

Zero to Cloud-Native with IBM Cloud

Part 10: Logging the Application

Kevin Collins

kevincollins@us.ibm.com

Technical Sales Leader

IBM Cloud Enterprise Containers – Americas

Kunal Malhotra

kunal.malhotra3@ibm.com

Cloud Platform Engineer

IBM Cloud MEA

1 - Introduction

Now that we have deployed the application and have it running, we can turn to day 2 operations aspects of running the application. One of the first things you will want to do is enable logging. IBM Log Analysis with LogDNA offers administrators, DevOps teams, and developers advanced features to filter, search, and tail log data, define alerts, and design custom views to monitor application and system logs.

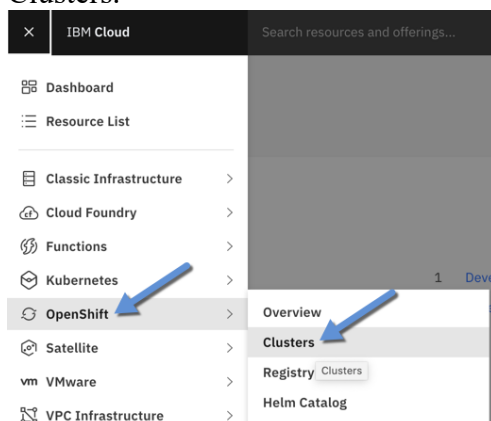
In this tutorial we will be focusing both on the developer view of logging in being able to troubleshoot the code itself and the DevOps team in looking holistically at the application. We will be creating specific reporting views of each microservice, graphical representation of the application performance, alerting when something goes wrong, and finally exporting a subset of the logs to an end-user with relevant information.

To begin with, let's look at the logging dashboard.

Note: This tutorial assumes you have already completed Part 7, section 1-1 where you have created a LogDNA instance and connected it your OpenShift cluster.



1-1 Starting the LogDNA Dashboard

There are two ways to start the LogDNA Dashboard. The first one is directly from you cluster dashboard. After logging into IBM Cloud, from the dashboard, select OpenShift and then Clusters.



Next, click on your cluster name where you deployed the zero to cloud native tutorial application.

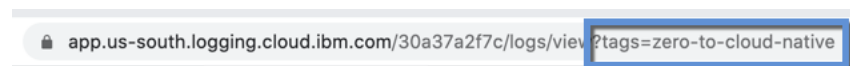
OpenShift clusters

Resource group: Filter...		Location: Filter...		Filter table		Create cluster	+
Name	State	Location	Worker Count	Created	Version		
zero-to-cloud-native	 Normal	Dallas	9	9/10/2020, 5:33 PM	 4.4.20_1518	⋮	
Items per page: 50		1-1 of 1 item			1	1 of 1 page	⏪ ⏩

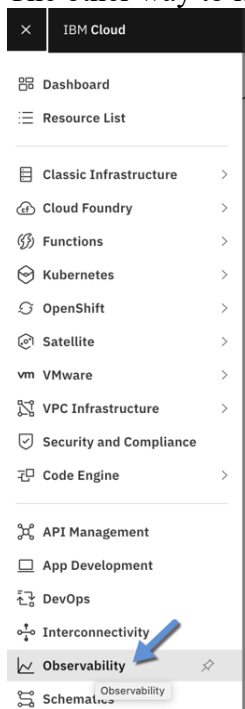
From this overview page of your cluster, click on Launch next to logging.

Summary	
Cluster ID	btd9n56d0rp32dnacs6g
Master status	Ready
Version	4.4.20_1518
Infrastructure	VPC Gen2
Zones	us-south-1, us-south-2, us-south-3
Created	9/10/2020, 5:33 PM
Ingress subdomain	zero-to-cloud-native-865ffcc72a16e7f393d2878612ad8f9c-0000.us-south.containers.appdomain.cloud
Ingress status	Healthy
Ingress message	All Ingress components are healthy
Resource group	zero-to-cloud-native
Logging	Launch Disconnect
Monitoring	Launch Disconnect
Key management service	Enabled Update
Image pull secrets	Enabled

When you start logDNA in this manner, it will take you automatically in context to the log messages just for that cluster. If you look at the URL, you will notice that it ends in tags=<cluster_name>, so you will only see relevant log messages for this cluster.



The other way to launch LogDNA is through the Observability from the IBM Cloud menu.



On the next screen, click on Logging and then View LogDNA.

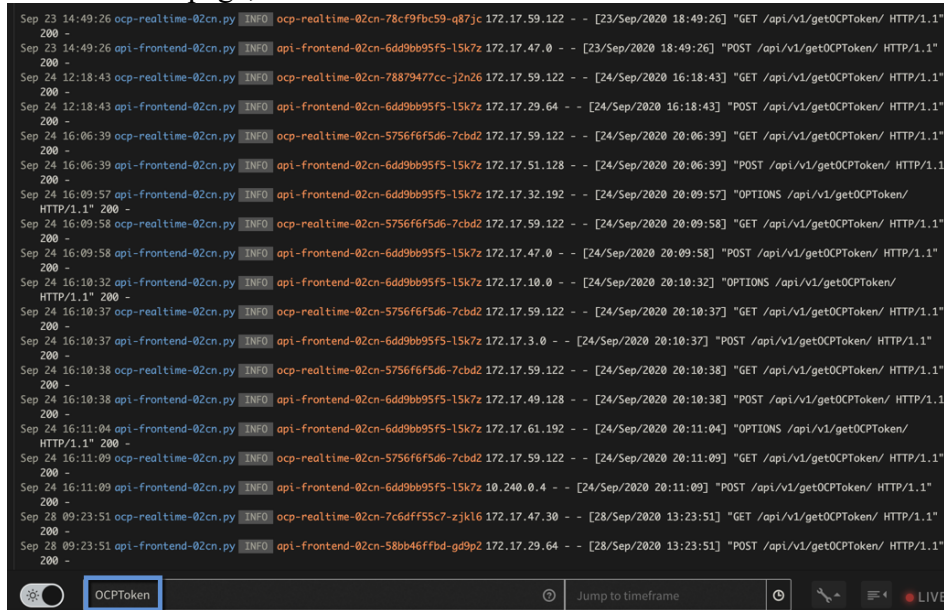
A screenshot of the IBM Cloud 'Logging' page. On the left, a dark sidebar shows 'Observability' selected, with 'Logging' and 'Monitoring' also visible. The main content area is titled 'Logging' and features a table of log resources. A blue arrow points to the 'View LogDNA' link in the 'View dashboard' column of the table. The table has columns for Name, Status, Resource Group, Region, Sources, Plan, and View dashboard. The first row shows 'IBM Log Analysis with LogDNA-zero...' as 'Active' in the 'zero-to-cloud-native' resource group in the 'Dallas' region, with a '7 day Log Search' plan. The 'View dashboard' column contains a 'View LogDNA' link. At the top right, there are buttons for 'Configure platform logs' and 'Create instance'. At the bottom, there are pagination controls showing '1 of 1 page'.

This time, you will see all the log messages for all the data sources you have enabled. Note, that even if you go through the cluster overview to get to LogDNA, you can still see all the log messages that LogDNA is capturing. You are just starting with a view of log messages only for that cluster.

1-2 Searching Log Messages

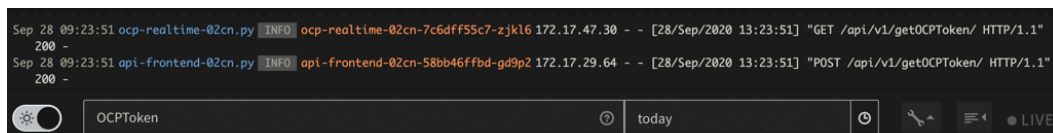
One of the great features of LogDNA is being able to perform a natural language search on all of your log messages. The following example will assume you have run the get OCP Token API from the tutorial application that you deployed and tested in the previous section.

I will search for OCPToken to see when this API has been called. In the search criteria at the bottom of the page, I'll enter **OCPToken**.



