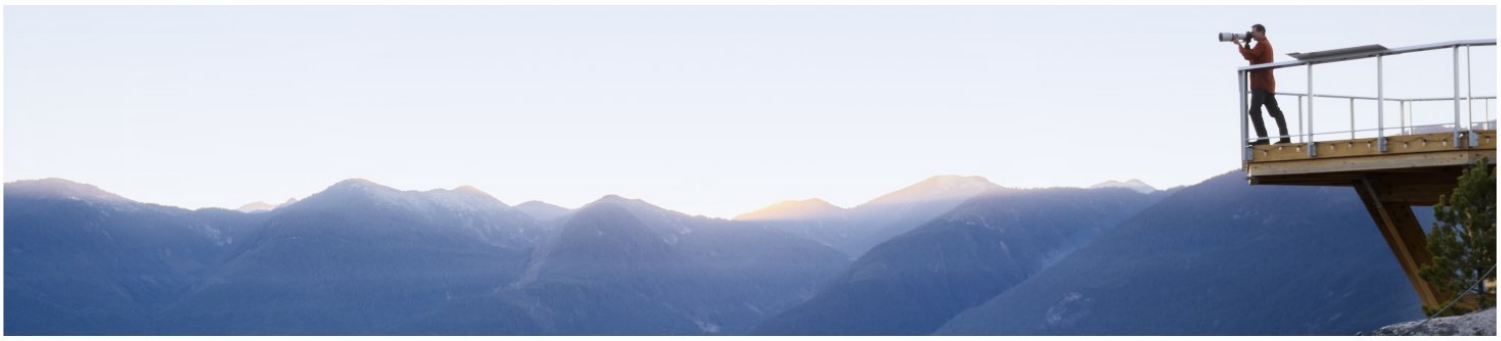
Real User Monitor

Version 9.50, Released May 2018

RUM for Docker – Getting Started

Published May 2018



Legal Notices

Disclaimer

Certain versions of software and/or documents (“Material”) accessible here may contain branding from Hewlett-Packard Company (now HP Inc.) and Hewlett Packard Enterprise Company. As of September 1, 2017, the Material is now offered by Micro Focus, a separately owned and operated company. Any reference to the HP and Hewlett Packard Enterprise/HPE marks is historical in nature, and the HP and Hewlett Packard Enterprise/HPE marks are the property of their respective owners.

Warranty

The only warranties for products and services of Micro Focus and its affiliates and licensors (“Micro Focus”) are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Micro Focus shall not be liable for technical or editorial errors or omissions contained herein. The information contained herein is subject to change without notice.

Restricted Rights Legend

Contains Confidential Information. Except as specifically indicated otherwise, a valid license is required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

Trademark Notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group.

This product includes an interface of the 'zlib' general purpose compression library, which is Copyright © 1995-2002 Jean-loup Gailly and Mark Adler.

Contents

Chapter 1: Introduction

Docker Technology

Containerization or Dockerization is currently a hot and trending topic in the IT world. It allows you to get an entire fleet of inter-connected software products up and running with just a few commands in a matter of minutes. No more tedious installations! No more variability based on the host operating system! You can ship your product with its ecosystem as a single image. And these files are only few hundred megabytes!

Sounds too good to be true, doesn't it? But this is indeed what Docker promises and delivers — the overhead of running an application server reduced to a fraction of the time of the former app-per-VM deployments.

Now you could ask, "With all this simplicity, there must be some downside! Why else doesn't everyone move to Docker right away?"

The downside to containerization is an added complexity in monitoring your applications. Here is why:

- Docker containers have a small footprint. Which means, there is a tendency to have many more applications running on each server.
- All these containers contend for limited resources in terms of RAM, CPU, etc. Over allocation of containers on a Docker host could have a serious impact on all the containers and applications running on that host.
- In such an overcrowded deployment, identifying a single container that starts to exhibit poor performance or low availability now becomes akin to searching for a needle in a haystack.

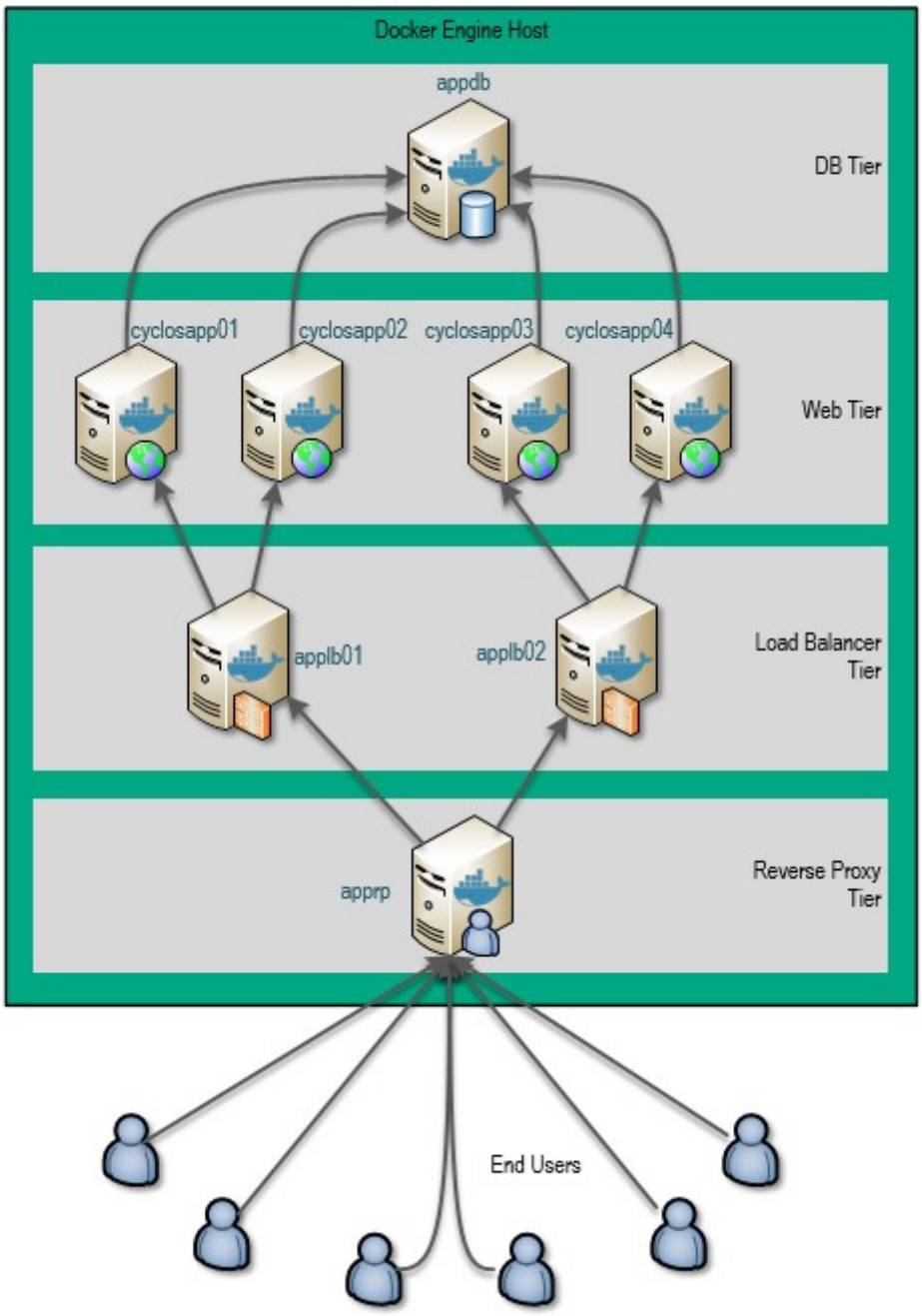
Traditional agent-based solutions do not help much here since having an agent per container defeats the "be-nimble" philosophy of containerization.

Furthermore, containers are usually deployed on the Docker engine's private network bridge. Therefore, IPs or hostnames are not useful anymore. You need container names and image names to correctly identify containers.

So, how do you maintain the same level of performance and availability monitoring for your applications once you containerize them?

