MODULE 5

CONTENTS

WORKING WITH REMOTE REPOSITORIES

- Docker Hub repositories allow you to share container images with your team, customers, or the Docker community at large.
- Docker images are pushed to Docker Hub through the ***docker push*** command.
- A single Docker Hub repository can hold many Docker images (stored as tags).
- The power of Docker images is that they're lightweight and portable—they can be moved freely between systems.
- You can easily create a set of standard images, store them in a repository on your network, and share them throughout your organization.
  - Or you could turn to Docker Inc., which has created various mechanisms for sharing Docker container images in public and private.

- The most prominent among these is Docker Hub, the company’s public exchange for container images.
- Many open source projects provide official versions of their Docker images there, making it a convenient starting point for creating new containers by building on existing ones, or just obtaining stock versions of containers to spin up a project quickly.
  - And you get one private Docker Hub repository of your own for free.

Explore Docker Hub

- The easiest way to explore Docker Hub is simply to browse it on the web. From the web interface, you can search for publicly available containers by name, tag, or description.
  - From there, everything you need to download, run, and otherwise work with container images from Docker Hub comes included in the open source version of Docker—chiefly, the docker pull and docker push commands.

Docker Hub organizations for teams

- If you’re using Docker Hub with others, you can create an organization, which allows a group of people to share specific image repositories.
- Organizations can be further subdivided into teams, each with their own sets of repository privileges. Owners of an organization can create new teams and repositories, and assign repository read, write, and admin privileges to fellow users.

Docker Hub repositories

- Docker Hub repositories can be *public or private*. Public repositories can be searched and accessed by anyone, even those without a Docker Hub account.
- Private repos are available only to users you specifically grant access to, and they are not publicly searchable. Note that you can turn a private repo public and vice versa.
  - Note also that if you make a private repo public, you'll need to ensure that the exposed code is licensed for use by all and sundry.
- Docker Hub does not offer any way to perform automatic license analysis on uploaded images; that's all on you.
- While it is often easiest to search a repository using the web interface, the Docker command line or shell also allows you to search for images.
  - Use ***docker search*** to run a search, which returns the names and descriptions of matching images.

- Certain repositories are tagged as official repositories.
  - These provide curated Docker images intended to be the default, go-to versions of a container for a particular project or application (e.g. Nginx, Ubuntu, MySQL).
  - Docker takes additional steps to verify the provenance and security of official images.
- If you yourself maintain a project that you want to have tagged as an official repository on Docker Hub, make a pull request to get the process started.
- Note, however, that it is up to Docker to determine whether your project is worthy of being included.

Docker push and Docker pull

- Before you can push and pull container images to and from the Docker Hub, you must connect to the Docker Hub with the docker login command, where you'll submit your Docker Hub username and password.
  - By default, docker login takes you to Docker Hub, but you can use it to connect to any compatible repository, including privately hosted ones.
- Generally, working with Docker Hub from the command line is fairly straightforward.
  - Use *docker search* as described above to find images, *docker pull* to pull an image by name, and *docker push* to store an image by name.
  - A *docker pull* pulls images from Docker Hub by default unless you specify a path to a different registry.

- Note that when you push an image, it’s a good idea to tag it beforehand.
  - Tags are optional, but they help you and your team disambiguate image versions, features, and other characteristics.
  - A common way to do this is to automate tagging as part of your image build process—for instance, by adding version or branch information as tags to images.

Commands to working with remote repositories

docker search

- docker search command search the Docker Hub for images.
- ***docker search [OPTIONS] TERM***
- Examples
  - **Search images by name :** *docker search busybox*
  - **Search images using stars :** *docker search --filter stars=3 busybox*
    - This example displays images with a name containing ‘busybox’ and at least 3 stars.

Commands to working with remote repositories

docker push

- docker push command push an image or a repository to a registry.
- ***docker push [OPTIONS] NAME[:TAG]***
- Example
  - **Push a new image to a registry** *: docker image push --all-tags registry-host:5000/myname/myimage*
    - When pushing with the --all-tags option, all tags of the *registry-host:5000/myname/myimage* image are pushed.

Commands to working with remote repositories

docker pull

- docker pull command Pull an image or a repository from a registry.
- ***docker pull [OPTIONS] NAME[:TAG|@DIGEST]***
- Example
  - **Pull an image from Docker Hub**: *docker pull debian*
    - If no tag is provided, Docker Engine uses the :latest tag as a default. This command pulls the debian:latest image.