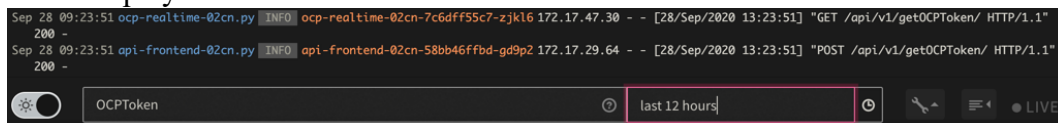
```
Sep 23 14:49:26 ocp-realtime-02cn.py INFO ocp-realtime-02cn-78cf9b59-q87jc 172.17.59.122 - - [23/Sep/2020 18:49:26] "GET /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 23 14:49:26 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z 172.17.47.0 - - [23/Sep/2020 18:49:26] "POST /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 12:18:43 ocp-realtime-02cn.py INFO ocp-realtime-02cn-78879477cc-j2n26 172.17.59.122 - - [24/Sep/2020 16:18:43] "GET /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 12:18:43 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z 172.17.29.64 - - [24/Sep/2020 16:18:43] "POST /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:06:39 ocp-realtime-02cn.py INFO ocp-realtime-02cn-5756f6f5d6-7cbd2 172.17.59.122 - - [24/Sep/2020 20:06:39] "GET /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:06:39 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z 172.17.51.128 - - [24/Sep/2020 20:06:39] "POST /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:09:57 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z 172.17.32.192 - - [24/Sep/2020 20:09:57] "OPTIONS /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:09:58 ocp-realtime-02cn.py INFO ocp-realtime-02cn-5756f6f5d6-7cbd2 172.17.59.122 - - [24/Sep/2020 20:09:58] "GET /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:09:58 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z 172.17.47.0 - - [24/Sep/2020 20:09:58] "POST /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:10:32 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z 172.17.10.0 - - [24/Sep/2020 20:10:32] "OPTIONS /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:10:37 ocp-realtime-02cn.py INFO ocp-realtime-02cn-5756f6f5d6-7cbd2 172.17.59.122 - - [24/Sep/2020 20:10:37] "GET /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:10:37 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z 172.17.3.0 - - [24/Sep/2020 20:10:37] "POST /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:10:38 ocp-realtime-02cn.py INFO ocp-realtime-02cn-5756f6f5d6-7cbd2 172.17.59.122 - - [24/Sep/2020 20:10:38] "GET /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:10:38 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z 172.17.49.128 - - [24/Sep/2020 20:10:38] "POST /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:11:04 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z 172.17.61.192 - - [24/Sep/2020 20:11:04] "OPTIONS /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:11:09 ocp-realtime-02cn.py INFO ocp-realtime-02cn-5756f6f5d6-7cbd2 172.17.59.122 - - [24/Sep/2020 20:11:09] "GET /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 24 16:11:09 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z 10.240.0.4 - - [24/Sep/2020 20:11:09] "POST /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 28 09:23:51 ocp-realtime-02cn.py INFO ocp-realtime-02cn-7c6dff55c7-zjk16 172.17.47.30 - - [28/Sep/2020 13:23:51] "GET /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 28 09:23:51 api-frontend-02cn.py INFO api-frontend-02cn-58bb46ffbd-gd9p2 172.17.29.64 - - [28/Sep/2020 13:23:51] "POST /api/v1/getOCPToken/ HTTP/1.1" 200 -
```

You can see all the log messages that contain OCPToken. I can then limit the results by specifying a time frame. To view all log messages containing OCPToken from today, I simply enter today at the timeframe.



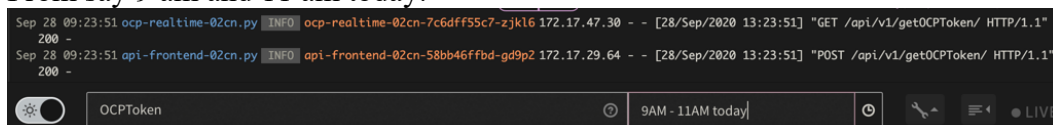
```
Sep 28 09:23:51 ocp-realtime-02cn.py INFO ocp-realtime-02cn-7c6dff55c7-zjk16 172.17.47.30 - - [28/Sep/2020 13:23:51] "GET /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 28 09:23:51 api-frontend-02cn.py INFO api-frontend-02cn-58bb46ffbd-gd9p2 172.17.29.64 - - [28/Sep/2020 13:23:51] "POST /api/v1/getOCPToken/ HTTP/1.1" 200 -
```

You can play around the time frame and enter other criteria like search the last 12 hours.



```
Sep 28 09:23:51 ocp-realtime-02cn.py INFO ocp-realtime-02cn-7c6dff55c7-zjk16 172.17.47.30 - - [28/Sep/2020 13:23:51] "GET /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 28 09:23:51 api-frontend-02cn.py INFO api-frontend-02cn-58bb46ffbd-gd9p2 172.17.29.64 - - [28/Sep/2020 13:23:51] "POST /api/v1/getOCPToken/ HTTP/1.1" 200 -
```

From say 9 am and 11 am today.

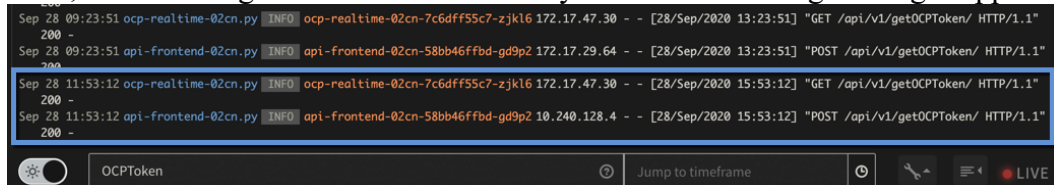


```
Sep 28 09:23:51 ocp-realtime-02cn.py INFO ocp-realtime-02cn-7c6dff55c7-zjk16 172.17.47.30 - - [28/Sep/2020 13:23:51] "GET /api/v1/getOCPToken/ HTTP/1.1" 200 -
Sep 28 09:23:51 api-frontend-02cn.py INFO api-frontend-02cn-58bb46ffbd-gd9p2 172.17.29.64 - - [28/Sep/2020 13:23:51] "POST /api/v1/getOCPToken/ HTTP/1.1" 200 -
```

To turn on a live tail to view log messages as they arrive, click on the Live button.

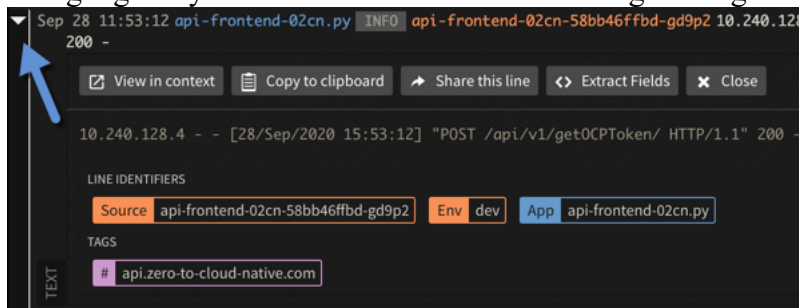


Now, execute the get OCP Token API and you will see new log messages appear.



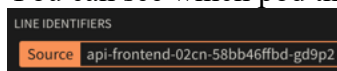
This just gives you an idea of how powerful and useful this search is. I probably use this every day to narrow down log messages in various applications I have running.

On the left-hand side of each message, you will see a triangle to example. Clicking on the triangle gives you more information about the log message.



This view shows us some very important information.

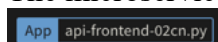
You can see which pod the log message came from in the Source section:



Which environment it came from:



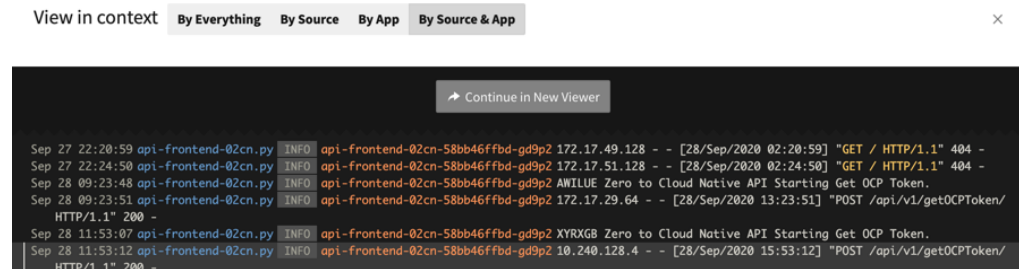
The microservice or App:



And any special tags that we add to the log message. In this case, our python code is adding `api.<domain_name>` to each log entry so we can easily filter only on those log messages relevant to the tutorial application.

Having this information added to each log messages is very beneficial to track down exactly where a log message came from especially if there is an error.

At the top of the message, there several helpful options as well. Clicking on **View in Context** shows you other log messages that occurred around the same time as the original message you viewed. Be default, the context is set to source and app, you can expand this by just app, just source, or everything if you wish.



Click **copy to clipboard** copies the log message itself to the clipboard. Note, that this only copies the log next and not the metadata.

```
10.240.128.4 - - [28/Sep/2020 15:53:12] "[37mPOST /api/v1/getOCPToken/ HTTP/1.1[0m" 200 -
```

Share this line item, allows you to share a URL that another user can look at to view the log messages starting with the specific message you have selected. Note that whomever opens the link must have permissions to view log messages in this logDNA instance. The nice thing about sharing the link however, is the that recipient will view the log message in context. The next part of this tutorial series 10A will go through how you can share log messages for those who do not have access to the LogDNA instance itself.

1-2-1 Configure Python and LogDNA

In Visual Studio code, or your favorite IDE, navigate to the app/api-frontend-02cn.py file in the api-frontend-02cn repository. In the beginning of the file, you will see where we set the logging parameters that show up as metadata in each log entry.

```
19     'handlers': {
20         'logdna': {
21             'level': logging.DEBUG,
22             'class': 'logging.handlers.LogDNAHandler',
23             'key': os.environ.get('LOGDNA_APIKEY'),
24             'options': {
25                 'app': 'api-frontend-02cn.py',
26                 'tags': os.environ.get('SERVERNAME'),
27                 'env': os.environ.get('ENVIRONMENT'),
28                 'url': os.environ.get('LOGDNA_LOGHOST'),
29                 'index_meta': True,
30             },
31         },
32     },
```

A couple of things to note here:

level – by default, the tutorial application is logging very granularly so we can view all log messages using the DEBUG level. For a production level system, you might consider changing this to INFO or ERROR.

key – this is the logDNA key we previously retrieved in part 7 section 5 of this tutorial series.

app – we are setting our microservice name as the app.

tags – we are using the server name that we set in our Kubernetes secrets as a special tag. In this case, the tag is api.zero-to-cloud-native.

env – environment this application represents. In our case dev, which we are setting in our Kubernetes secrets.