Dockerized Multi-tier Application

To help visualize a deployment in the Docker world, let's use the example of a multi-tier banking application, Cyclos. Let's say you recently containerized this application to be entirely hosted on a single Docker server and you would like to ensure the same level of monitoring through RUM as before. The deployment is as follows:



The application has four tiers:

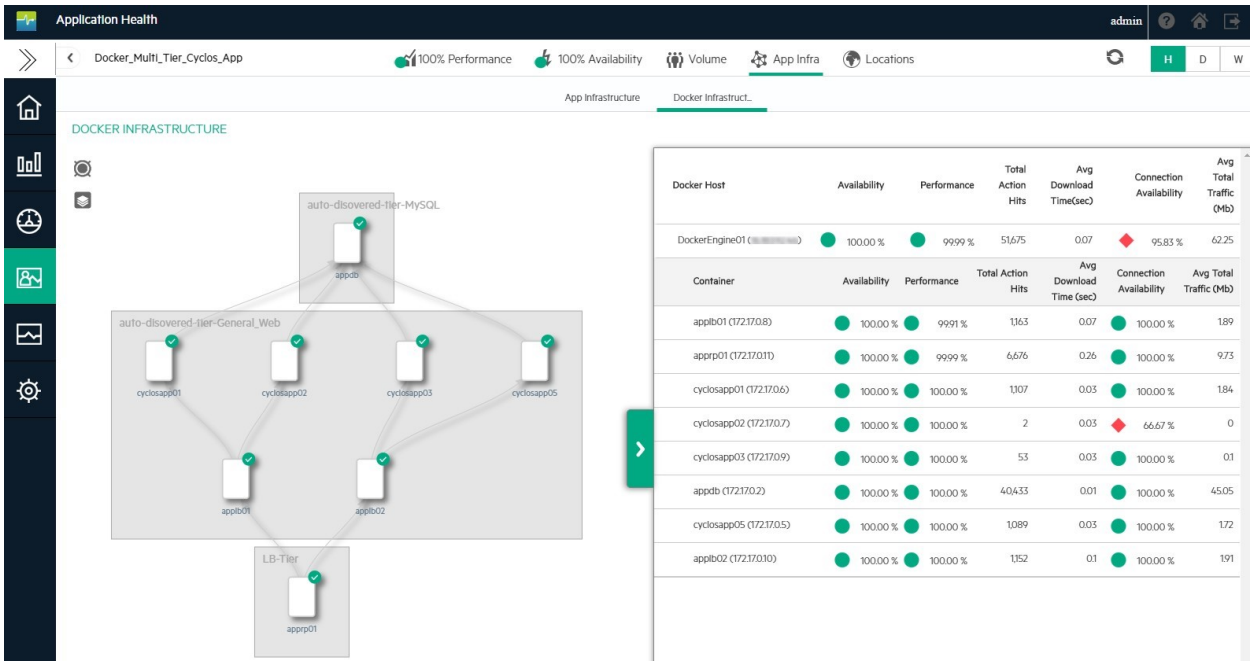
- **Reverse Proxy tier** – Receives requests from End Users and routes these requests to two load balancers
- **Load Balancer tier** – Receives requests from the reverse proxy and distributes these requests between the web servers
- **Web Server tier** – Receives forwarded requests from the Load Balancer
- **Database tier** – Persistence layer for the Web Servers

There are eight containers deployed on the Docker Host:

- A single Reverse Proxy container named **apprp01**
- Two Load Balancer containers named **applb01** and **applb02**
- Four Web Server containers named **cyclosapp01**, **cyclosapp02**, **cyclosapp03**, and **cyclosapp05**
- A single Database server container named **appdb**

```
docker ps | grep app
 docker.io/httpd:latest       "/bin/bash -c 'cp -f "   6 hours ago        Up 6 hours          80/tcp, 0.0.0.0:8080->9090/tcp   apprp01
 docker.io/httpd:latest       "/bin/bash -c 'cp -f "   6 hours ago        Up 6 hours          80/tcp, 0.0.0.0:8081->9090/tcp   applb01
 docker.io/httpd:latest       "/bin/bash -c 'cp -f "   6 hours ago        Up 6 hours          80/tcp, 0.0.0.0:8082->9090/tcp   applb02
 tomcat:7-jre7                "catalina.sh run"        6 hours ago        Up About an hour    0.0.0.0:9296->8080/tcp           cyclosapp05
 tomcat:7-jre7                "catalina.sh run"        6 hours ago        Up 6 hours          0.0.0.0:9294->8080/tcp           cyclosapp03
 tomcat:7-jre7                "catalina.sh run"        6 hours ago        Up 6 hours          0.0.0.0:9292->8080/tcp           cyclosapp01
 tomcat:7-jre7                "catalina.sh run"        6 hours ago        Up 6 hours          0.0.0.0:9293->8080/tcp           cyclosapp02
 mysql:5.5                    "/entrypoint.sh mysql"   6 hours ago        Up 6 hours          3306/tcp                         appdb
```

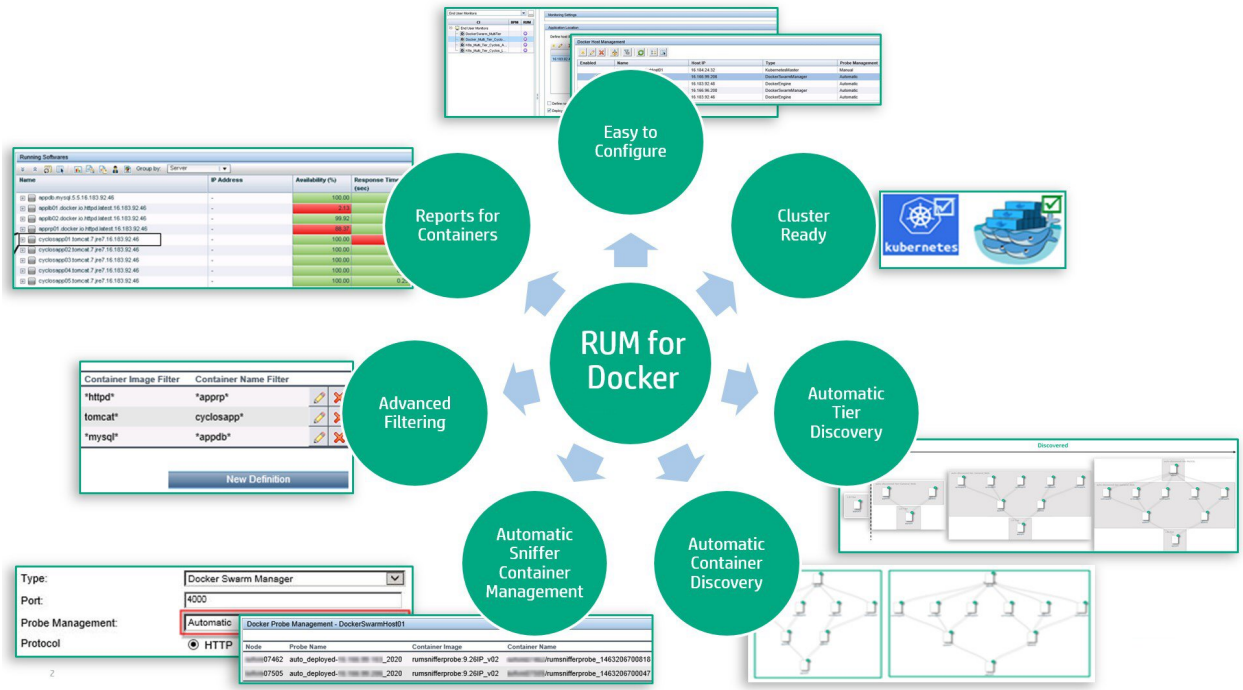
On being monitored by RUM, you would see the following topology discovered and displayed in the Docker infrastructure report:



Chapter 2: Inter-container Traffic Monitoring with RUM

Features

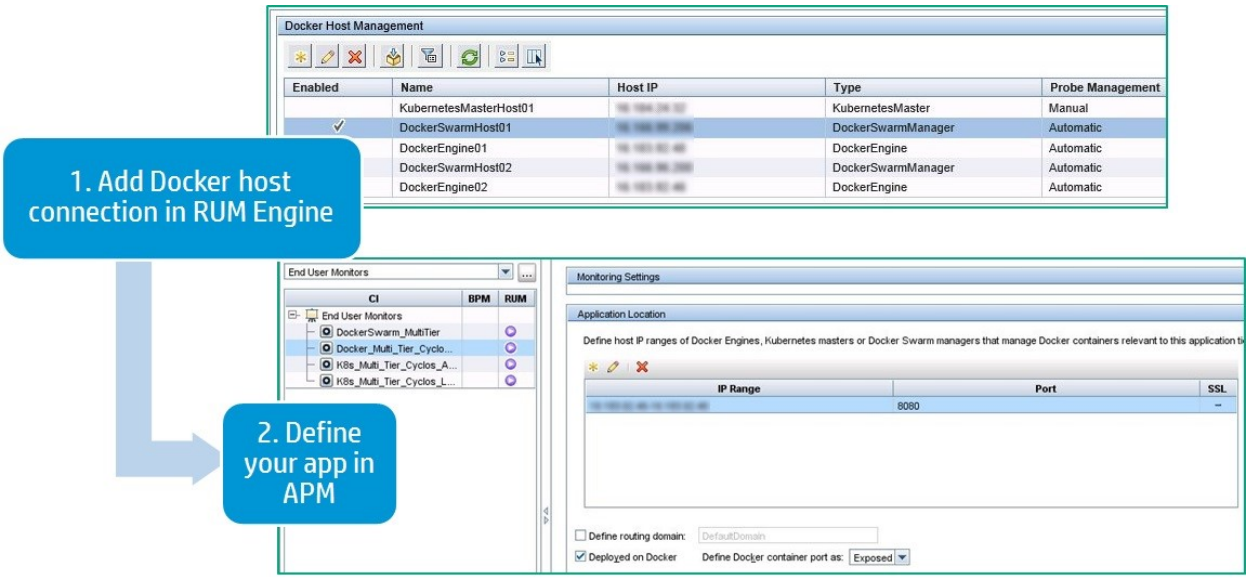
- **Easy 2-step configuration** – Add Docker host connection details to the RUM engine, define your app in APM and your application is monitored
- **Cluster ready** – Support for monitoring Docker Swarm and Kubernetes based clusters
- **Automatic app tier discovery** – Define only the front end. Backend tiers are auto discovered
- **Automatic container discovery** – Ongoing changes such as new container additions during scale-up, container deletions during scale-down, etc., are automatically detected and seamlessly monitored
- **Automatic Sniffer probe container management** – RUM Sniffer probes are automatically deployed by the RUM Engine onto monitored Docker hosts and their health is monitored. Zero manual intervention required for probe installation and maintenance.
- **Advanced filtering based configuration** – Filter containers per app based on wildcards for container names and images
- **All new Docker Infra report** – View Docker topology (Docker hosts and their associated containers), application tier topology, and Docker container interaction graphs in a single report
- **Docker data in regular RUM reports** – View data in regular RUM reports in the context of Docker container names, container images, and the Docker host



Prerequisites

- Docker Engines hosting the containers to be monitored must be version 1.10.3 or higher.
- For monitoring Docker Swarms, the Swarm managers must be version 1.1.3 or higher
- For monitoring Kubernetes clusters, the Kubernetes master must be version 1.1.7 or higher
- RUM Engine and APM must be version 9.30 or higher.
- All servers to be monitored under a RUM application with the **Deployed on Docker** option enabled must be containers with bridge networking. Containers with host networking (started with --net 'host') must be monitored as regular RUM applications (with the **Deployed on Docker** option disabled).
- Remote API ports of monitored Docker Engines, Docker Swarm managers, or Kubernetes Masters must be enabled. See Appendix A: "Enabling Remote API Access" on page 21 for details.
- For monitoring configuration, keep a list of your Docker Engine, Docker Swarm manager, or Kubernetes Master host IP addresses and a list of your container's private and/or exposed ports handy for configuration.

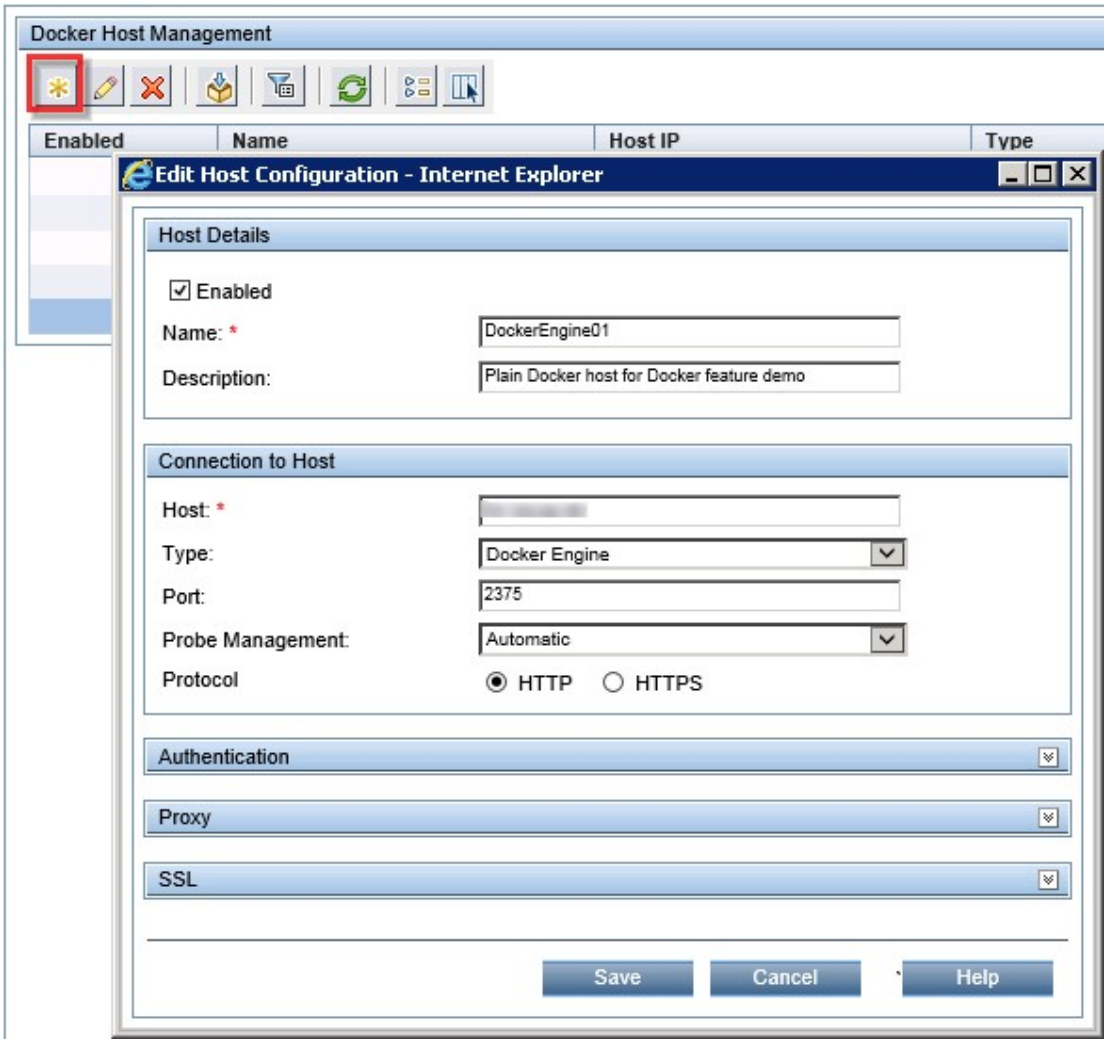
Step-by-step Guide



Follow these steps to setup RUM monitoring for a Dockerized Application:

1. Connect the RUM Engine to the Docker host and deploy the Sniffer probe container:
   a. Ensure that the API port on your Host is open and accessible from the RUM Engine server. See Appendix A:"Enabling Remote API Access" on page 21 for instructions.
   b. Use the Docker Host Management screen to configure the connection to the Docker Host (standalone Docker Engine, Kubernetes Master, or Docker Swarm Manager). Important fields to be noted are:

- **Host:** Provide the IP of the Docker Host
- **Type:** Select the type of Docker Host being managed
- **Port:** Provide the exposed API port of the Docker Host
- **Probe Management:** Select whether Sniffer Probe containers should be automatically managed by the engine or not (automatic probe management is currently supported for hosts of type Docker Engine and Docker Swarm Manager).