Automated builds on Docker Hub

- Container images (hosted on Docker Hub) can be built automatically from their components hosted in a repository.
  - With automated builds, any changes to the code in the repo are automatically reflected in the container; you don’t have to manually push a newly built image to Docker Hub.
- Automated builds work by linking an image to a build context, i.e. a repo containing a Dockerfile that is hosted on a service like **GitHub or Bitbucket**.
  - Although Docker Hub limits you to one build every five minutes, and there’s no support yet for Git large files or Windows containers, automated builds are nevertheless useful for projects updated daily or even hourly.

- If you have a paid Docker Hub account, you can take advantage of **parallel builds**.
  - An account eligible for **five parallel builds** can build containers from up to five different repositories at once.
  - Note that each individual repository is allowed only one container build at a time; the parallelism is across repos rather than across images in a repo.
- Another convenience mechanism for developers in Docker Hub is **webhooks**.
  - Whenever a certain event takes place involving a repository—an image is rebuilt, or a new tag is added— Docker Hub can send a POST request to a given endpoint.
  - You could use webhooks to automatically deploy or test an image whenever it is rebuilt, or to deploy the image.

SECURITY AND ISOLATION

- To use Docker safely, you need to be aware of the potential security issues and the major tools and techniques for securing container-based systems.
- We begins by exploring some of the issues surrounding the security of container-based systems that you should be thinking about when using containers.

Things to Worry About

What sorts of security issues should you been thinking about in a container-based environment?

- **Kernel exploits**
  - Unlike in a VM, the kernel is shared among all containers and the host, magnifying the importance of any vulnerabilities present in the kernel. Should *a container cause a kernel panic, it will take down the whole host*. In VMs, the situation is much better: an attacker would have to route an attack through both the VM kernel and the hypervisor before being able to touch the host kernel.
- **Denial-of-service (DoS) attacks**
  - All containers share kernel resources. *If one container can monopolize access to certain resources*—including memory and more esoteric resources such as user IDs (UIDs)—*it can starve out other containers on the host*, resulting in a denial-of-service, where legitimate users are unable to access part or all of the system

- **Container breakouts**
  - An attacker *who gains access to a container should not be able to gain access to other containers or the host*. Because users are not namespaced, any process that breaks out of the container will have the same privileges on the host as it did in the container; if you were root in the container, you will be root on the host.
  - This also means that you need to worry about potential privilege escalation attacks—where a user gains elevated privileges such as those of the root user, often through a bug in application code that needs to run with extra privileges.
- **Poisoned images**
  - How do you know that the images you are using are safe, haven't been tampered with, and come from where they claim to come from? If an attacker can trick you into running his image, *both the host and your data are at risk*. Similarly, you want to *be sure the images you are running are up to date and do not contain versions of software with known vulnerabilities*.