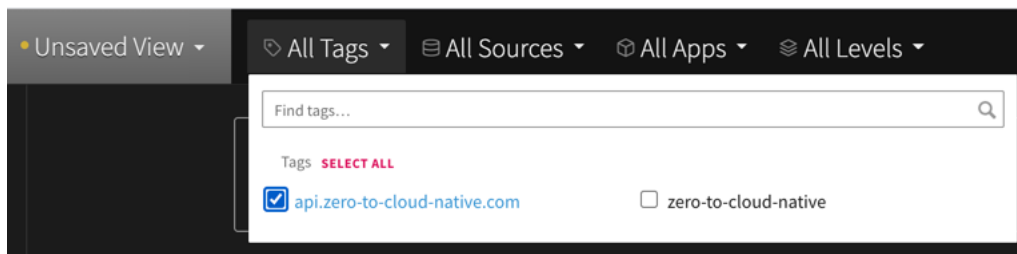
url – this is the LogDNA ingestion URL we retrieved with our key in section 7 part 5 and set in our Kubernetes secrets.

1 – 3 Filtering Log Messages

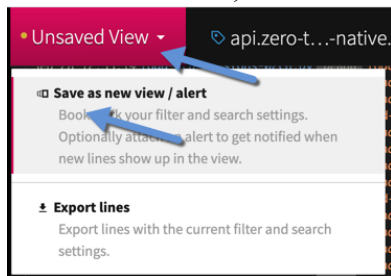
By default, LogDNA captures log messages from every component of your Kubernetes cluster which can be overwhelming if you don't narrow down your search criteria. LogDNA provides several tools to filter the log messages so you can focus on what is important to you.

1-3-1 Filter on an Application

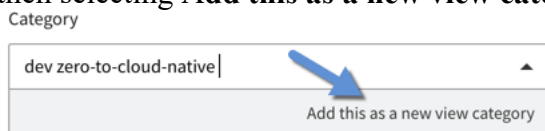
One very helpful view is to only view those log messages for a specific application. In this example, we will filter only on those log messages for our tutorial application. To do so, click on All Tags, and select the tag for our tutorial application – api.zero-to-cloud-native.com



After clicking this, our view will change to only show those log messages for our tutorial application. This will be a very common and helpful view that that we will want to view often. To save this view, click on **Unsaved View** and select **Save as new view/alert**.



The next thing we will want to do is create a category or grouping of like views. We will create a **dev zero-to-cloud-native** category by typing in **dev zero-to-cloud-native** under Category and then selecting **Add this as a new view category**.



Next, enter a name of the View, in this case set the name to **All**, then click **Save View**.

Create new view

Tags: `api.zero-to-cloud-native.com`

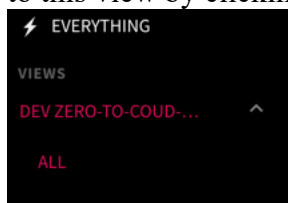
Name: **ALL**

Category: DEV ZERO-TO-CLOUD-NATIVE

Alert: Type to find or add alerts

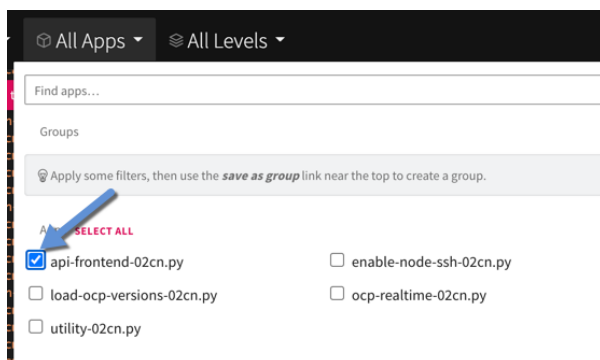
Save View

Now, on the left hand side of the View tab, you will see a new category **DEV ZERO-TO-CLOUD-NATIVE** with one view called **ALL**, that we just created. Now we can easily go back to this view by clicking on the name of the view.

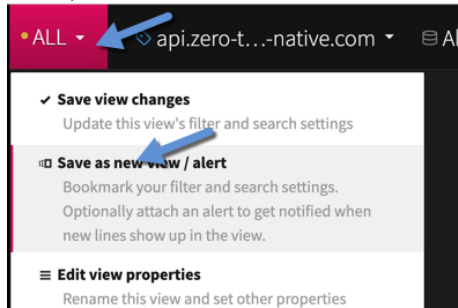


While viewing log messages at the application level view is very useful, there are still a lot of log messages across all microservices that make up the application. A best practice is to have a separate view for each microservice.

While on the ALL view of the DEV ZERO-TO-CLOUD-NATIVE category, we can now filter on each microservice. Starting with the `api-frontend-02cn` microservice, click on **All Apps**, and then select **api-frontend-02cn.py**



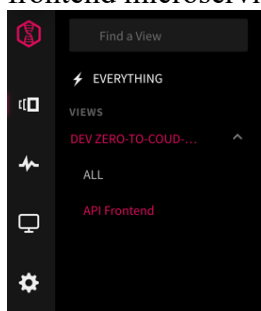
The view will now update to only show those log messages from this microservice. To save this view, click on ALL and then **Save as new view / alert**.



Change the name of **API Frontend**.

A screenshot of the 'Create new view' form. It shows a 'Name' field with 'API Frontend' entered, a 'Category' dropdown set to 'DEV ZERO-TO-CLOUD-NATIVE', and an 'Alert' dropdown. A 'Save View' button is at the bottom.

Now you will see a second view that we can quickly access to view log messages from the api frontend microservice.

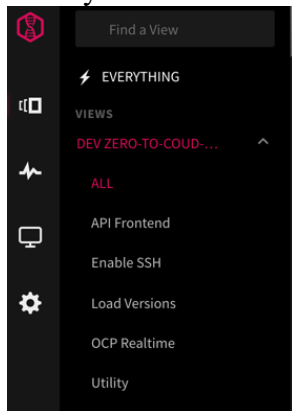


Repeat the same steps for the remaining microservices:

- enable-node-ssh-02cn
- load-ocp-version-02cn
- ocp-realtime-02cn
- utility-02cn

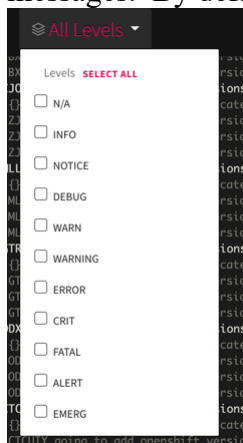
* note: we are not logging the web frontend as it is a simple web server but this is something you can do as extra credit if you like.

After you add new views for each microservice, you should see the following views.

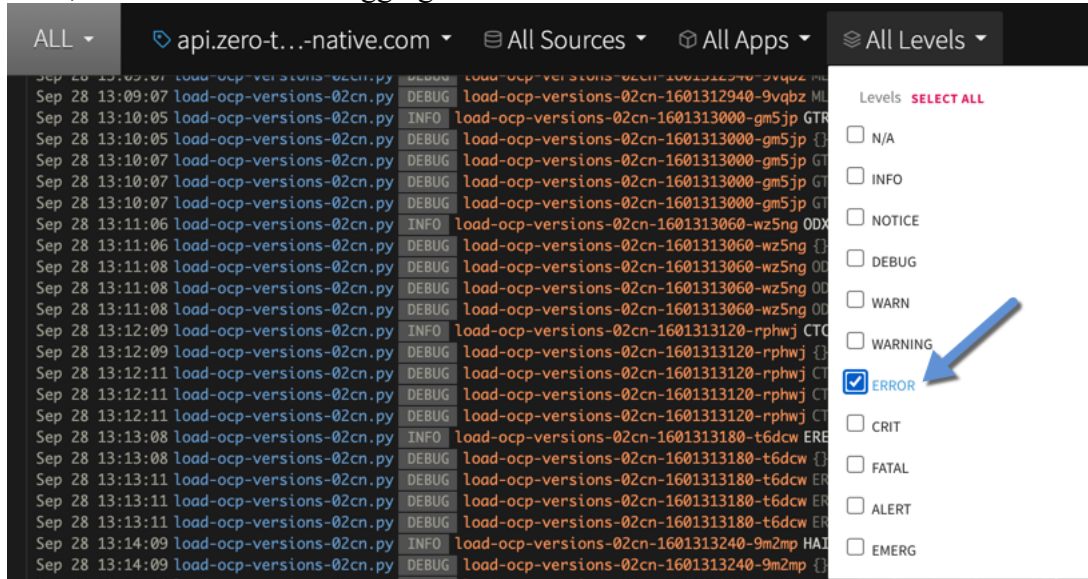


Now, we can very easily view log messages specific to any of our microservices which is very useful for developers of those microservices.

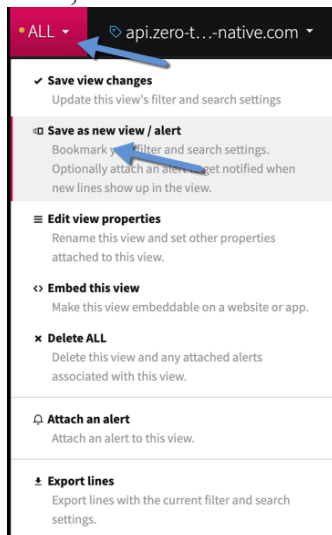
The last thing to point out on the Views tab is being able to filter on different levels of log messages. By default, all levels are selected.



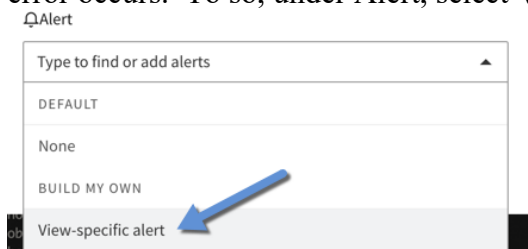
One view that is useful, is to see all error messages for the application. While on the 'ALL' view, select **Error** as the logging level.



