NurtchDocs Documentation

Release 0.0.1

Amit Rathi

Jun 23, 2018

Table of Contents:

is an internal documentation platform based on . It's used by teams to write executable runbooks for quick incident response. Notebook supports markdown text, images, executable code, and output all within the single document served in a browser.

Nurtch is self hosted for complete control, security, and access to your infrastructure in VPC. All Notebooks are stored in S3 bucket you configure. *See this* for installation.

While Nurtch is great for storing runbooks, it's also suitable to share any internal documentation within team. You can write onboarding guides, retrieve metrics, automate walk-up requests and such. This documentation is split into two sections:

- **Nurtch Platform** We talk about the actual Jupyter UI that's used to write, edit and execute Notebooks. We provide ability to search documents, publish them to S3, store credentials, add team members to Nurtch etc. We also talk about some useful features that are built into Jupyter Notebooks.
- **Rubix Library** Rubix ™ is a Python library that simplifies common DevOps actions by leveraging infrastructure APIs. Simply put, we abstract the complexity of communicating with AWS/Kubernetes/<Your favourtute tool> APIs.

  Here are some examples of Rubix methods:

  - `plot_metrics` method fetches metrics data from cloudwatch and renders the graph in Notebook.
  - `rollback_deployment` quickly rollbacks an ECS service to any prior version. First it communicates with ECS to retrieve rollback candidates. Performs a rollback deployment to a version that you select & shows deployment progress.

CHAPTER 1

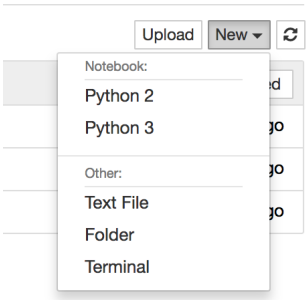
Nurtch Platform

1.1 Install

Here are for setting up Nurtch. Up to 10 Notebooks and unlimited users are free forever. Beyond that, see . To get in touch, or .

1.2 Creating New Runbook

Once you are set up and logged into Nurtch, you can create Notebooks/plain text files/directories. Click *New* and choose the appropriate option.

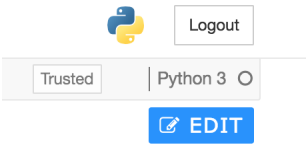


You can choose programming language while creating Notebooks. Thanks to Jupyter, Nurtch supports execution of . Python kernel comes installed out-of-the-box. For other language kernels and we'll help you with the build.

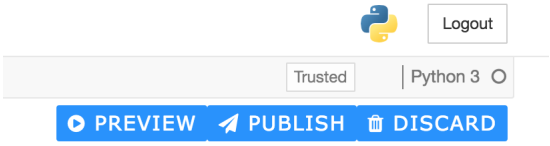
1.3 Editing Runbook

Any Notebook is opened in the viewing mode by default. You can view content, execute code cells, and see output. Editing is disabled to avoid committing unintentional changes. Click on the *EDIT* button if you want to edit the

Notebook.



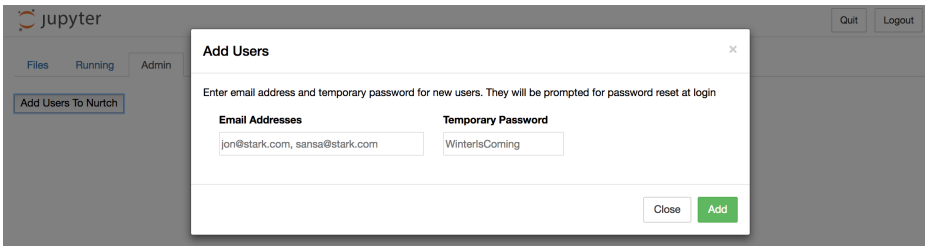
Once editing is enabled you can modify the Notebook as you wish. After editing is done you can *Preview* the Notebook, if you are satisfied with modifications then *Publish* the changes to S3. At any time you can *Discard* your changes to go back to original version.



1.4 Add Users to Nurtch

It's easy to invite your teammates to Nurtch. Go to the *Admin* tab after you login and click on *Add Users To Nurtch*. Type in email addresses of users you want to add (separated by comma). Key in a temporary password for them to login with.