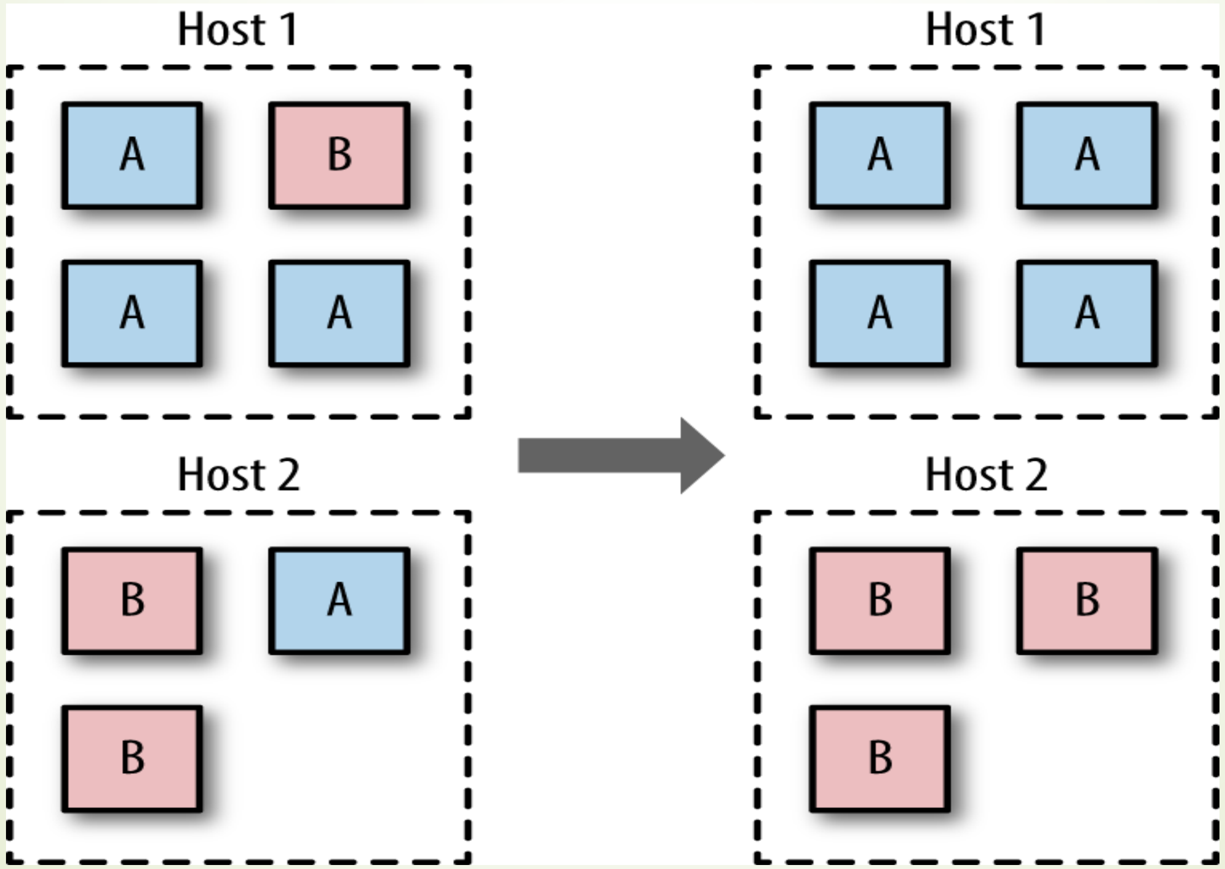
- **Compromising secrets**
- When a container accesses a database or service, it will likely require some secret, such as *an API key or username and password*. An attacker who can get access to this secret will also have access to the service.
    - This problem becomes more acute *in a microservice architecture* in which containers are constantly stopping and starting, as compared to an architecture with small numbers of long-lived VMs.
- The simple fact is that both Docker and the underlying Linux kernel features it relies on are still young and nowhere near as battle hardened as the equivalent VM technology.
- For the time being at least, *containers do not offer the same level of security guarantees as VMs*.

- **Defence-in-Depth**
- The defence for your system should *consist of multiple layers*. For example, your *containers will most likely run in VMs* so that if a container-breakout occurs, another level of defence can prevent the attacker from getting to the host or containers belonging to other users.
- *Monitoring systems* should be in place to alert admins in the case of unusual behavior. *Firewalls* should restrict network access to containers, limiting the external attack surface.
- **Least Privilege**
- Another important principle to adhere to is least privilege; each process and container should run with the *minimum set of access rights and resources* it needs to perform its function.
- The main benefit of this approach is that if one container is compromised, the attacker should still *be severely limited in being able to access further data or resources.*

- In regard to least privilege, you can take many steps to reduce the capabilities of containers, such as:
  - Ensure that processes in containers *do not run as root* so that exploiting a vulnerability present in a process does not give the attacker root access.
  - Run *filesystems as read-only* so that attackers cannot overwrite data or save malicious scripts to file.
  - *Cut down on the kernel calls* a container can make to reduce the potential attack surface.
  - *Limit the resources* a container can use to avoid DoS attacks where a compromised container or application consumes enough resources (such as memory or CPU) to bring the host to a halt.

- *The key point to remember is that the more checks and boundaries you have in place, the greater the chances of stopping an attack before it can do real harm.*
  - Used incorrectly, Docker will reduce the security of the system by opening up new attack vectors (an attack vector is a method or pathway used by a hacker to access or penetrate the target system).
  - Used correctly, it will improve security by adding further levels of isolation and limiting the scope of applications to damage the system.

- **Segregate Containers by Host**
  - If you have a multitenancy setup where you are *running containers for multiple users* (whether these are internal users in your organization or external customers), *ensure each user is placed on a separate Docker host*, as shown in Figure.