For further details on the various fields, refer to the *Docker Host Management* section of the RUM Administration Guide, available on the Software Support web site (https://softwaresupport.softwaregrp.com/).

c. If Automatic Probe Management has been selected for a Docker Host, RUM can automatically deploy one RUM Sniffer probe container per Docker Host node.

If Manual Probe Management has been selected for a Docker Host:

i. Run the following command on the Host. For Cluster managed hosts, run the command on each Docker Swarm node or each Kubernetes node. The RUM Sniffer probe image is downloaded automatically from the Docker Hub and the container is started.

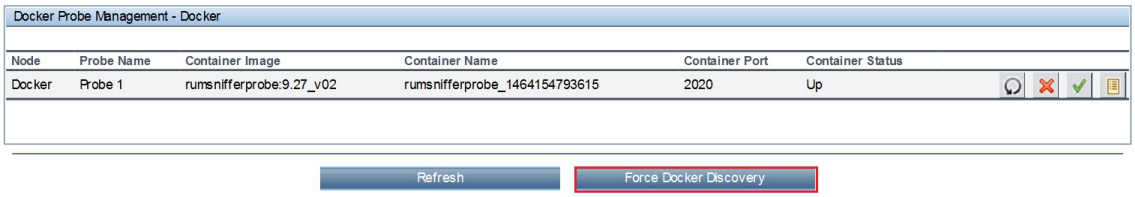
```
docker run -d --name rumsnifferprobe --net 'host' --cap-add=NET_ADMIN
hpsoftware/rumsnifferprobe:latest
```

ii. Run the command `docker ps | grep rumsnifferprobe` to confirm that the new container is running successfully.

iii. Define the manually created probe under **Configuration > Probe management**.

d. RUM Engine runs discovery on enabled Docker Hosts every five minutes. For on demand discovery, click the **Force Docker Discovery** button on the **Docker Probe Management** screen.

Click the  icon on the Docker Host Management screen to navigate to the **Force Docker Discovery** button.



2. Configure your Application as Deployed on Docker.

a. Ensure that Docker support for RUM Applications has been enabled in APM under **Admin > Platform Administration > Setup and Maintenance > Infrastructure Settings > Foundations > EUM Administration > Enable Docker support for RUM applications**.

b. Under **Admin > End User Management > Monitoring**, define a new RUM Application using the following details:

i. Add the IP address of the Docker Engine host, or Kubernetes Master or Docker Swarm Manager.

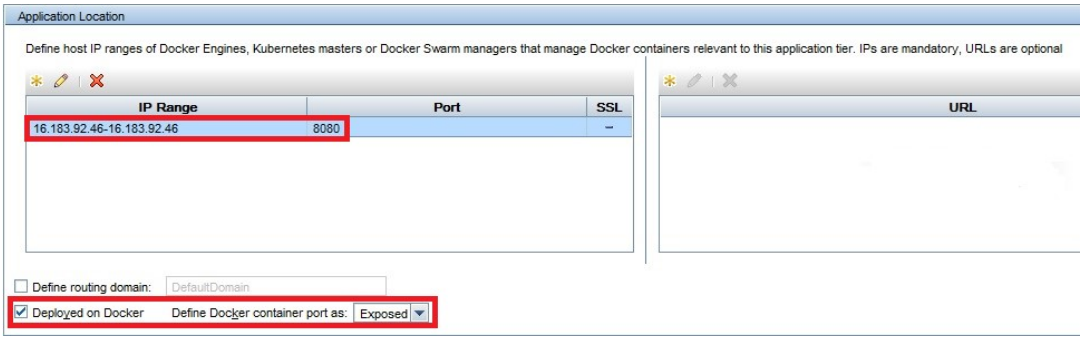
> **Note:** For cluster managed deployments, RUM will automatically discover the nodes associated with your cluster via the cluster manager's API.

ii. Add the port that identifies the set of containers that would run this application. The port can be:

- **Exposed** - The published port of the container available for accessing the application.
- **Private** - The private port of the container that is available only to the other containers on the same bridge.

See Appendix B: "Identifying Exposed vs Private Ports" on page 23 for assistance in identifying which port works best for you.

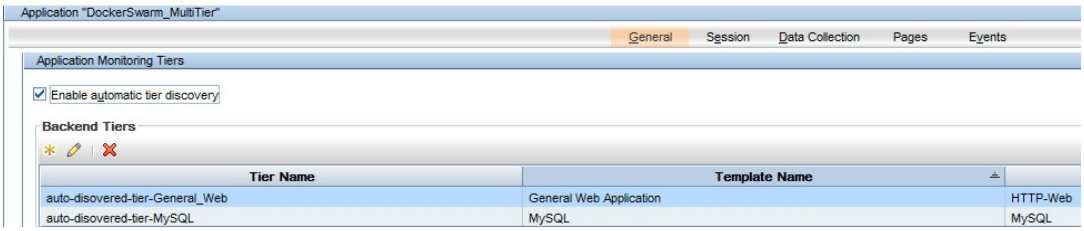
iii. Select the **Deployed on Docker** option and from the **Define Docker container port as** drop down list, select **Private** or **Exposed** based on the type of port defined above.



iv. Assign the application to the correct RUM Engine and select **All probes**.

Note: The engine routes the configuration to the correct RUM Sniffer probe container (the Sniffer probes present on Docker Hosts that have containers that match your application definition).

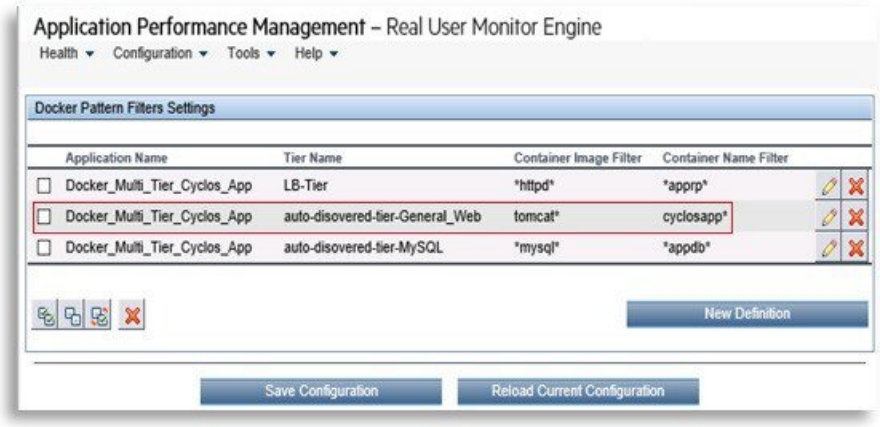
v. To allow RUM to automatically discover backend tiers, select the **Enable automatic tier discovery** option. It is recommended that this option be unchecked once all tiers relevant to your application have been discovered.



3. Advanced configuration – fine tuning container selection per tier.

   To control exactly which containers are mapped to each of your Application Tiers, click the  icon on the Docker Host Management screen to add filters. You can use the wild-card character "*" to build rules for container-to-app tier matching.

   The example below shows the image-container name filter applied to the multi-tier Cyclos application described earlier.



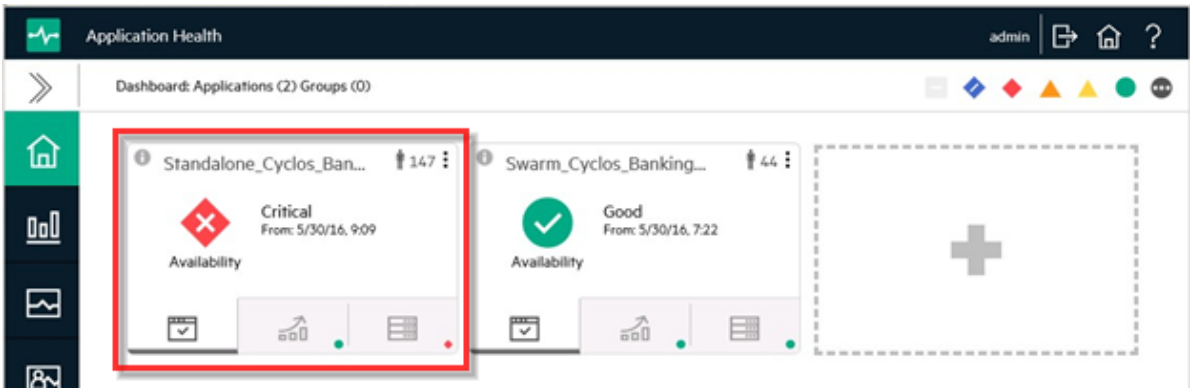
   The image-container name filter combination `tomcat*` and `cyclosapp*` listed for auto-discovered-tier-General_Web ensures that only containers with names such as `cyclosapp01` or `cyclosapp02`

spawned from any version of the tomcat image are monitored for that tier. This also ensures that containers with names like `jpetstore01` are not monitored for this tier.