Once added these users can login with their email address and the temporary password. They are forced to change the password when they login for the first time. Note that, we do not send email invites via Nurtch to avoid you any SMTP setup. Simply communicate the username (email) and temporary password to newly added users via your regular means of communication.



1.5 Run SQL queries in Notebook

It's super easy to connect to any sql database and run queries against it from the Notebook. There's a that helps you do it. See example below.

```
In [2]: # load the extension
        %load_ext sql

        # connect to database
        %sql postgres://$DB_USER:$DB_PASSWORD@$DB_ENDPOINT:5432/xyz_api_db

        # run your query
        %sql SELECT pid, query_start, query FROM pg_stat_activity ORDER BY query_start;

The sql extension is already loaded. To reload it, use:
  %reload_ext sql
 * postgres://tankman:***@xyz-api-prod-db.cxvlabuzou3z.ap-south-1.rds.amazonaws.com:5432/xyz_api_db
4 rows affected.

Out[2]:
```

pid	query_start	query
5543	2018-05-08 15:22:23.584144+00:00	select pg_sleep(5 * 60);
5122	2018-05-08 15:22:49.450373+00:00	ROLLBACK
5847	2018-05-08 15:22:49.481600+00:00	SELECT pid, query_start, query FROM pg_stat_activity ORDER BY query_start;
4541	None	<insufficient privilege>

1.6 Run shell commands in Notebook

You can run shell commands in the Notebook with the help of `!` operator. You can also use `%%bash` magic to run multi-line bash script. Commands are run on the server where Nurtch is hosted. You can also SSH onto a different machine and run commands there from within the Notebook. See examples below.

In [1]:
```
! df -h  # check disk usage
```

```
Filesystem      Size   Used  Avail Capacity iused               ifree %iused  Mounted on
/dev/disk1s1   466Gi  384Gi   78Gi    84% 2580428 9223372036852195379     0%   /
devfs          190Ki  190Ki    0Bi   100%     658                   0  100%   /dev
/dev/disk1s4   466Gi  3.0Gi   78Gi     4%       3 9223372036854775804     0%   /private/var/vm
map -hosts       0Bi    0Bi    0Bi   100%       0                   0  100%   /net
map auto_home    0Bi    0Bi    0Bi   100%       0                   0  100%   /home
```

In [5]:
```
%%bash
RESTART_SCRIPT="sudo service nginx restart"
STATUS_SCRIPT="sudo service nginx status"

# ssh onto a machine and restart nginx
ssh -i ~/.ssh/2018-us-east-2.pem ec2-user@18.204.202.175 "${RESTART_SCRIPT} && ${STATUS_SCRIPT}"
```

```
Stopping nginx: [  OK  ]
Starting nginx: [  OK  ]
nginx (pid  29647) is running...
```

CHAPTER 2

Rubix Library

Rubix ™ is a Python library that simplifies common DevOps actions by leveraging infrastructure APIs. Simply put, we abstract the complexity of communicating with AWS/Kubernetes/<Your favourtute tool> APIs and presenting the result in Notebook. Explore the integrations below to know more.

There are lots of services/tools we would like to integrate with Rubix. We are adding new integrations everyday and deepening the existing ones. We are prioritizing integrations based on customer requirements. if you are looking for specific integration and we'll be happy to build it for you.

2.1 Cloudwatch

plot_metric(*namespace*, *metric_name*, ***kwargs*)

Fetch metric data from Cloudwatch and render it as a graph inside Notebook.

Parameters

- **namespace** (`str`) – The namespace of the metric e.g. `AWS/EC2`. All .
- **metric_name** (`str`) – Name of the metric e.g. `Latency`. Here's a to list all possible metrics for your namespace.
- ****kwargs** – These are optional. See below.

Keyword Arguments (Optional)

- ***start_time*** **(datetime.datetime)** Time from which to fetch metrics data. Defaults to (end_time - 12 hours)
- ***end_time*** **(datetime.datetime)** Time until which to fetch metrics data. Defaults to current time.
- ***statistics*** **(str)** Metric statistics for your graph e.g. `Minimum`, `Maximum`, `Sum`, `Average`. All . Deaults to `Average`
- ***markers*** **([datetime.datetime])** Markers to indicate timestamp of significant events e.g. you can fetch deployment times with *this method* and plot them as markers to

see metrics's corelation with deployment. Any marker not between `start_time` and `end_time` is simply ignored. Defaults to [].

