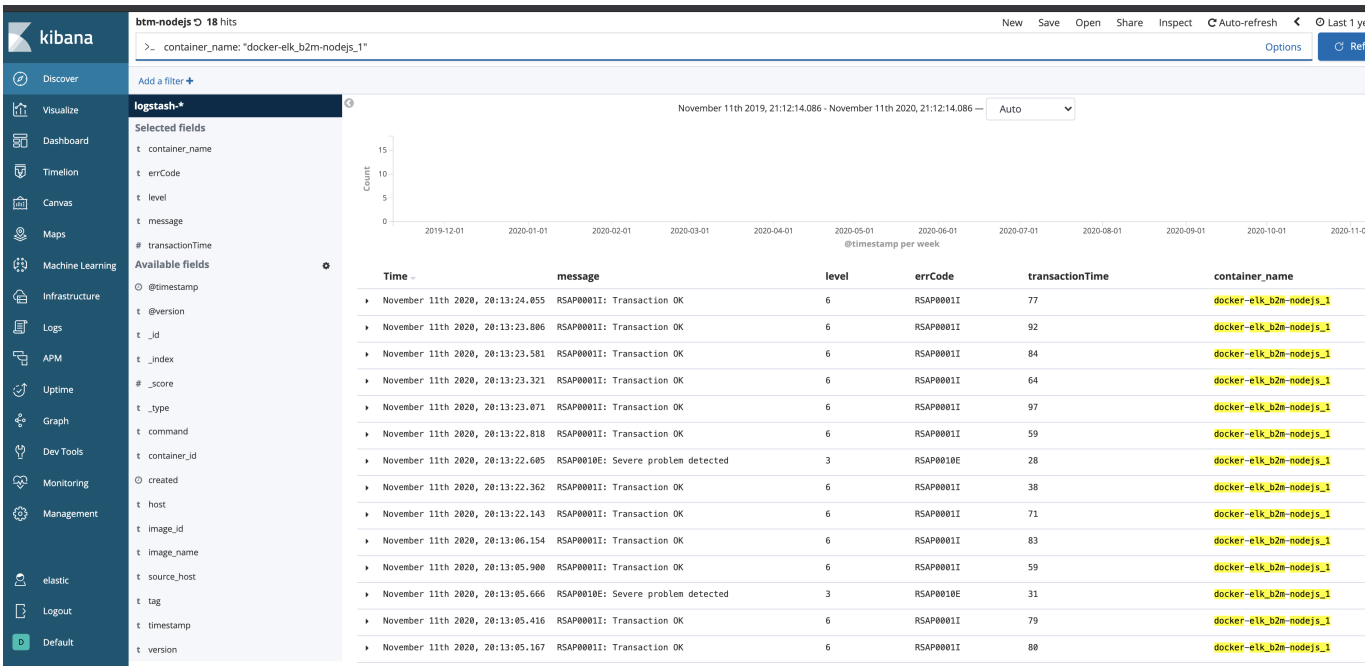
> If you run the lab locally, you can use the `localhost`

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

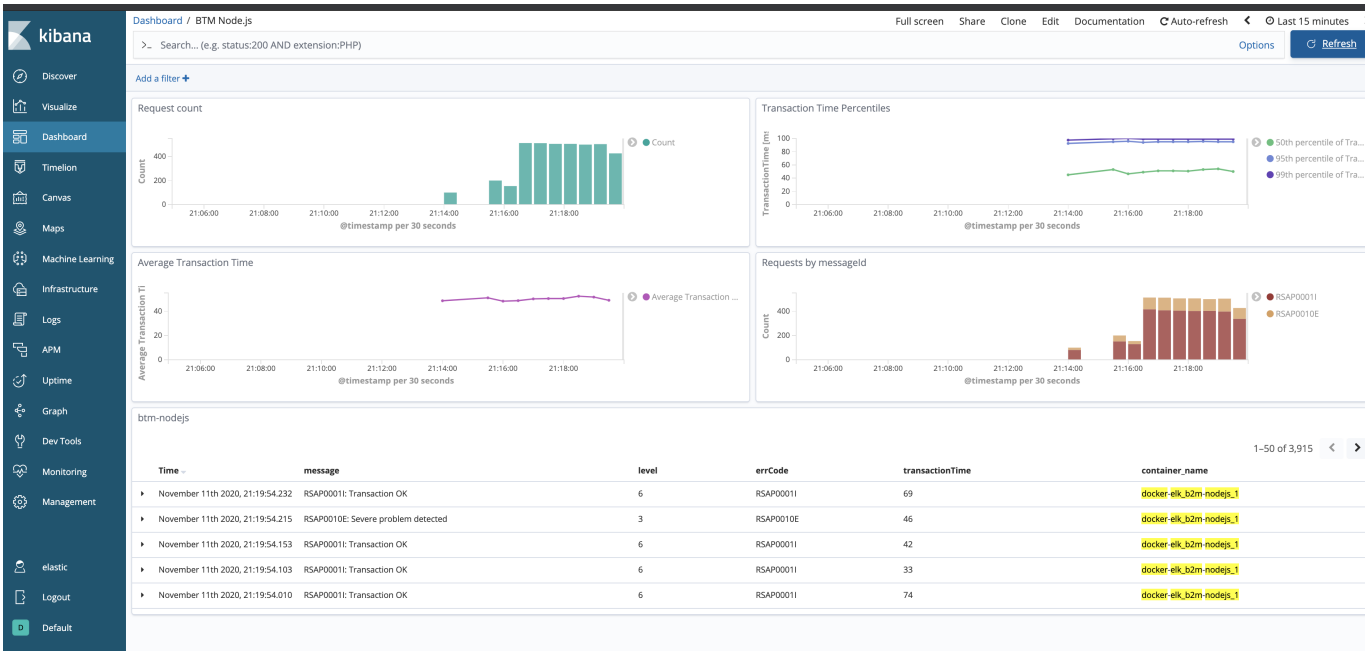
- The **Discover** view should be similar to:



- 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.

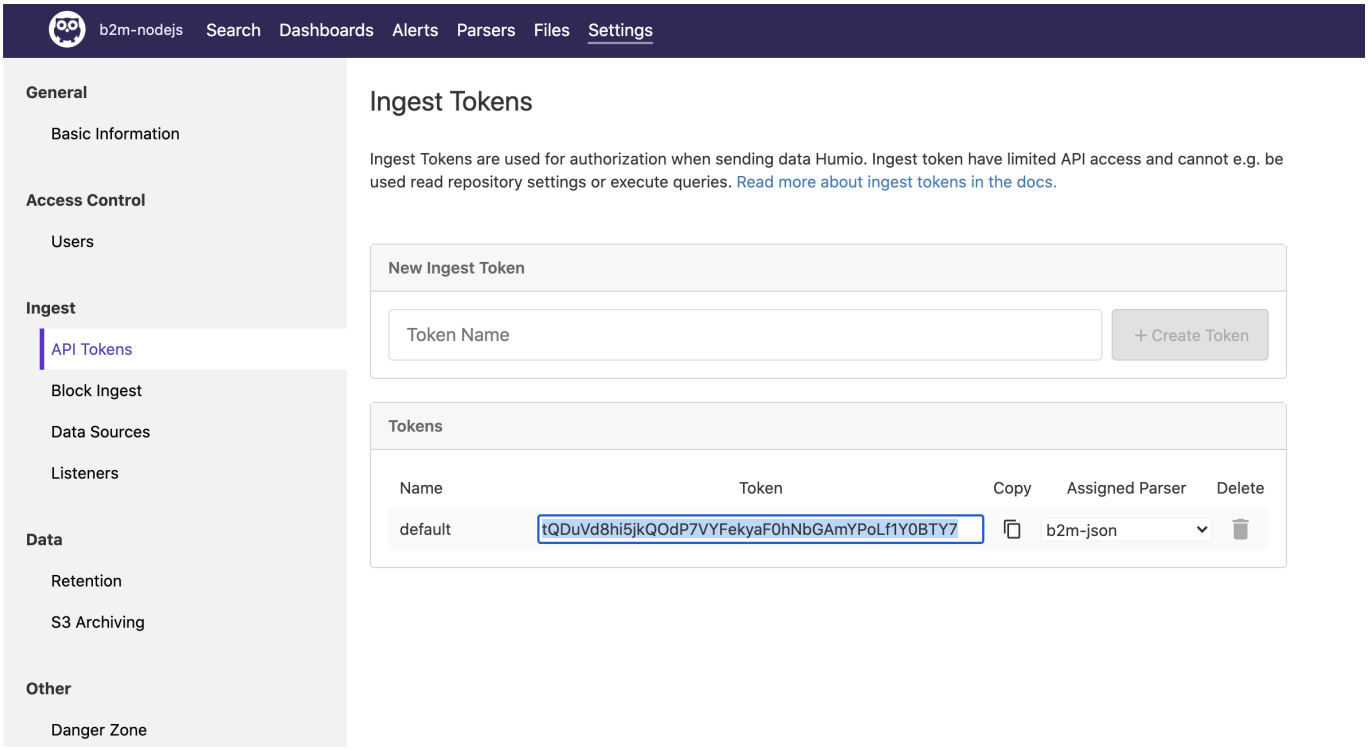
- `b2m-nodejs-v2/lab-2/docker-compose.yaml` - this is the main config file for docker-compose stack which specifies