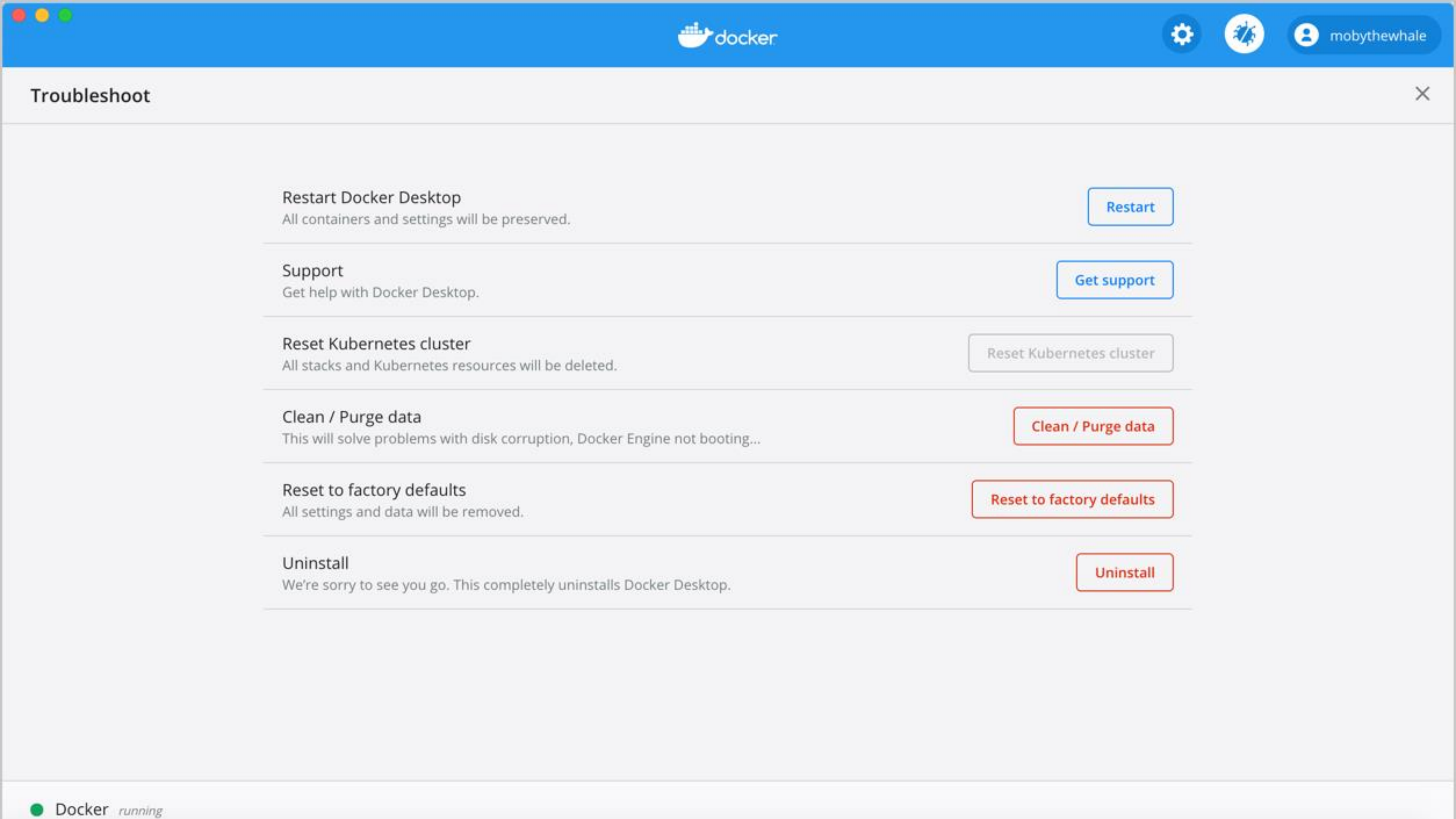


This is less efficient than sharing hosts between users and will result in a higher number of VMs and/or machines than reusing hosts but is important for security.