- ***dimensions*** **(dict)** A name/value pair that uniquely identifies a metric. See and examples below. When not specified all metrics matching the `namespace` and `metric_name` are graphed.
- ***aws_access_key_id*** **(str)** AWS access key of an IAM user to call cloudwatch API. Defaults to environment variable `AWS_ACCESS_KEY_ID`. Can be overwritten per method by supplying this keyword argument.
- ***aws_secret_access_key*** **(str)** AWS secret access key of an IAM user to call cloudwatch API. Defaults to environment variable `AWS_SECRET_ACCESS_KEY`. Can be overwritten per method by supplying this keyword argument.
- ***aws_region*** **(str)** AWS region for the resource whose metrics you are plotting. Defaults to environment variable `AWS_REGION`. Can be overwritten per method by supplying this keyword argument.

Examples

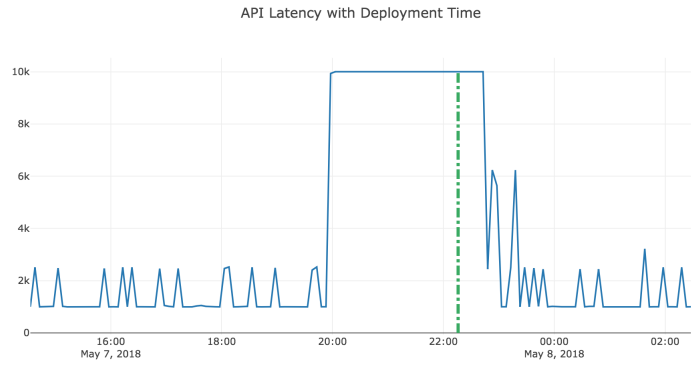
```
from rubix.aws.cloudwatch import plot_metric

# Load balancer P90 latency with deployment time markers
plot_metric(namespace='AWS/ELB',
      metric_name='Latency',
      dimensions={'LoadBalancerName': 'prod-xyz-lb'},
      markers=deployment_times,
      statistics='p90')

# Maximum CPU Utilization across EC2 for a specific time period
plot_metric(namespace='AWS/EC2',
      metric_name='CPUUtilization',
      start_time=datetime.datetime(2018, 04, 25),
      end_time=datetime.datetime(2018, 04, 26)
      statistics='Maximum')
```

Sample Usage and Output

```
In [2]: # plot latency graph with deployment time markers
        from rubix.aws.cloudwatch import plot_metric
        plot_metric(namespace='AWS/ELB',
                    metric_name='Latency',
                    dimensions={'LoadBalancerName': 'prod-xyz-lb'},
                    markers=deployment_times)
```



2.2 Elastic Container Service (ECS)

rollback_deployment (*service*, ***kwargs*)

Quickly rollback an ECS service to any prior version. This method first communicates with ECS to retrieve rollback candidates (prior task definitions that you can rollback to). Performs a rollback deployment to a version that you select. Shows deployment progress.

Parameters

- **service** (str) – The name of your ECS service.
- ****kwargs** – These are optional. See below.

Keyword Arguments (Optional)

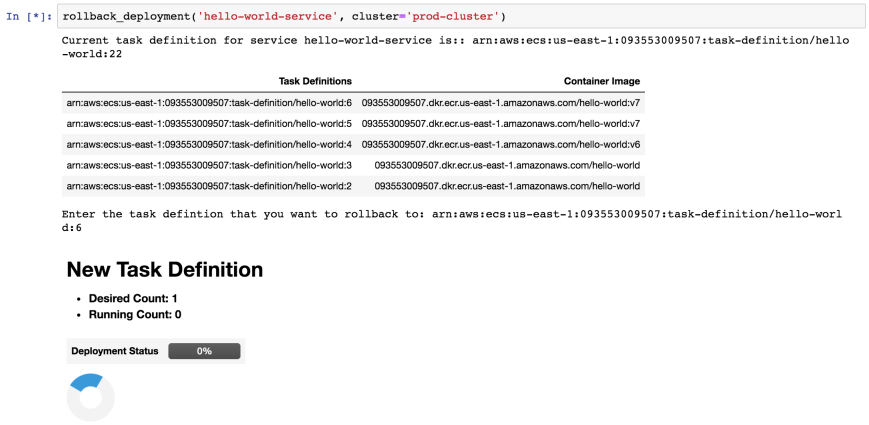
- ***cluster*** **(str)** Name of the ECS cluster to which the service belongs. If the cluster name is not given, ECS uses *default cluster*. We highly recommend putting your services inside clusters and not using *default cluster* unless you are just experimenting.
- ***aws_access_key_id*** **(str)** AWS access key of an IAM user to call ECS APIs. Defaults to environment variable AWS_ACCESS_KEY_ID. Can be overwritten per method by supplying this keyword argument.
- ***aws_secret_access_key*** **(str)** AWS secret access key of an IAM user to call ECS APIs. Defaults to environment variable AWS_SECRET_ACCESS_KEY. Can be overwritten per method by supplying this keyword argument.
- ***aws_region*** **(str)** AWS region where the service resides. Defaults to environment variable AWS_REGION. Can be overwritten per method by supplying this keyword argument.