all options for all containers in the stack.

- `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`.

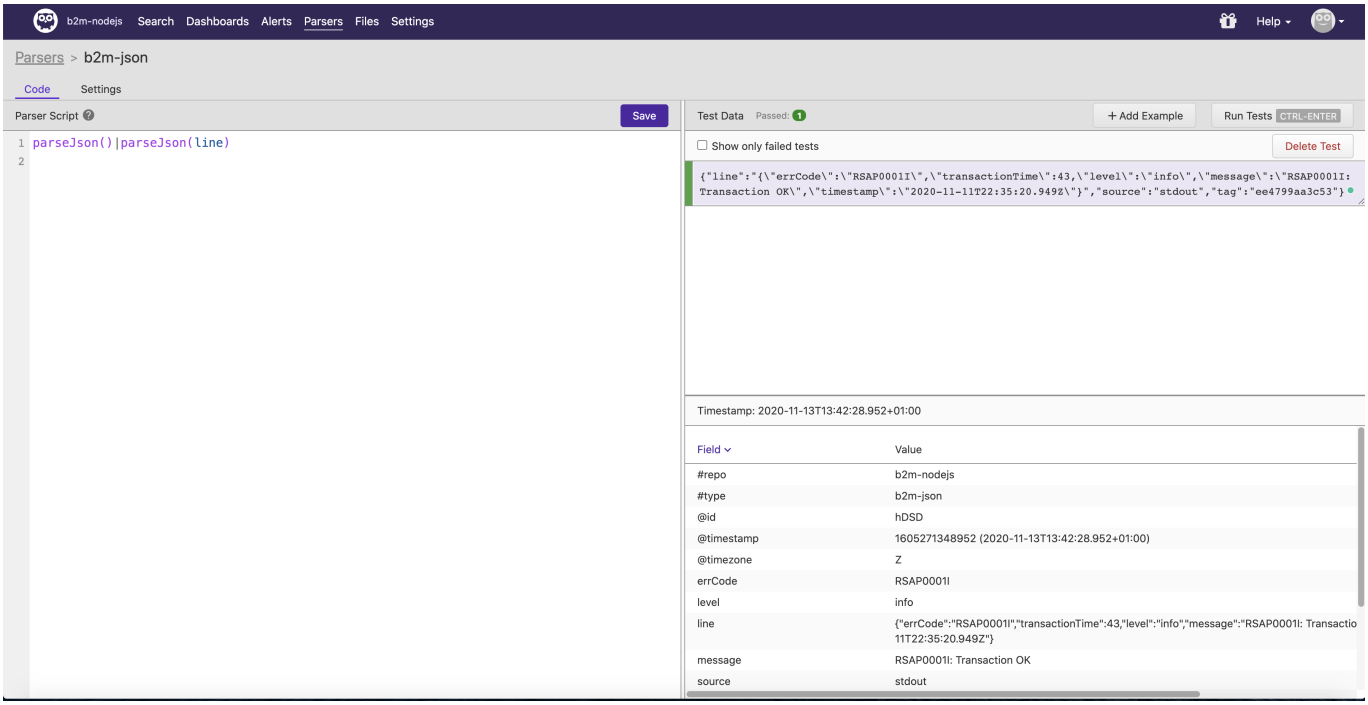
11). Replace the default content on the left side with the following parser script:

```
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):

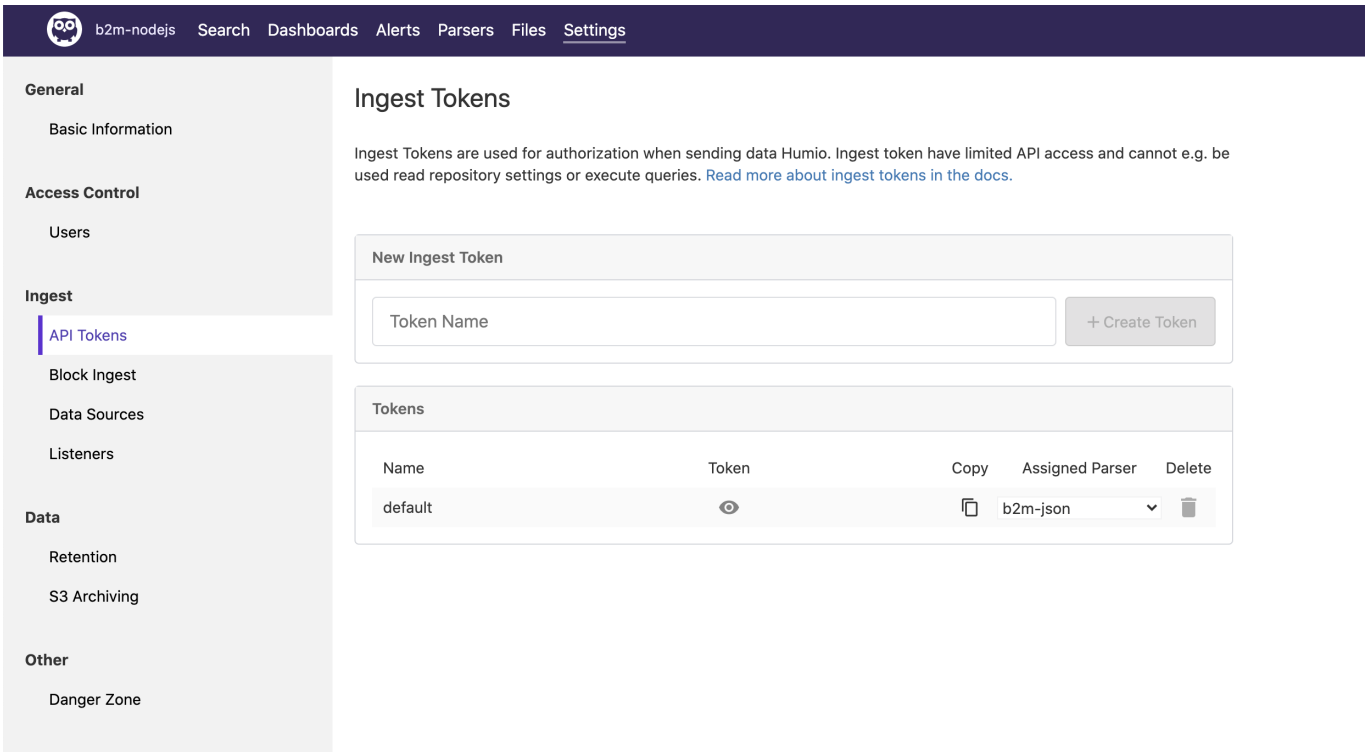
```
{"line":"
{\"errCode\":\"RSAP0001I\",\"transactionTime\":43,\"level\":\"info\",\"mes
sage\":\"RSAP0001I: Transaction OK\",\"timestamp\":\"2020-11-