Chapter 3: Viewing Monitored Data and Topology in Reports

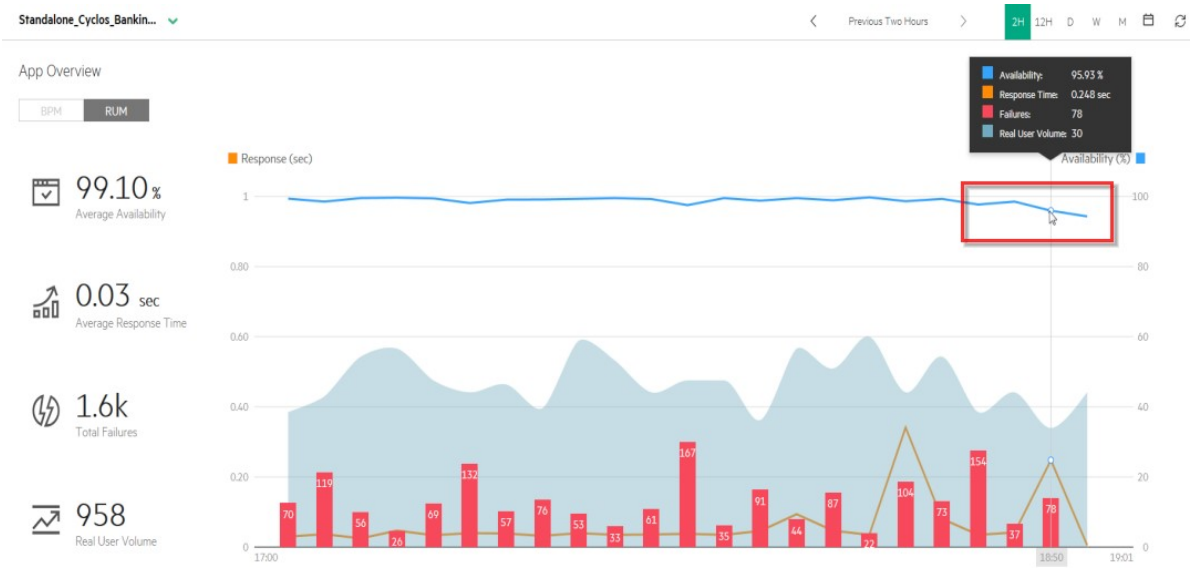
Now that you set up the monitoring, you can start keeping an eye on Docker container level data for your Dockerized Multi-Tier Cyclos application in APM's Application Health.

1. Let's begin with the Application Health Dashboard. The Application Health Dashboard provides a real-time view of the availability and performance of your applications.



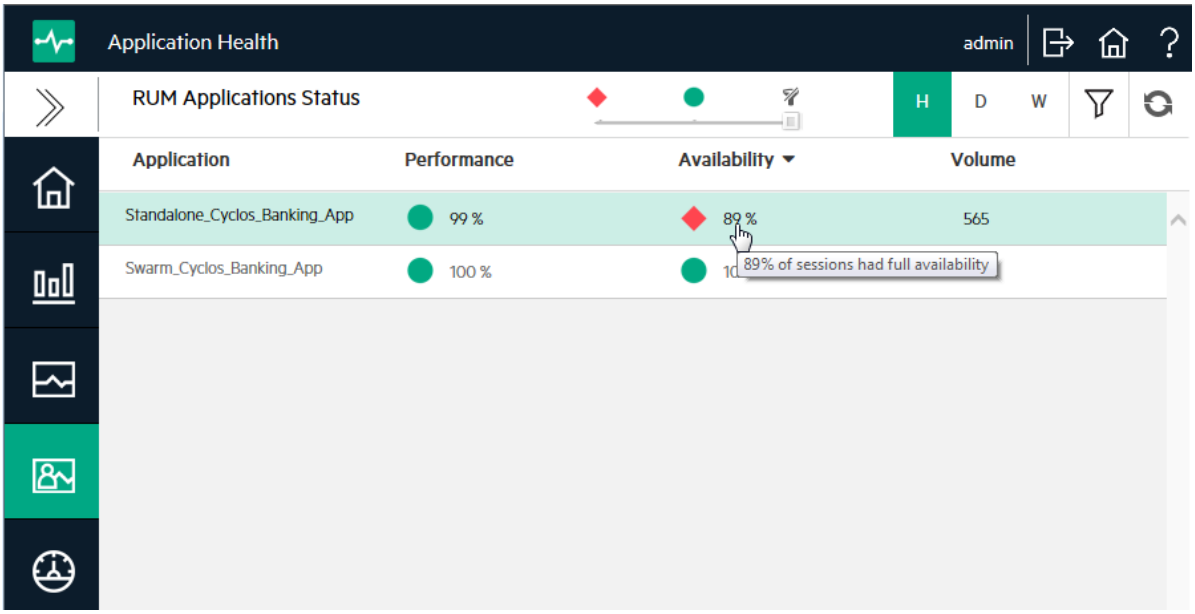
   In this scenario, RUM has detected a problem with the availability of the Cyclos application.

2. Click the red availability icon to drill down into the App Overview report.

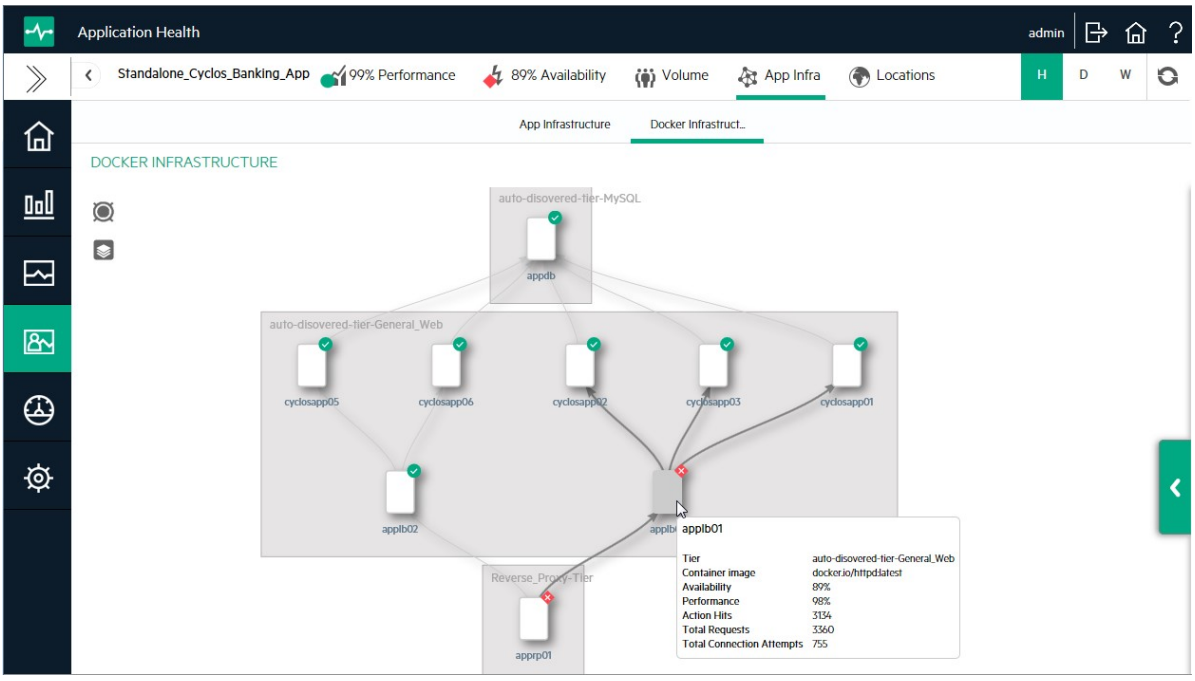


3. Next, click the Real User Monitor  icon on the left panel on the screen.

Here we see RUM data for all applications. The Cyclos application's availability is 89%.