Examples

```
from rubix.aws.ecs import rollback_deployment

rollback_deployment(service='xyz-api', cluster='prod-cluster')
```

Sample Usage and Output



get_latest_deployment_status (*service*, ***kwargs*)

Retrieve metadata of last deployment on your ECS service. Metadata includes deployment time, desired/pending/running counts, task definition etc.

Parameters

- **service** (str) – The name of your ECS service.
- ****kwargs** – These are optional. See below.

Returns dict – See response section below.

Keyword Arguments (Optional)

- ***cluster*** **(str)** Name of the ECS cluster to which the service belongs. If the cluster name is not given, ECS uses *default cluster*.
- ***aws_access_key_id*** **(str)** AWS access key of an IAM user to call ECS APIs. Defaults to environment variable `AWS_ACCESS_KEY_ID`. Can be overwritten per method by supplying this keyword argument.
- ***aws_secret_access_key*** **(str)** AWS secret access key of an IAM user to call ECS APIs. Defaults to environment variable `AWS_SECRET_ACCESS_KEY`. Can be overwritten per method by supplying this keyword argument.
- ***aws_region*** **(str)** AWS region where the service resides. Defaults to environment variable `AWS_REGION`. Can be overwritten per method by supplying this keyword argument.

Response

- ***id*** **(str)** The ID of the deployment.
- ***taskDefinition*** **(str)** The most recent task definition that was specified for the service to use.
- ***desiredCount*** **(int)** The most recent desired count of tasks that was specified for the service to deploy or maintain.
- ***pendingCount*** **(int)** The number of tasks in the deployment that are in the PENDING status.
- ***runningCount*** **(int)** The number of tasks in the deployment that are in the RUNNING status.
- ***createdAt*** **(datetime.datetime)** The Unix time stamp for when the deployment was created.
- ***updatedAt*** **(datetime.datetime)** The Unix time stamp for when the service was last updated.

Examples

```
from rubix.aws.ecs import get_latest_deployment_status

get_latest_deployment_status(service='hello-world-service', cluster='prod-cluster
↪')
```

Sample Usage and Output

```
In [3]: get_latest_deployment_status('hello-world-service', cluster='prod-cluster')

Out[3]: {'id': 'ecs-svc/9223370511067826302',
         'taskDefinition': 'arn:aws:ecs:us-east-1:093553009507:task-definition/hello-world:23',
         'desiredCount': 1,
         'pendingCount': 0,
         'runningCount': 0,
         'createdAt': datetime.datetime(2018, 5, 8, 19, 12, 29, 505000, tzinfo=tzlocal()),
         'updatedAt': datetime.datetime(2018, 5, 8, 19, 12, 29, 505000, tzinfo=tzlocal())}
```

2.3 Relational Database Service (RDS)

Work in progress. Stay tuned.

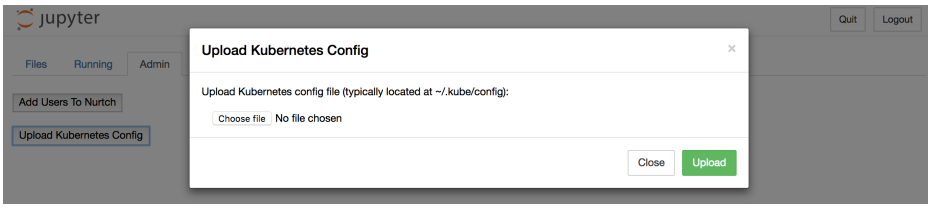
If you are simply looking for a way to run SQL queries, see *Run SQL queries in Notebook*.