11T22:35:20.949Z\"}","source":"stdout","tag":"ee4799aa3c53"}
```

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`

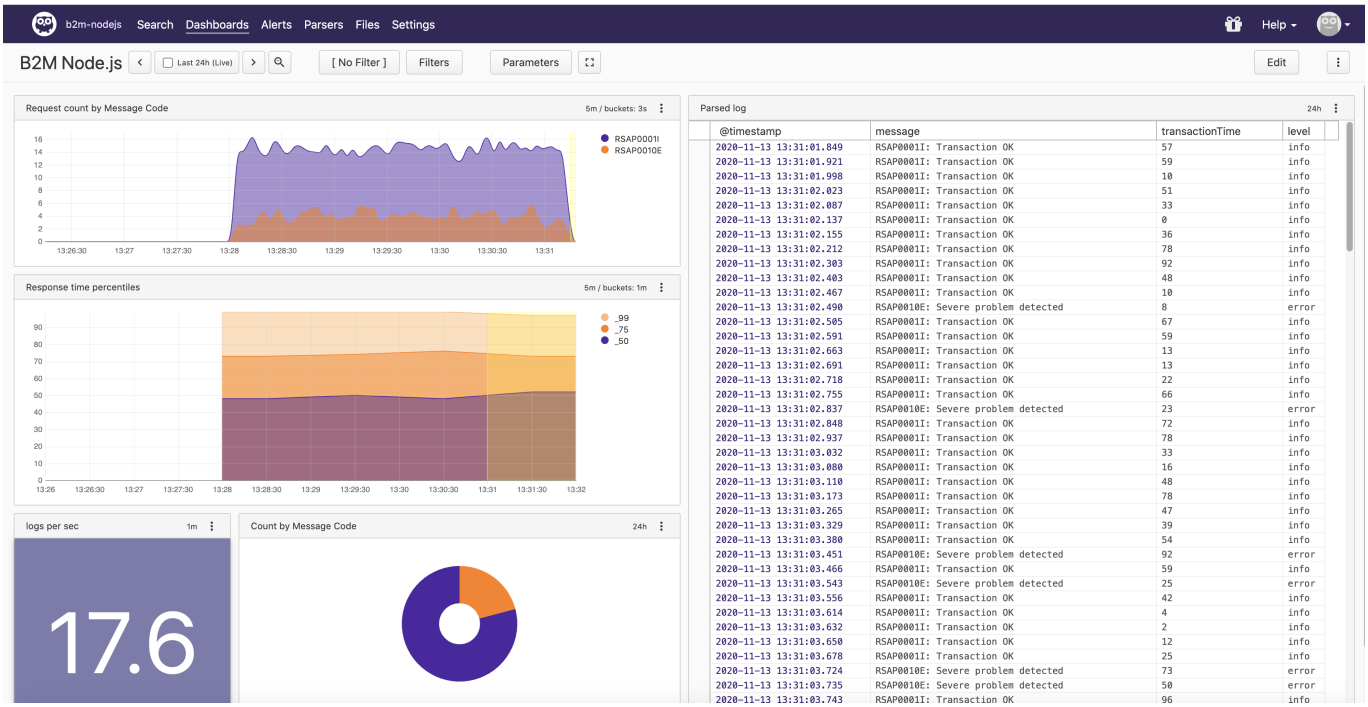
Verify results in the `Search` view.