Now, save this view as new view, by selecting Save a new view/alert.



Enter, **ERRORS – ALL** as the name of the view. For errors, we want to be notified when an error occurs. To so, under Alert, select **View-specific alert**.



When you create an alert, you have four choices out of the box, email, slack, pager duty, and a webhook.

Levels: ERROR
Tags: api.zero-to-cloud-native.com

Name
ERRORS - ALL

Category
DEV ZERO-TO-CLOUD-NATIVE

Alert
View-specific alert

+ [Channel Name]

Slack Email Webhook PagerDuty

Save View

For this tutorial, we will be creating simple email alert. To do so, I'll click on **email** and then enter my email address. You can also see options of aggregating alerts and log messages together to limit the number of emails you receive.

Email [Test] [Delete Channel]

Type: Presence (selected) | Absence

When: 1 or more matches appear within 30 seconds

Send an alert: ☒ At the end of 30 seconds
☐ Immediately after 1 line

Recipients: kevincollins@us.ibm.com

Timezone: Preferred timezone (optional)

I will just keep the default settings and click on **Save View**. Now, when there is an error coming from the tutorial application running on my cluster, I will be alerted with the log message.

1 – 4 Creating Logging Graphs and Boards

Another of the great features of LogDNA is the ability to create a graphical representation of your log messages. For example, we can create a graph that shows error messages over a period of time.

A board is a named collection of graphs. Boards can be found in the Log-viewer sidebar on the left. Changes made to a board are automatically saved. Similar to views, all of your users have access to all boards in your account.

You can view more about boards from the LogDNA website.

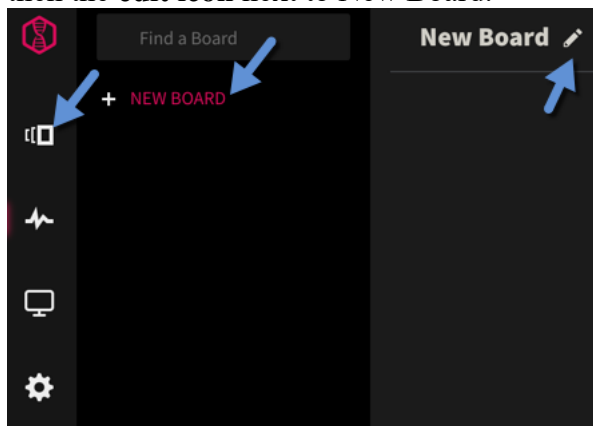
<https://docs.logdna.com/docs/graphs>

In this tutorial, we are going to create a couple of different views.

Total API Calls for the application itself.

Number of calls per microservice.

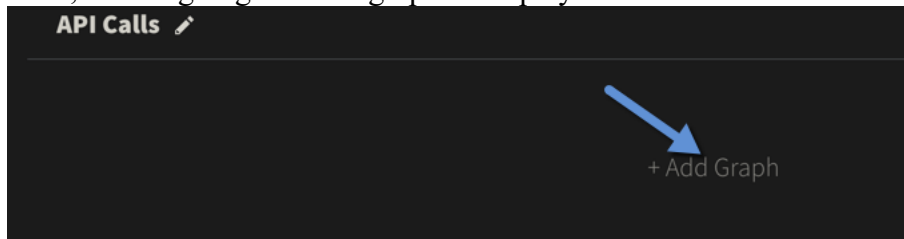
To start with, click on the **Boards icon** on the left hand navigation pane, click **New Board** and then the **edit icon** next to New Board.



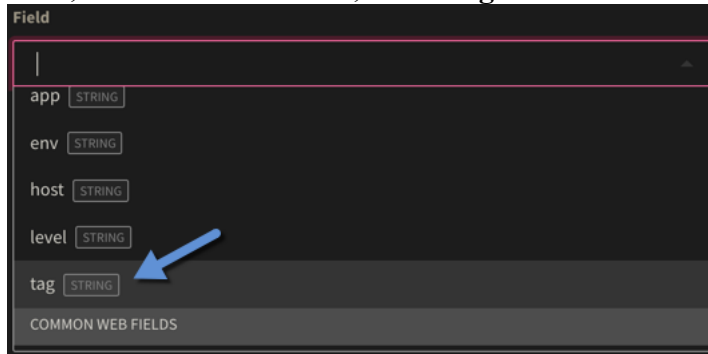
On the next screen, enter **API Calls** as the name and create a new category called **DEV Zero To Cloud Native** as the category name and then 'Add this as a new board category', and then click **Save**.

A screenshot of the 'Edit Board' modal form. The modal has a title bar with 'Edit Board' and a close button. It contains two input fields: 'Name' and 'Category'. The 'Name' field has 'API Calls' entered. The 'Category' field has 'DEV ZERO TO Cloud Native' entered. Below the 'Category' field is a button labeled 'Add this as a new board category'. At the bottom of the modal are two buttons: 'Cancel' and 'Save'. Blue arrows point to the 'Name' field, the 'Category' field, the 'Add this as a new board category' button, and the 'Save' button.

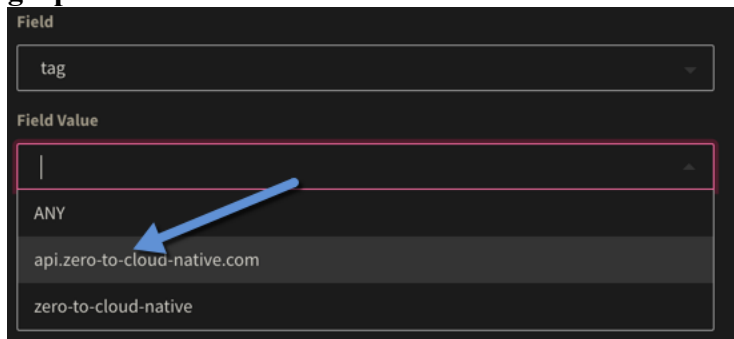
Next, we are going to add a graph to display the number of API calls. Click on **Add Graph**.



Next, in the Field selection, **select tag**.



Next, select the tag for the tutorial application, **api.zero-to-cloud-native.com** and then click **add graph**.

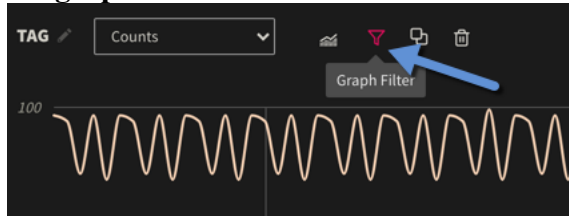


At this point, we are showing a graph showing the number for all log messages for the application.

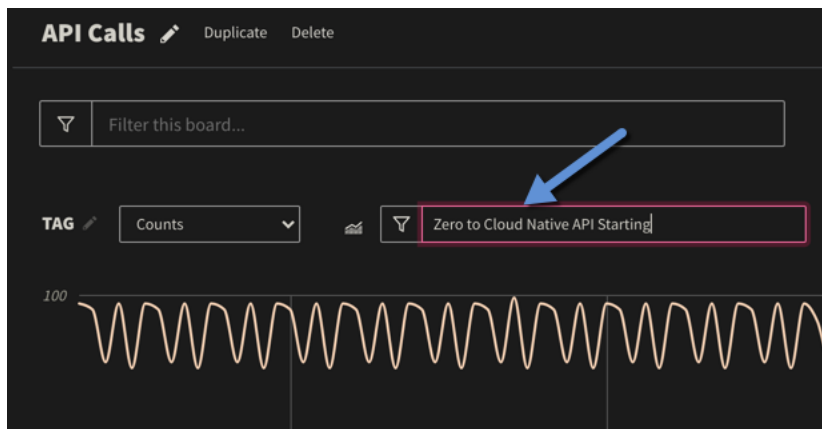
Looking into python code under **app/api-frontend-02cn.py** file under the api-frontend-02cn repository, scroll down to one of the APIs such as **EnableSSH**. You will notice that each time an API is called, we are creating a log message that starts with '**Zero to Cloud Native API Starting**'

```
class EnableSSH(Resource):  
    def post(self):  
        try:  
            input_json_data = request.get_json()  
            reqid=getRequestId()  
            app.logger.info("{} Zero to Cloud Native API Starting end")
```

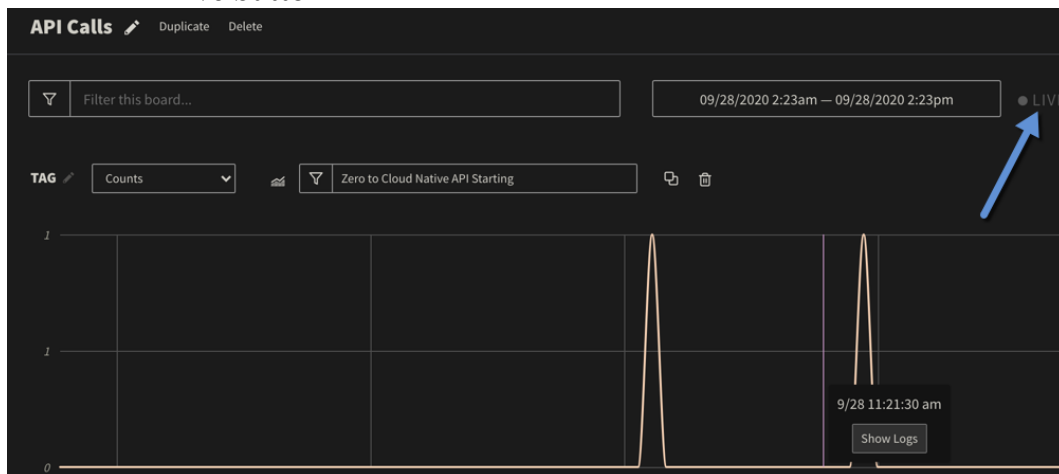
To change the graph to count the number of api calls, create an additional filter by clicking on the **graph filter icon**.