4. To navigate to the Docker Infra report, click **Availability value (89%) > App Infra > Docker Infrastructure**. The Docker Infrastructure report displays:
   - Docker containers that make up your application
   - Network connections made between the containers. Notice the impact the current issue has on your application's containers.
   - End-user facing container **apprp01** that is part of the Reverse-Proxy tier is impacted.
   - One of the two backend load-balancer containers, **applb01**, is impacted.

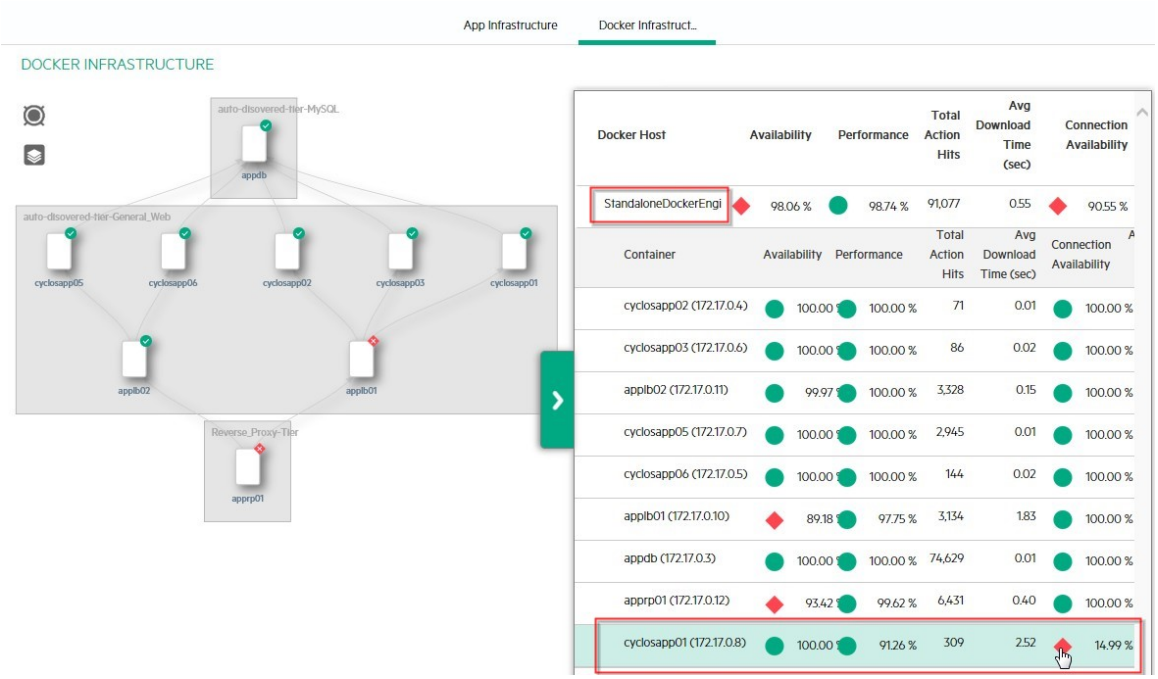


5. Click the icon on the right ( ) to view the deployment of your containers over Docker nodes.

   The Docker Infrastructure report displays:

   - Docker nodes that host the application's containers
   - Application availability, performance, and connection metrics per container and aggregated upwards to the parent node.

     Notice the container named **cyclosapp01** (one of the containers that **applb01** routes traffic to) is facing connection problems. This container is the root cause of the current application level availability drop. Also, we can see that the container is hosted on the Docker Engine node **StandaloneDockerEngine**.



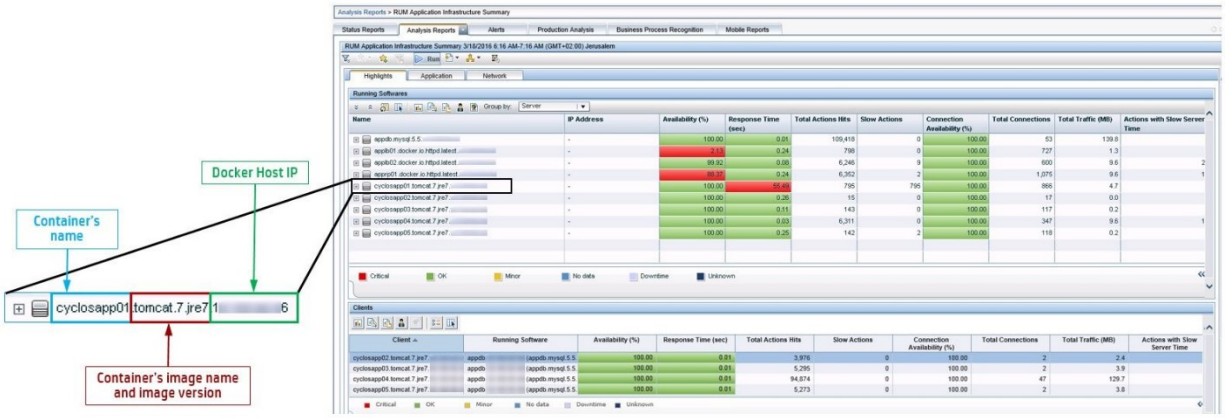
The only remaining action is to resolve the problem by recreating or fixing the affected root-cause container **cyclosapp01** that is hosted on the Docker Engine **StandaloneDockerEngine**.

In addition to the Docker Infrastructure report, Docker containers are also represented in traditional RUM reports such as the Session Analyzer report and Application Infrastructure Summary report, and also within active filters of RUM reports under the **Servers** tab.

Docker containers are identified by their Fully Qualified Container Names (FQCN) in the following format:

```
<container_name>.<image_name>.<image_version>.<docker_host>
```

The following shows the mapping for our representative application, **Cyclos**, in the Application Infrastructure Summary report.


Docker Host IP
Container's name
cyclosapp01.tomcat.7.jre7.1
6
Container's image name and image version

Chapter 4: Troubleshooting

Cannot See Containers

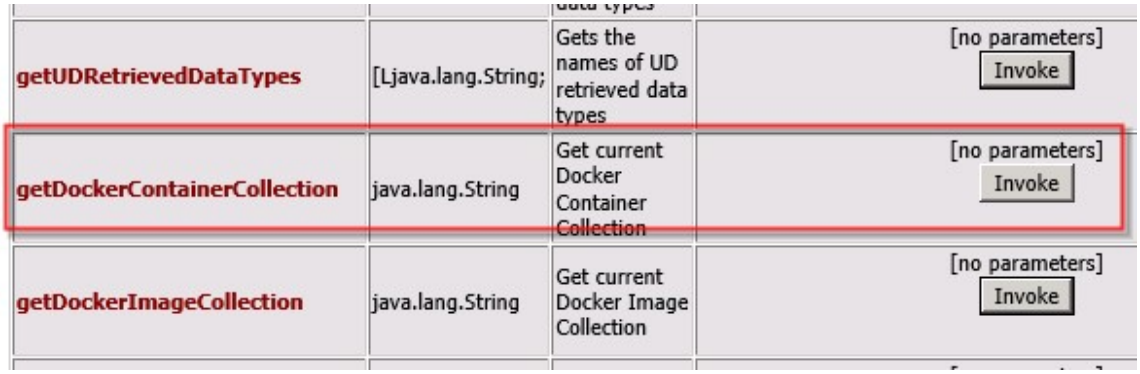
Symptom: After configuring a Docker Host and assigning an application to the Engine, no container(s) appear in the **Application Health > Docker Infrastructure** report.