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

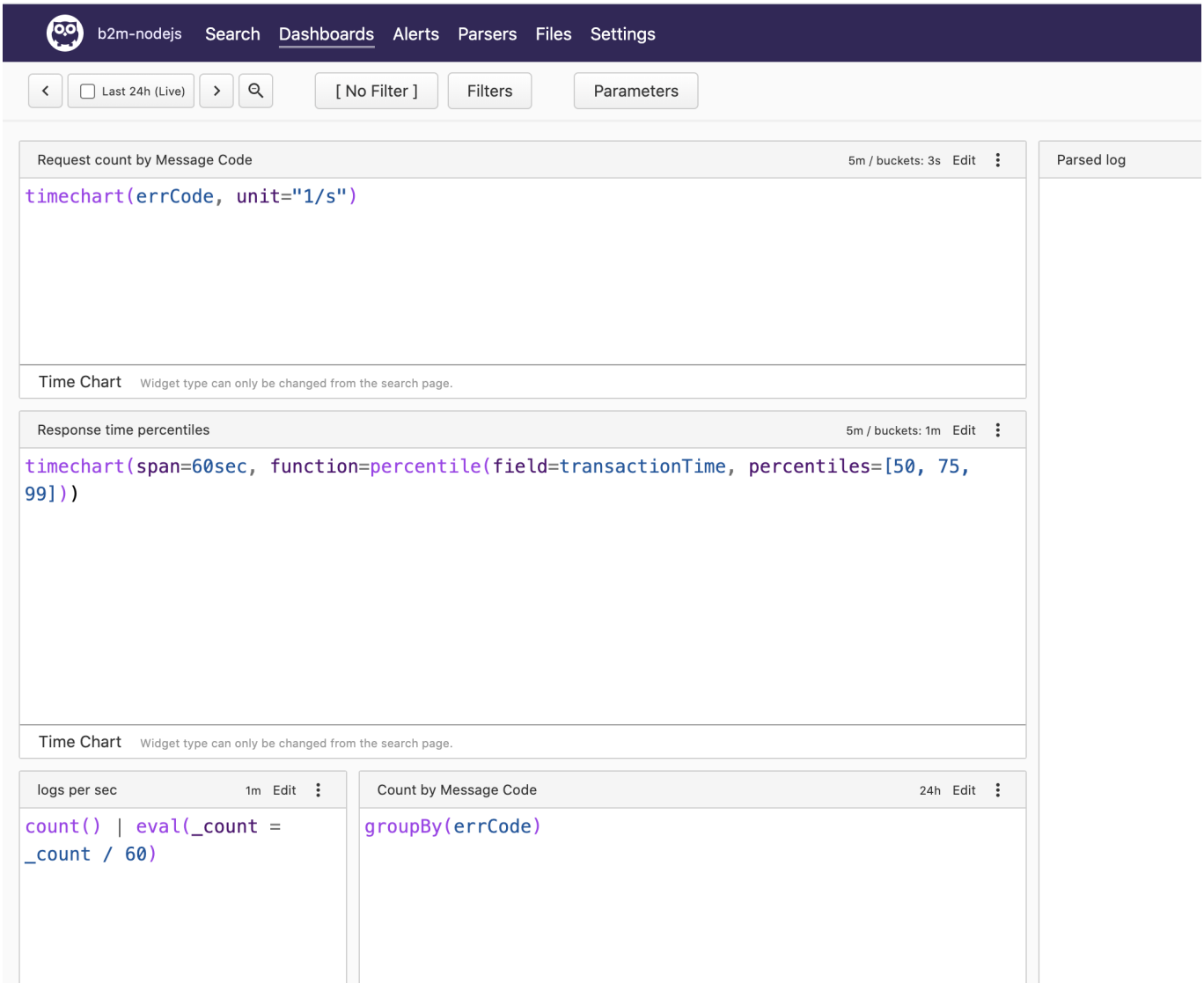
- Go to `Dashboards`, click `+ New Dashboard`
- Name your dashboard `B2M Node.js` and select `Template file` option.
- Click `Upload Template` and select `b2m-nodejs-v2/lab-2/humio-dashboard.yaml`
- Click `Create Dashboard`

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
```