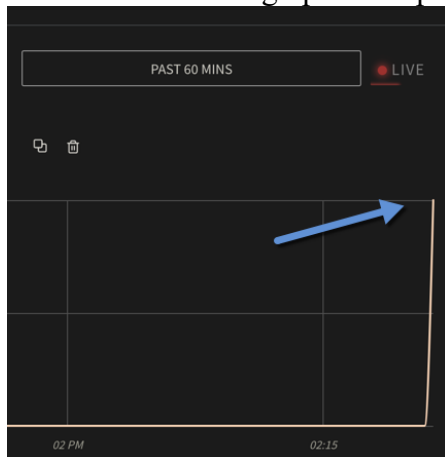
On the next screen, enter '**Zero to Cloud Native API Starting**' as the search string and hit enter.



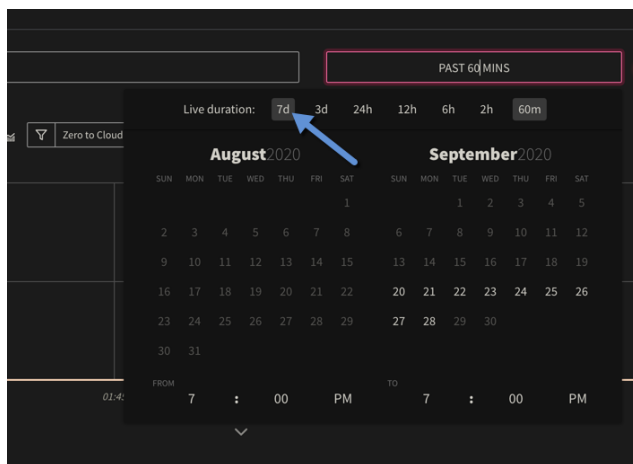
Next, to test the graph, execute one of the APIs such as get OCP Token. After you run the API, click on the **Live** button.



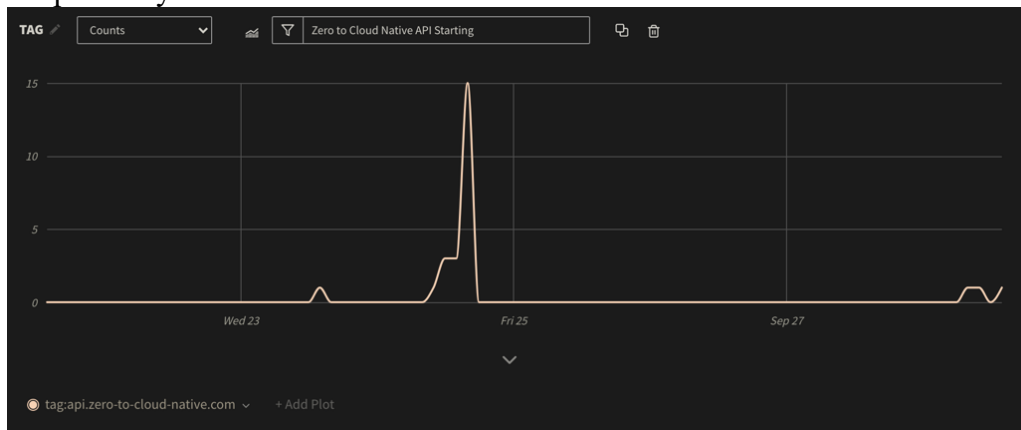
You will notice the graph was updated to show that 1 API call was just made.



Depending on how many APIs you have run in your tutorial application, you will see different results. To view the graph over several days as an example, click on the 60 MINS and change this to **7 days**.

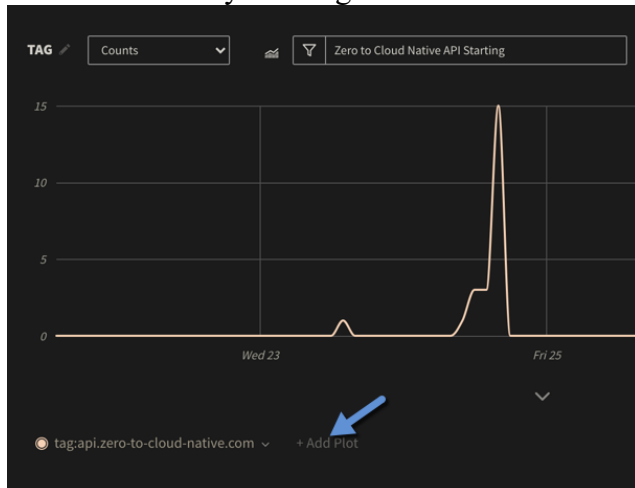


You can see my example, I called one of the APIs a number of times last Friday and now just a couple today.



Now I can see how the number of APIs are trending over time. While this is interesting, viewing just the number of times doesn't give us a view on how the application is performing. What would be more insightful is adding the number of errors to this view.

We can do that by clicking on **Add Plot**.



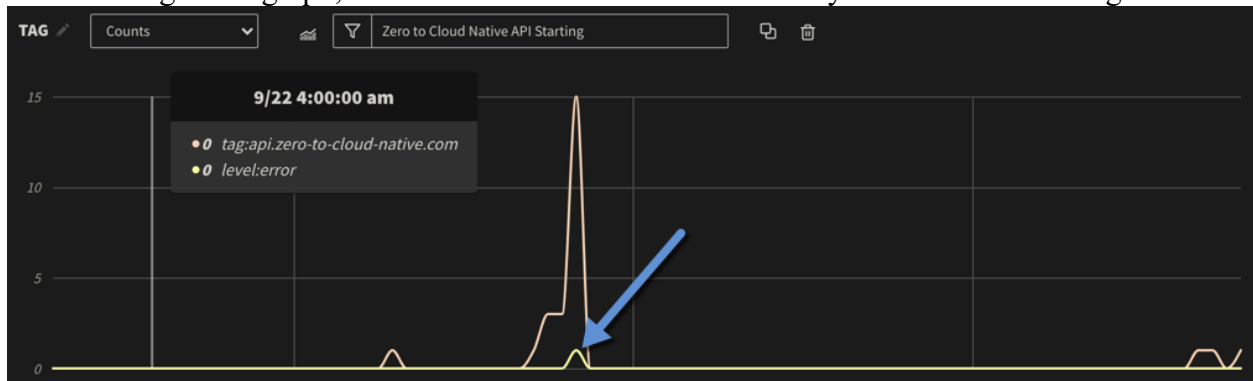
On the next screen, select **level**.

The screenshot shows a configuration screen for the new plot. The 'level' field is highlighted with a blue arrow. The 'level' field is currently set to 'STRING'.

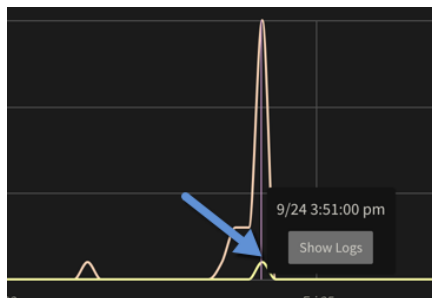
Select error and then **save**.

The screenshot shows the configuration screen with the 'level' dropdown menu open. The 'error' option is selected. The 'Save' button is visible.

Now looking at the graph, I can see that I had an error last Friday when there was a higher load.



Clicking on the error graph where there was an error message, I can then click on **Show Logs** to see what caused the error.



This takes me in-context to where log messages where the error occurred.

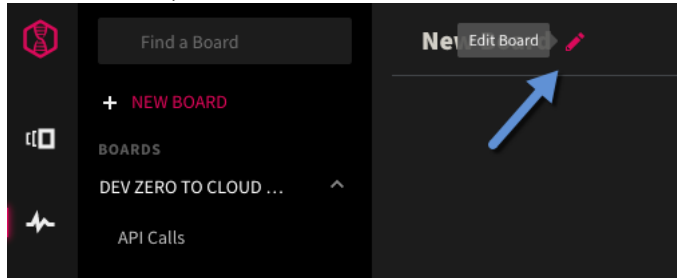
```
Sep 24 16:09:57 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z GKZQI Zero to Cloud Native API Starting Get OCP Token.
Sep 24 16:09:58 api-frontend-02cn.py ERROR api-frontend-02cn-6dd9bb95f5-15k7z GKZQI Zero to Cloud Native API Starting Get OCP token 'token'
Sep 24 16:10:32 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z ARQRLY Zero to Cloud Native API Starting Get OCP Token.
Sep 24 16:10:36 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z SWBPRY Zero to Cloud Native API Starting Get OCP Token.
Sep 24 16:11:05 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z YZZWQN Zero to Cloud Native API Starting Get OCP Token.
Sep 24 16:11:21 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z TEAXMB Zero to Cloud Native API Starting enable SSH.
Sep 24 16:12:17 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z QJWOOI Zero to Cloud Native API Starting Get OCP Versions.
Sep 24 16:12:22 api-frontend-02cn.py INFO api-frontend-02cn-6dd9bb95f5-15k7z SRPPLF Zero to Cloud Native API Starting Get OCP Versions.
Sep 24 16:19:26 utility-02cn.py INFO utility-02cn-64876fd5b5-nxlvz 172.17.32.239 - - [24/Sep/2020 20:19:26] "GET /api/v1/getiamtoken/ HTTP/1.1" 200 -
Starting Zero To Cloud Native Utility - request to get IAM Token.
Successfully got iam token
<< >> Thursday, Sep 24th 2020, 4:22pm
Sep 24 16:26:01 api-frontend-02cn.py DEBUG api-frontend-02cn-58bb46ffbd-gd9p2 Starting zero to cloud native api frontend
Sep 24 16:26:01 utility-02cn.py INFO utility-02cn-64876fd5b5-8r164 172.17.47.30 - - [24/Sep/2020 20:26:01] "GET /api/v1/getiamtoken/ HTTP/1.1" 200 -
```

Again, you will have different log messages. As extra credit, you can modify the code and introduce errors if you would like to see similar log messages.