Troubleshooting Steps:

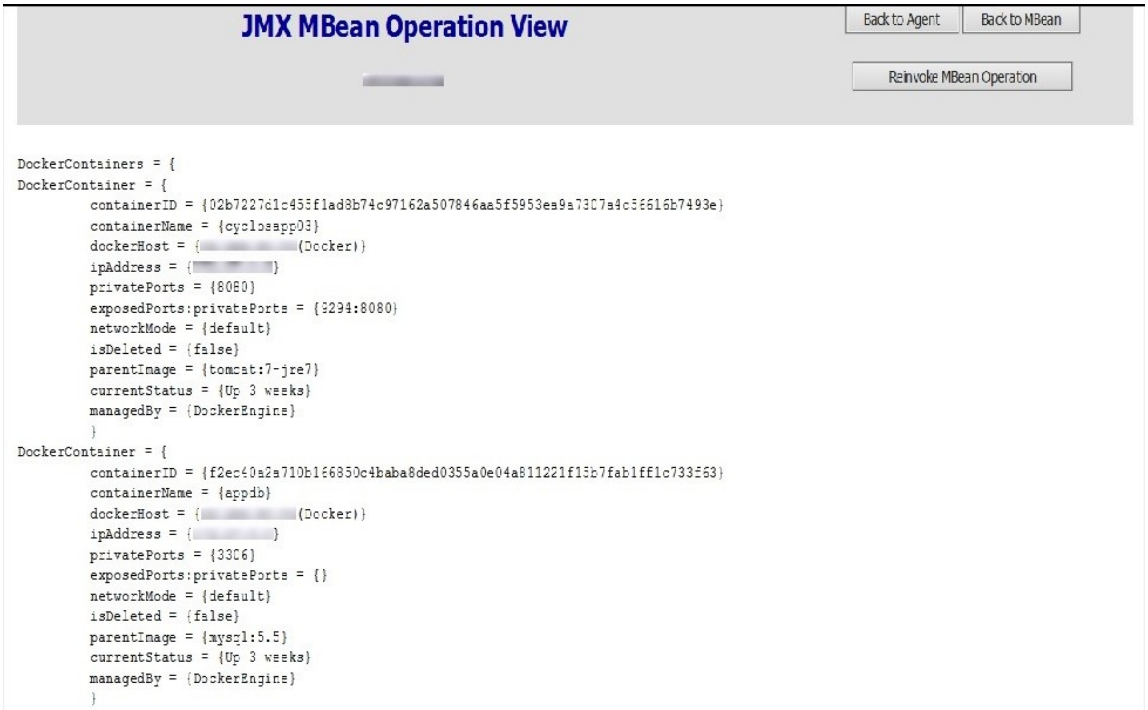
1. Check the API connection between RUM and the Docker Host:

   From the RUM Engine server, access the Docker Host server with a browser using the following URL: **http://<docker_host_ip>:<docker_host_port>/version**

   You should receive a **version.json** file to download in response. If **version.json** is not returned as the response, you may be using the wrong port or IP address to connect to the Docker Host.
2. Next, check the containers retrieved by the Engine:
   a. Open the Engine JMX: **http://<RUM_Engine>:8180/jmx-console/**
   b. Click **RUM.modules** on the left.
   c. Click **service=ConfigurationManagerConf**.
   d. Search for the operation **getDockerContainerCollection** and click **Invoke**.



   e. Verify that the container you expect to see in the RUM reports appears in the output.



f. If you do not see your container listed, check whether the container is actually running on the Docker host.

3. Finally, check the **<RUM>\log\config.manager.log** for any errors or exceptions thrown during RUM Engine's discovery run.

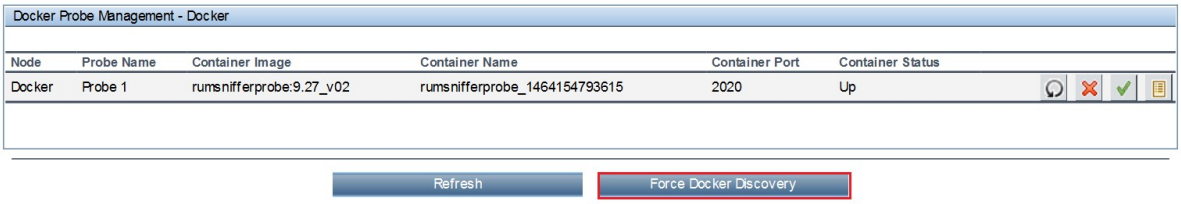
Cannot See Data

Symptom: After configuring a Docker Host and assigning an application to the Engine, no data appears for the application in RUM reports.

Troubleshooting Steps:

For monitoring, RUM requires the RUM Sniffer probe container to run on the Docker Host and a connection to the probe must be established in the Engine's Probe Management page. These steps are handled automatically for Docker Hosts flagged for Automatic Probe Management.

1. From the Docker Host Management page, navigate to the Docker Probe Management page by clicking the  icon. This page displays the list of Nodes managed by your Docker Host and details of the probe container that resides on each node.

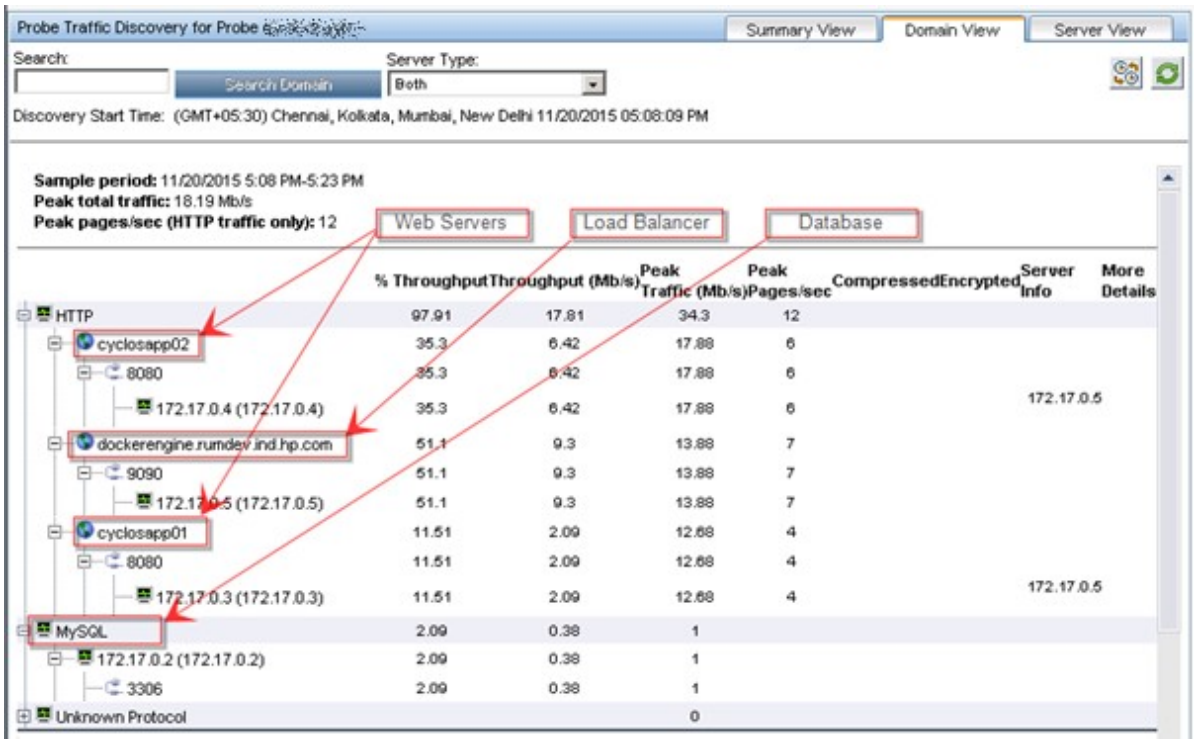


For each probe:

a. Check the Container Status column to ensure that your probe container is currently Up.
b. Click the **Check RUMProbe Process Status** button to check whether the probe's **RUMProbe** process is currently running in the probe container.
c. Click the **Retrieve Container Log** button to check the last 20 lines of the probe container's **capture.log**.
d. Click the **Remove and Recreate Container** button to force a cleanup and re-creation of the probe container.

2. When you are satisfied that the probe container is healthy, enable traffic discovery for a few minutes to view the traffic that the probe actually sees. In **Configuration > Probe Management**, select the probe deployed on the Docker Host that contains your application and click **Probe Traffic Discovery**.

You should see traffic relevant to your application in the discovery result. If you do not see traffic, it could mean that there is no traffic being generated on your application for the probe to capture.



Appendix A: Enabling Remote API Access

Docker Engine

RUM requires access to the Docker Engine's remote API for container discovery. Steps to enable remote access are detailed in the Docker documentation in Bind Docker to another host/port or a UNIX socket.

The sample steps below are specifically for Docker Engine's deployed on Ubuntu 16.04.