- The main reason is to prevent container breakouts resulting in a user gaining access to another user's containers or data.
- *If a container breakout occurs, the attacker will still be on a separate VM or machine and unable to easily access containers belonging to other users.*
- Similarly, if you have containers that process or store sensitive information, keep them on a host separate from containers handling less sensitive information and, in particular, away from containers running applications directly exposed to end users.
  - For example, containers processing credit-card details should be kept separate from containers running the Node.js frontend.
- Segregation and use of VMs can also provide added protection against DoS attacks; users won't be able to monopolize the memory on the host and starve out other users if they are contained within their own VM.

TROUBLESHOOTING

- It contains information on how to diagnose and troubleshoot Docker Desktop issues, request Docker Desktop support, send logs and communicate with the Docker Desktop team, use our forums and Success Center, browse and log issues on GitHub, and find workarounds for known problems.
- Troubleshoot
- Choose > Troubleshoot from the menu bar to see the troubleshoot options.



Troubleshoot

Restart Docker Desktop
All containers and settings will be preserved.

Support
Get help with Docker Desktop.

Reset Kubernetes cluster
All stacks and Kubernetes resources will be deleted.

Clean / Purge data
This will solve problems with disk corruption, Docker Engine not booting...

Reset to factory defaults
All settings and data will be removed.

Uninstall
We're sorry to see you go. This completely uninstalls Docker Desktop.

● Docker *running*

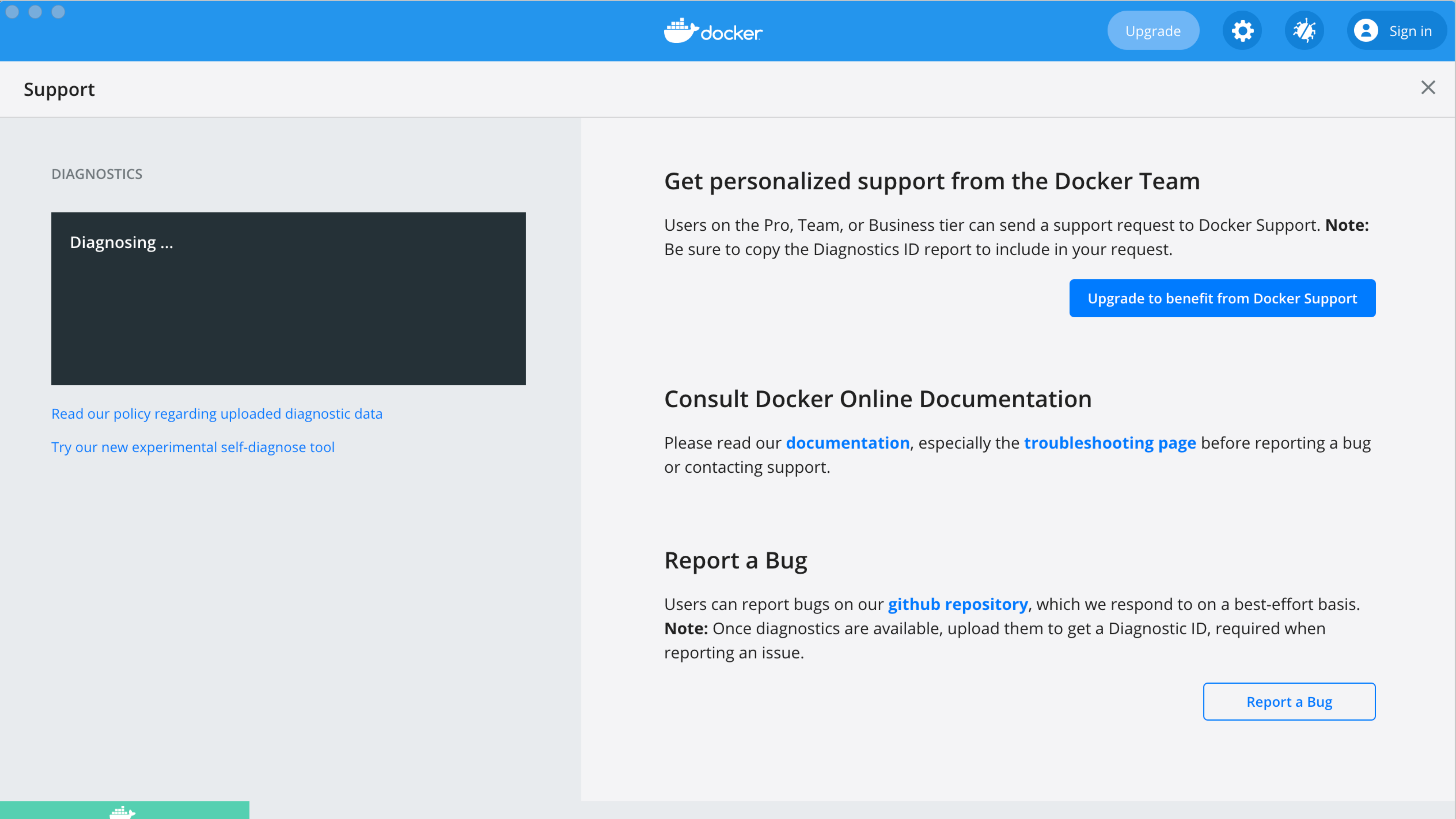
The Troubleshoot page contains the following options:

- **Restart Docker Desktop**: Select to restart Docker Desktop.
- **Support**: Users with a paid Docker subscription can use this option to send a support request. Other users can use this option to diagnose any issues in Docker Desktop.
- **Reset Kubernetes cluster**: Select this option to delete all stacks and Kubernetes resources.
- **Clean / Purge data**: Select this option to delete container and image data. Choose whether you'd like to delete data from Hyper-V, WSL 2, or Windows Containers and then click **Delete** to confirm.
- **Reset to factory defaults**: Choose this option to reset all options on Docker Desktop to their initial state, the same as when Docker Desktop was first installed.

Diagnose and feedback

- In-app diagnostics

1. Choose > Troubleshoot from the menu.
2. Sign into Docker Desktop. In addition, ensure you are signed into your Docker account.
3. Click Get support. This opens the in-app Support page and starts collecting the diagnostics.
4. When the diagnostics collection process is complete, click **Upload to get a Diagnostic ID**.
5. When the diagnostics have been uploaded, Docker Desktop prints a Diagnostic ID. Copy this ID.
6. If you have a paid Docker subscription, click **Contact Support**. This opens the *Docker Desktop support* form. Fill in the information required and add the ID you copied earlier to the Diagnostics ID field. Click **Submit** to request Docker Desktop support.



Support

DIAGNOSTICS

Diagnosing ...

Read our policy regarding uploaded diagnostic data

Try our new experimental self-diagnose tool

Get personalized support from the Docker Team

Users on the Pro, Team, or Business tier can send a support request to Docker Support. **Note:** Be sure to copy the Diagnostics ID report to include in your request.

Consult Docker Online Documentation

Please read our **documentation**, especially the **troubleshooting page** before reporting a bug or contacting support.

Report a Bug

Users can report bugs on our **github repository**, which we respond to on a best-effort basis. **Note:** Once diagnostics are available, upload them to get a Diagnostic ID, required when reporting an issue.

7. If you don’t have a paid Docker subscription, click **Upgrade to benefit from Docker Support** to upgrade your existing account.

   Alternatively, click **Report a Bug** to open a new Docker Desktop issue on GitHub. This opens Docker Desktop for Windows on GitHub in your web browser in a ‘New issue’ template. Complete the information required and ensure you add the diagnostic ID you copied earlier. Click **submit new issue** to create a new issue.

Diagnosing from the terminal

- On occasions it is useful to run the diagnostics yourself, for instance if Docker Desktop for Windows cannot start.
- First locate the com.docker.diagnose, that should be in C:\Program Files\Docker\Docker\resources\com.docker.diagnose.exe.
- To create and upload diagnostics in Powershell, run:
  - **PS C:\> & "C:\Program Files\Docker\Docker\resources\com.docker.diagnose.exe" gather –upload**
- After the diagnostics have finished, you should have the following output, containing your diagnostic ID:
  - **Diagnostics Bundle: C:\Users\User\AppData\Local\Temp\CD6CF862-9CBD-4007-9C2F-5FBE0572BBC2\20180720152545.zip**
  - **Diagnostics ID: CD6CF862-9CBD-4007-9C2F-5FBE0572BBC2/20180720152545 (uploaded)**

Self-diagnose tool

- Docker Desktop contains a self-diagnose tool which helps you to identify some common problems. Before you run the self-diagnose tool, locate com.docker.diagnose.exe. This is usually installed in
  - C:\Program Files\Docker\Docker\resources\com.docker.diagnose.exe.
- To run the self-diagnose tool in Powershell:
  - **PS C:\> & "C:\Program Files\Docker\Docker\resources\com.docker.diagnose.exe" check**
- The tool runs a suite of checks and displays PASS or FAIL next to each check. If there are any failures, it highlights the most relevant at the end.