Next, let's create one more board showing the number of times the microservices are called. Because the load OCP versions microservice is a cronjob and runs every 5 minutes tracking the number of times this microservice is called is not that interesting or insightful. We will create a new board that will count the number of times the following microservices are invoked:

- Enable SSH
- Utility
- OpenShift Realtime
- Frontend API

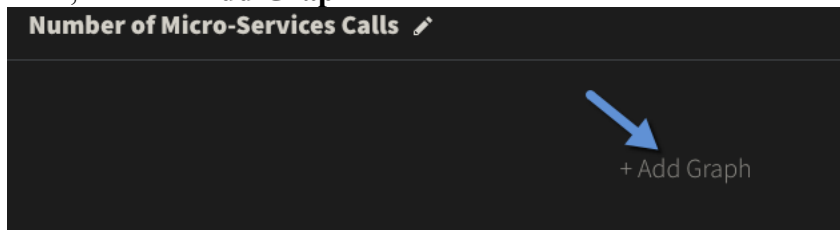
To start with, click on **New Board** and then **Edit Board** to change the name.



Enter **Number of Micro-Services Calls** as the name and select 'DEV Zero to Cloud Native' as the Category, then click **Save**.

A screenshot of the 'Edit Board' dialog box. It has a title bar 'Edit Board' with a close button. Below the title bar, there is a 'Name' label and a text input field containing 'Number of Micro-Services Calls'. Below that is a 'Category' label and a dropdown menu showing 'DEV Zero to Cloud Native'. At the bottom right, there are two buttons: 'Cancel' and 'Save'. Blue arrows point to the 'Name' field, the 'Category' dropdown, and the 'Save' button.

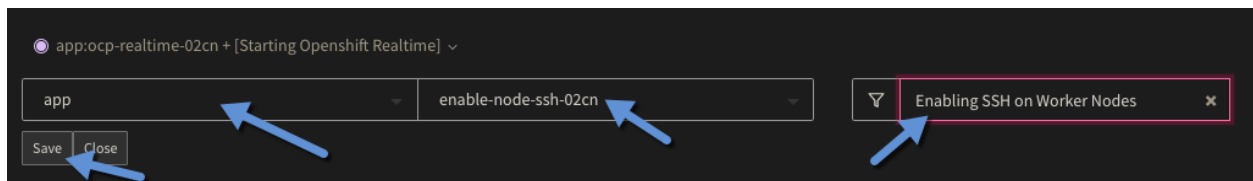
Next, click on **Add Graph**.



On the next screen, enter **app** as the field, select **ocp-realtime-02cn** as the field value. Click on **Advanced Filtering** and enter 'Starting Openshift Realtime' to limit the log messages so we can get a count of the number of times this microservice is called. After entering all of these settings, click on **Add Graph**.



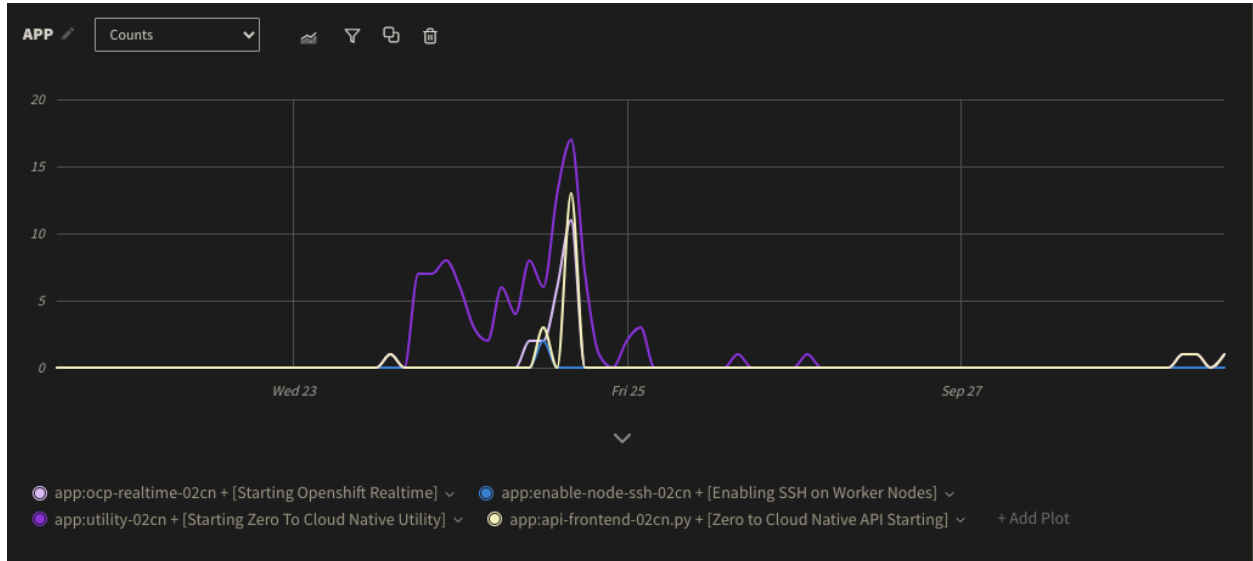
Next, we need to enter additional plots of the remaining microservices. Starting with Enable SSH on Worker Nodes, click on **Add Plot**. For enable node ssh enter:
Field: **app**
Field Value: **enable-node-ssh-02cn**
Advanced Filtering: **Enabling SSH on Worker Nodes**
Click **Save**.



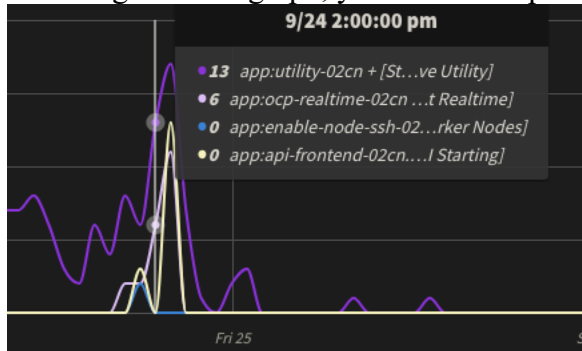
Repeat the same process for Utility Microservice. Click on **Add Plot**. For utility enter:
Field: **app**
Field Value: **utility-02cn**
Advanced Filtering: **Starting Zero To Cloud Native Utility**
Click **Save**.

Repeat the same process of the API Frontend Microservice. Click on **Add Plot**. For api frontend:
Field: **app**
Field Value: **api-frontend-02cn**
Advanced Filtering: **Zero to Cloud Native API Starting**
Click **Save**.

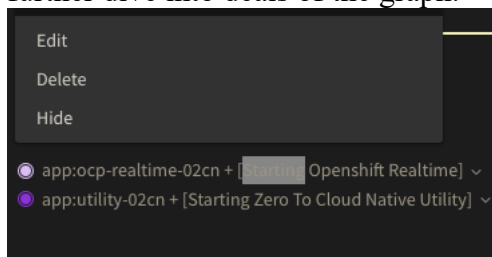
Depending on how many APIs you have made, you will see a graph showing how many times each of the four APIs were called.



Hovering over the graph, you can view specifics of how many times each API was called.



Clicking on one of the plots you will also have the option to hide the plot from the graph to further dive into details of the graph.

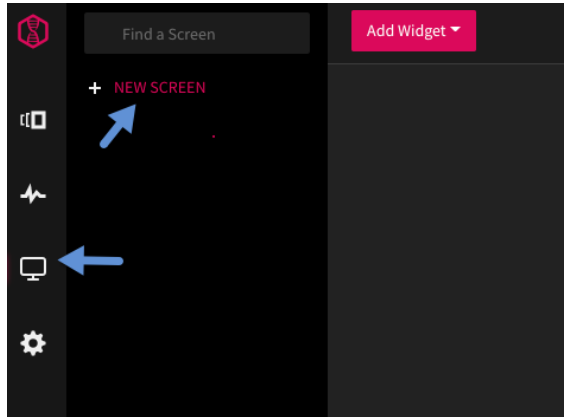


1 – 5 Logging Metrics via Screens

The last visibility tool we will go through with LogDNA is Screens. LogDNA Screens let you create detailed customized dashboards using your log data. Screens provide several configurable widgets, which display aggregate metrics about logs over a given period of time. You can use Screens to visualize overall log volume, key performance metrics (KPIs), and more. You can view more about screens on the LogDNA site:

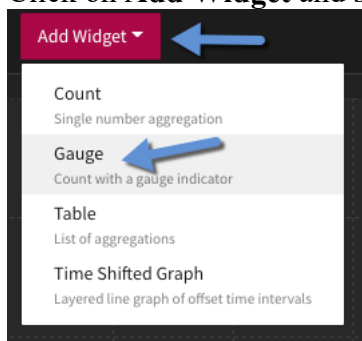
<https://docs.logdna.com/docs/screens>

To create a new screen, click on the **Screens icon**, and then **New Screen**.