1. Open the file **/lib/systemd/system/docker.service**
2. Modify the following line:

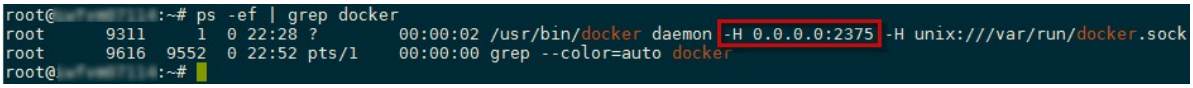
   ```
   ExecStart=/usr/bin/docker daemon -H fd:// -H tcp://0.0.0.0:2375
   ```

3. Reload the configuration and restart the Docker daemon:

   ```
   sudo systemctl daemon-reload

   sudo systemctl restart docker.service
   ```

4. Check that the Docker daemon successfully started with the API port.

5. From the RUM Engine server, access the Docker host server with a browser (like IE) using the following URL: **http://<docker_host_ip>:2375/version**.

   You should receive a **version.json** file to download in response.

6. Use the port configured above (2375 by default) to configure Docker hosts on the RUM Engine.

Docker Swarm

The Docker Swarm API is mostly compatible with the Docker Engine Remote API. As with Docker hosts, RUM leverages the Swarm Manager remote API for container and node discovery and probe deployment.

Steps to enable remote access are detailed in the Docker documentation at Docker Swarm Discovery and Docker Swarm API.

Run the command `ps -ef | grep swarm manage` to determine the API port.

Kubernetes

Like the Docker offerings, Kubernetes API subsystem is available for RUM to perform Container Discovery and Management. By default, it is available on port 8080. Choose **--secure-port** when a secure connection is mandated. Details are available on the Kube-API Server page.

Run the command `ps -ef | grep kube-api` to determine the API port.

```
root@iwf-vm00054:~# ps -ef | grep kube-api
root      1502     1  1 Feb18 ?        1-06:08:22 /opt/bin/kube-apiserver --insecure-bind-address=0.0.0.0
--insecure-port=8080 --etcd-servers=http://127.0.0.1:4001 --logtostderr=true --service-cluster-ip-range=19
2.168.3.0/24 --admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount,ResourceQuota,SecurityConte
xtDeny --service-node-port-range=30000-32767 --client-ca-file=/srv/kubernetes/ca.crt --tls-cert-file=/srv/
kubernetes/server.cert --tls-private-key-file=/srv/kubernetes/server.key
```

Appendix B: Identifying Exposed vs Private Ports

There are two ways to determine exposed and private ports of containers.

- Directly on the Docker Host:

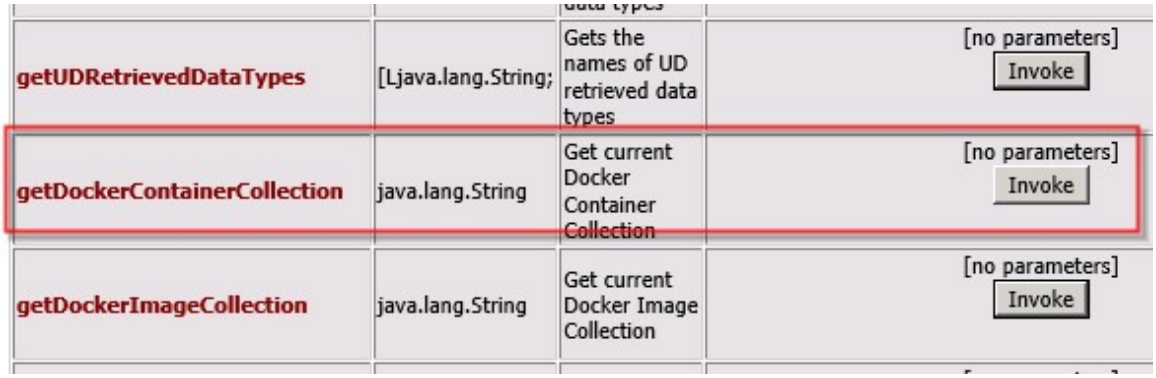
  On the Docker Host, run `docker ps` to show the container ports.

```
docker ps --filter name=cyclosapp*
 IMAGE               COMMAND             CREATED             STATUS              PORTS                    NAMES
 tomcat:7-jre7       "catalina.sh run"   4 days ago          Up 4 days                                    cyclosapp10
 tomcat:7-jre7       "catalina.sh run"   5 days ago          Up 5 days           0.0.0.0:9295->8080/tcp   cyclosapp05
 tomcat:7-jre7       "catalina.sh run"   9 days ago          Up 5 days           0.0.0.0:9293->8080/tcp   cyclosapp02
```

  In the screenshot above:

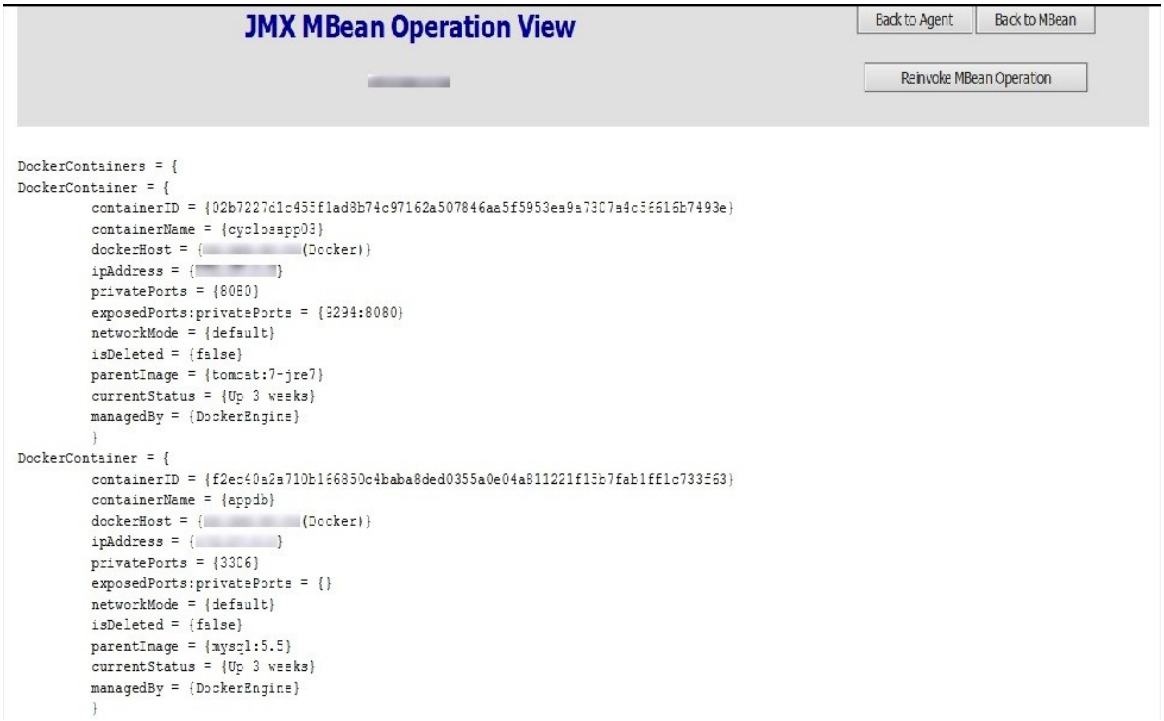
  - **9295** is the exposed port for container cyclosapp05. 8080 is the private port.
  - **9293** is the exposed port for container cyclosapp02. 8080 is the private port.
  - Container **cyclosapp10** has no private or exposed port.

- From the RUM Engine (after connecting the Docker Host to it):
  a. Open the Engine JMX: **http://<RUM_Engine>:8180/jmx-console/**
  b. Click **RUM.modules** on the left.
  c. Click **service=ConfigurationManagerConf**.
  d. Search for the operation **getDockerContainerCollection** and click **Invoke**.



  e. In the sample output screenshot below, we see that container **cyclosapp03** has a private port **8080** and a corresponding exposed port **9294**.

We also see that container **appdb** has a private port **3306**. It has no exposed ports.



Send Documentation Feedback

If you have comments about this document, you can contact the documentation team by email. If an email client is configured on this system, click the link above and an email window opens with the following information in the subject line:

Feedback on RUM for Docker – Getting Started (Real User Monitor 9.50)

Just add your feedback to the email and click send.

If no email client is available, copy the information above to a new message in a web mail client, and send your feedback to docteam@microfocus.com.

We appreciate your feedback!