2.4 Kubernetes

Now you can operate Kubernetes cluster from Nurtch notebooks. You can either use the familiar *kubectl* commands or use higher level APIs provided by Rubix library.

2.4.1 Setup

Once you login, go to the admin tab and upload Kubernetes config file. The file is typically located at ~/.kube/config. *Tip: You might need to press (CMD + Shift + .) on mac to show hidden files in the finder.*



Once uploaded, wait for a minute for the config to propagate to all the nodes in your cluster. You can verify if the config is propagated as shown below.

```
In [1]: ! ls -l ~/.kube/config
        -rw-r--r-- 1 root root 5769 Jun 21 12:11 /root/.kube/config
```

That's it! Now you can use *kubectl* commands and *rubix.kubernetes.** methods to operate your cluster (examples below).

2.4.2 Command Line Usage

You can upload and use your existing scripts in the notebook or use one-off commands as shown in the examples below.

- List all running services.

```
In [5]: ! kubectl get services
        NAME         TYPE           CLUSTER-IP     EXTERNAL-IP        PORT(S)        AGE
        kubernetes   ClusterIP      100.64.0.1     <none>             443/TCP        4d
        nurtch-1     LoadBalancer   100.69.64.62   a1921bef272e8...   80:31616/TCP   3d
```

- See when deployments occurred in your cluster by checking replica sets.

```
In [12]: ! kubectl get rs --sort-by=metadata.creationTimestamp
         NAME                  DESIRED   CURRENT   READY     AGE
         nurtch-1-76c866799d   0         0         0         4d
         nurtch-1-5c8968b559   0         0         0         3d
         nurtch-1-77df554b74   0         0         0         3d
         nurtch-1-b468b8649    0         0         0         3d
         nurtch-1-674f87f94d   0         0         0         3d
         nurtch-1-6467789fc4   0         0         0         2d
         nurtch-1-6f46b6b5dd   0         0         0         1d
         nurtch-1-6b6bcbb6bb   0         0         0         1d
         nurtch-1-688d45f894   0         0         0         1d
         nurtch-1-849c4f8884   0         0         0         20h
         nurtch-1-8668bd5495   1         1         1         19h
```

- See the rollout history with *kubectl rollout history <resource_name>* and rollback to the version you wish.

```
In [2]: !kubectl rollout undo deployment/nurtch-1 --to-revision=3

        deployment.apps "nurtch-1"
```

- Check the status of your deployment rollout.

```
In [1]: ! kubectl rollout status deployment/nurtch-1

        deployment "nurtch-1" successfully rolled out
```

2.4.3 API Usage

get_latest_deployment_status(*service_name*, *namespace='default'*, *context=None*)

Retrieve metadata of last deployment on your Kubernetes service. Metadata includes deployment time, desired/available/current counts, container image etc.

Parameters

- **service** (str) – Name of your Kubernetes service.
- **namespace** (str) – Namespace under which your service is running, if using namespaces.
- **context** – Context under which your service is running, if using context. Since context specifies the trio of (cluster, user, namespace) you don't need to specify namespace separately while using context.

Returns dict – See response section below.

Response

- ***desiredCount*** **(int)** The desired number of replicas of the application.
- ***availableCount*** **(int)** The number of replicas that are available to your users.
- ***currentCount*** **(int)** The number of replicas that are currently running.
- ***createdAt*** **(datetime.datetime)** The Unix time stamp for when the deployment was created.
- ***containerImage*** **(str)** The name of container image + tag that got deployed.

Examples

```
from rubix.kubernetes import get_latest_deployment_status

get_latest_deployment_status(service_name='nurtch-1')
```

Sample Usage and Output

```
In [1]: from rubix.kubernetes import get_latest_deployment_status

        get_latest_deployment_status(service_name='nurtch-1')

Out[1]: {'desiredCount': 1,
         'availableCount': 1,
         'currentCount': 1,
         'containerImage': 'nurtch/nurtch:30793a725',
         'createdAt': datetime.datetime(2018, 6, 22, 8, 23, 28, tzinfo=tzutc())}
```

Index