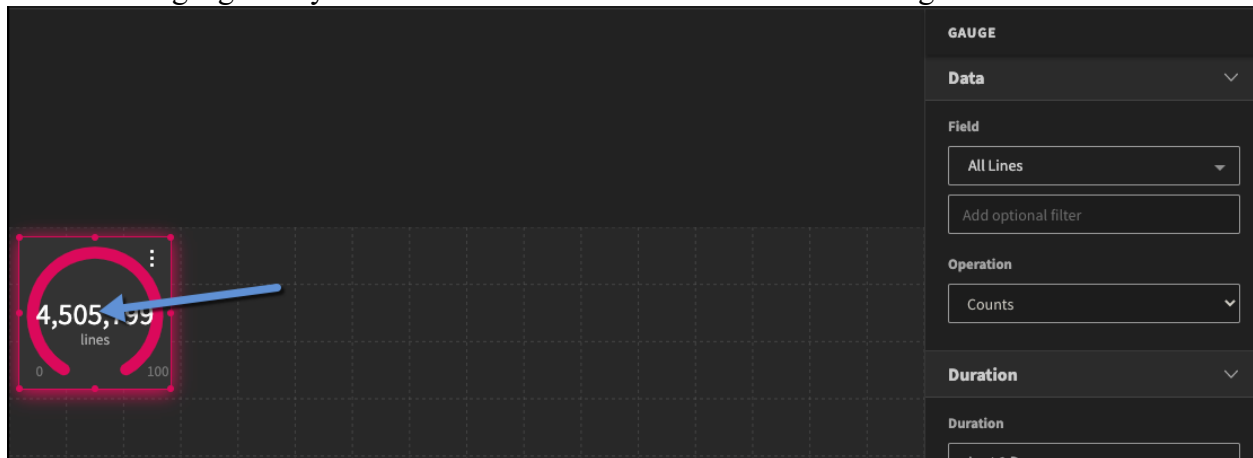
The first screen we will build is a gauge showing how many API calls we have processed.

Click on **Add Widget** and select **count**.



The gauge will now show the total number of logging lines, which is not that helpful. To display the number of API calls, we will need to customize the view.

Click on the gauge and you will be able to customize the view on the right hand side.



As we did when we created a board showing the total number of API calls, enter the following values:

Field: **tag**

Field Value: **api.zero-to-cloud-native.com**

Advanced Filtering: **Zero to Cloud Native API Starting**

Duration: **Last 1 Week**

Note: your field value should refer to the domain you created.

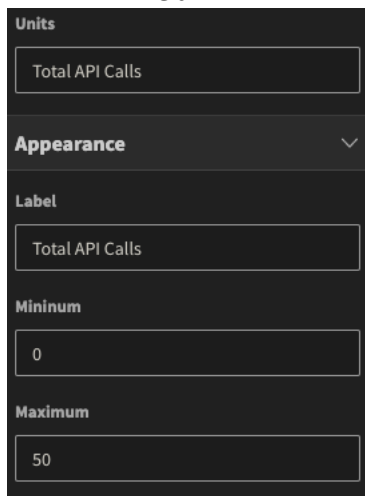
The image shows the configuration panel for a gauge chart. The panel has a dark background and a title 'GAUGE'. It contains several sections: 'Data' (a dropdown menu), 'Field' (a dropdown menu showing 'tag'), 'Field Value' (a dropdown menu showing 'api.zero-to-cloud-native.com' and a text input field containing 'Zero to Cloud Native API Starting'), 'Operation' (a dropdown menu showing 'Counts'), and 'Duration' (a dropdown menu showing 'Last 1 Week'). There is also a button 'Hide Advanced Filtering'.

Next, we will customize how the gauge looks. Scrolling down enter the following:

Units: **Total API Calls**

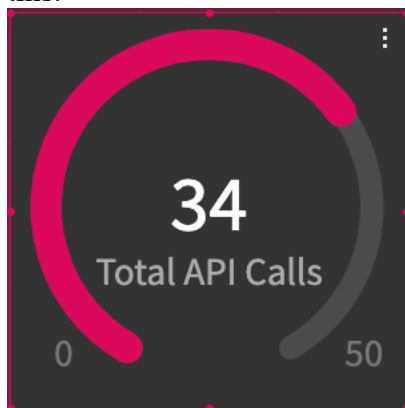
Label: **Total API Calls**

Maximum: **50**

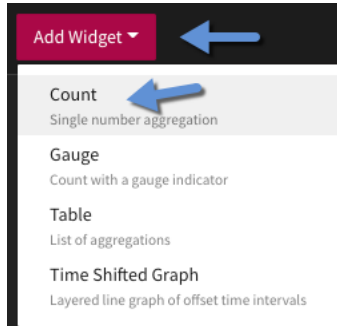


The image shows a configuration panel for a gauge widget. It has a dark background with white text. The sections are: 'Units' with a text input field containing 'Total API Calls'; 'Appearance' with a dropdown arrow; 'Label' with a text input field containing 'Total API Calls'; 'Minimum' with a text input field containing '0'; and 'Maximum' with a text input field containing '50'.

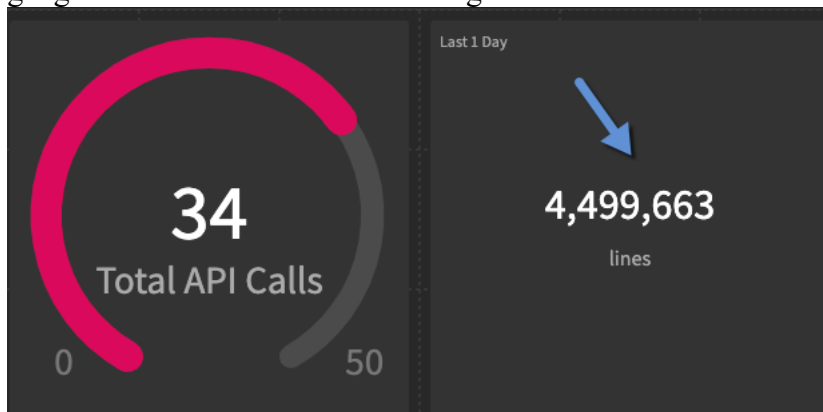
Depending on how many API calls you have made in the last week, your gauge should look like this.



Next, let's add a simple count of the number of API calls that failed. Click on **Add Widget** and then **Count**.



Just like we did before, we will need to customize the count view. By default, the count shows the total number of log lines. Before we customize the view, let's move the count of errors next to the total number of api calls. **Click and drag** the count widget you just created next to the gauge. Then click on the count widget to customize it.



On the right-hand side of the screen, enter the following information:

Field: **level**

Field value: **error**

Advanced Filtering: **tag:api.zero-to-cloud-native.com AND Problem**

Duration: **Last 1 Week**

The configuration panel for a dashboard widget. It includes a 'Field' dropdown set to 'level', a 'Field Value' dropdown set to 'error', an 'Advanced Filtering' text input containing 'tag:api.zero-to-cloud-native.com AND P', a 'Hide Advanced Filtering' button, an 'Operation' dropdown set to 'Counts', a 'Duration' dropdown set to 'Last 1 Week', and a 'Duration' label.

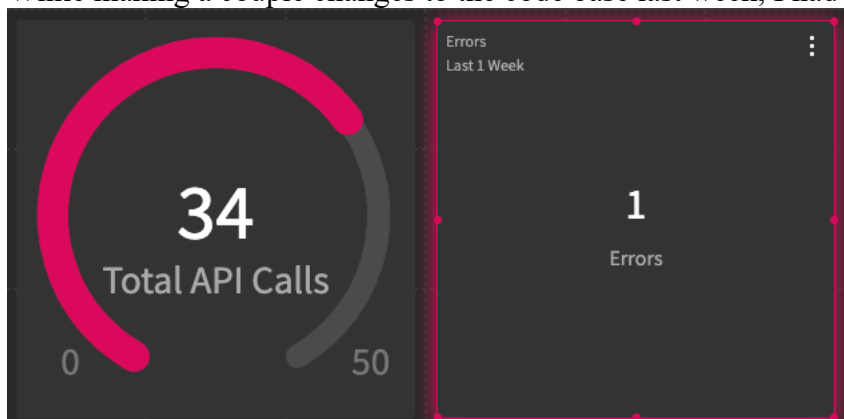
Lastly, we will customize the view, enter the following values:

Units: **Errors**

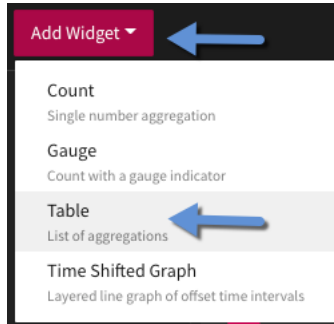
Label: **Errors**

The customization panel for a dashboard widget. It includes a 'Units' text input set to 'Errors', an 'Appearance' dropdown, a 'Label' text input set to 'Errors', and a 'Label' label.

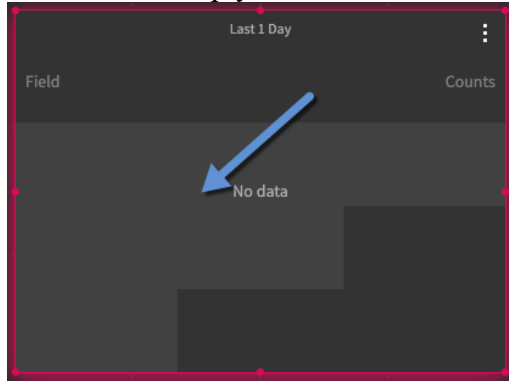
Hopefully, your error will show zero meaning your application hasn't encountered any problems. While making a couple changes to the code base last week, I had a problem and show 1 error.



Next, we will create another view, this time a table showing the number of calls to each microservice. Click on **Add Widget** and choose **Table**.



Click on the empty table to customize it:



On the right-hand side, enter the following information:

Group By: **app**

Field: **All Lines**

Advanced Filtering:

(app:ocp-realtime AND Starting Openshift Realtime) OR

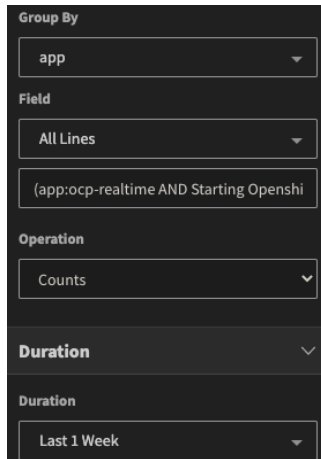
(app:utility-02cn AND Starting Zero To Cloud Native Utility) OR

(app:api-frontend-02cn AND Zero to Cloud Native API Starting) OR

(app:enable-node-SSH-02cn AND Enabling SSH on Worker) OR

(app:load-OCP-versions-02cn AND Starting to load)

Duration: Last 1 Week




Next, we need to customer the view:
Number of Rows: **5**

Appearance:


Table Label: **Number of microservices calls**

Left Column Label: **Microservice**


Right Column Label: **Calls**

Data Format 


Number of Rows

5 

Sorting

Descending (High to Low) 

Value Formatting

Select option 


Appearance 

Table Label

Number of microservices calls

Left Column Label

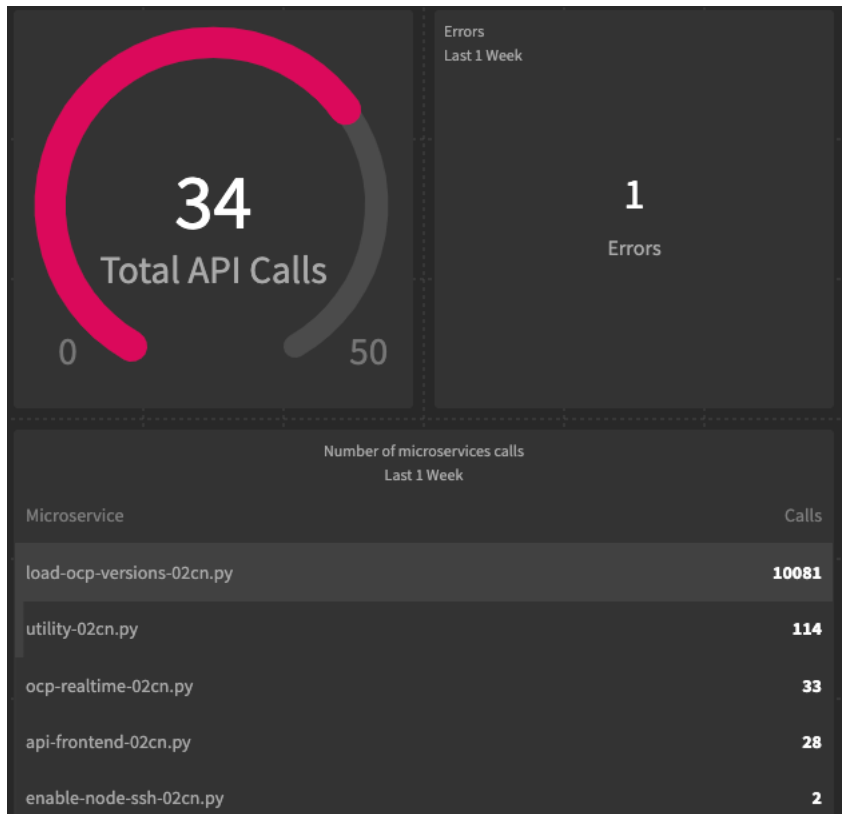
Microservice

Right Column Label

Calls

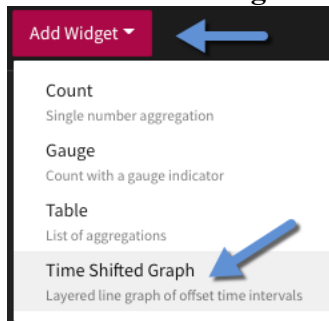
Finally, to make the table look nice, resize the table so it fits under the Total API Calls and Errors Widgets.

Your table should look like this:



The last widget we are going to create is a graph showing the total number of microservice calls compared week to week.

Click on **Add Widget** and then select **Time Shifted Graph**.



Click on the graph to customize it:



Enter the following values:

Field: **app**

Field Value: **ANY**

Advanced Filtering:

(app:ocp-realtime AND Starting Openshift Realtime) OR (app:utility-02cn AND Starting Zero To Cloud Native Utility) OR (app:api-frontend-02cn AND Zero to Cloud Native API Starting) OR (app:enable-node-SSH-02cn AND Enabling SSH on Worker) OR (app:load-OCP-versions-02cn AND Starting to load)

Operation: **Counts**

Duration: **Today vs Yesterday**

Label: **Microservices Calls**

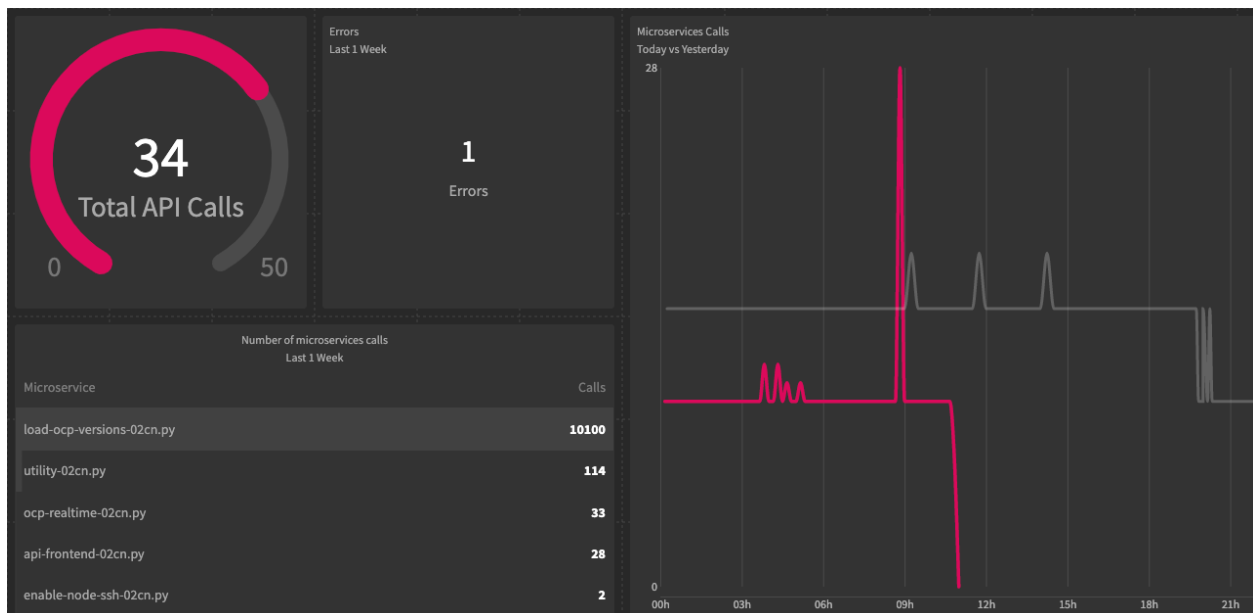
The image shows a configuration panel with a dark background. It contains several sections with labels and input fields:

- Field**: A dropdown menu showing 'app'.
- Field Value**: A dropdown menu showing 'ANY'.
- Advanced Filtering**: A text input field containing the query '(app:ocp-realtime AND Starting Openshift Realtime) OR (app:utility-02cn AND Starting Zero To Cloud Native Utility) OR (app:api-frontend-02cn AND Zero to Cloud Native API Starting) OR (app:enable-node-SSH-02cn AND Enabling SSH on Worker) OR (app:load-OCP-versions-02cn AND Starting to load)'. This field is highlighted with a red border.
- Hide Advanced Filtering**: A button.
- Operation**: A dropdown menu showing 'Counts'.
- Duration**: A dropdown menu showing 'Today vs Yesterday'.
- Appearance**: A dropdown menu.
- Label**: A text input field containing 'Microservices Calls'.

The resulting graph should look something like this:



To make this screen look nice, drag the time series graph next to the other graphs and expand it for better readability. Your screen should now look something like this.



The last thing we need to do is save the screen. Click on **Save Screen**. Enter **Overview** for the Name, **DEV Zero to Cloud Native** (and click **Add this as a new screen category**), finally click **Save**.

So now we have a good graphical overview of which APIs are being called, how many errors occurred and some insight into the actual microservices that are being called as well.

Keep in mind, this is just an example of what we can do with LogDNA. In a future session, we will dive into RedHat Service Mesh and gain much more visibility into our microservices.

Now we have our logging dashboard setup so we can see what is going with our application both at the API and micro-service level and we are getting alerts to when something goes wrong with the application. In the next section, we will go through how to export log entries for users to see the status of their long running API requests.