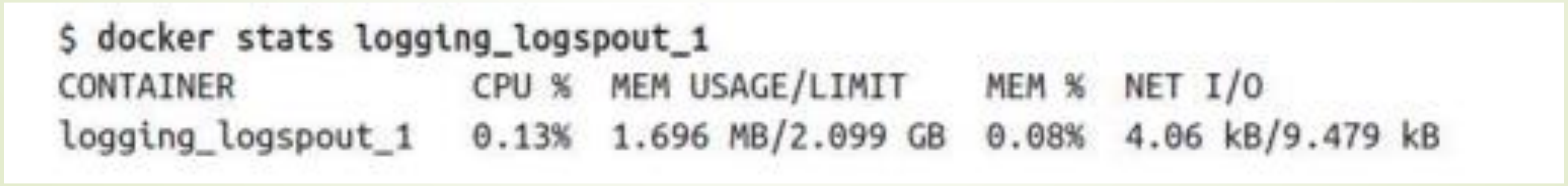
MONITORING AND ALERTING

- In a micro service system, you are likely to have dozens, possibly hundreds or thousands, of running containers.
- You are going to want as much help as you can get to monitor the state of running containers and the system in general.
- *A good monitoring solution should show at a glance the health of the system and give advance warning if resources are running low (e.g., disk space, CPU, memory).*
- *We also want to be alerted should things start going wrong (e.g., if requests start taking several seconds or more to process).*

MONITORING AND ALERTING

Monitoring with Docker Tools

- Docker comes with a basic CLI tool, *docker stats*, that returns a live stream of resource usage.
- The command takes the name of one or more containers and prints various statistics for them, in much the same way as the Unix application top.
- For example:

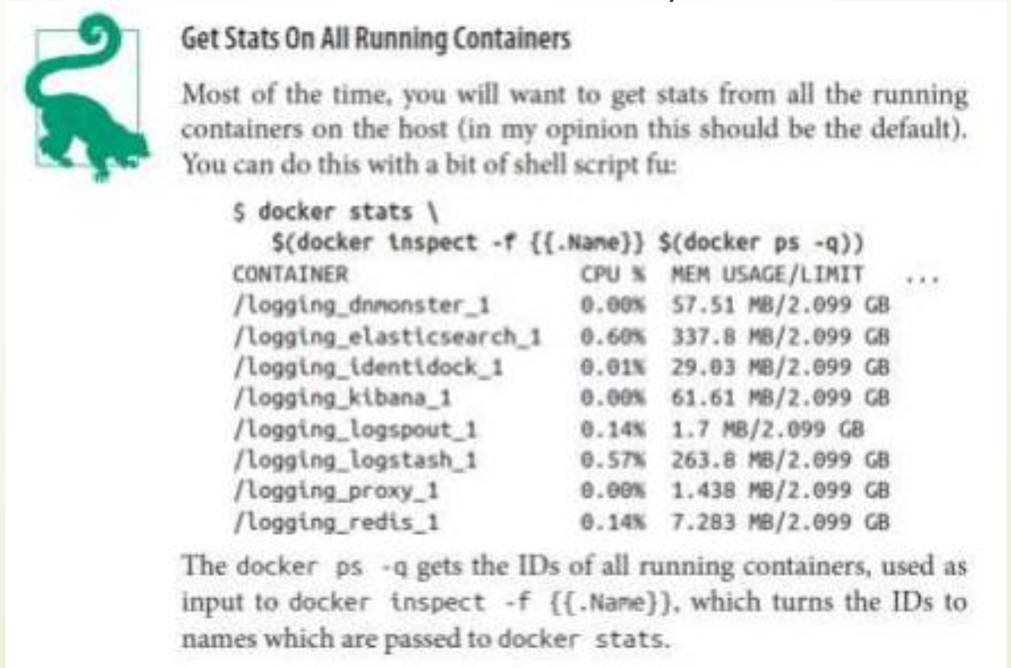


```
$ docker stats logging_logspout_1
CONTAINER            CPU %   MEM USAGE/LIMIT     MEM %   NET I/O
logging_logspout_1   0.13%   1.696 MB/2.099 GB   0.08%   4.06 kB/9.479 kB
```

MONITORING AND ALERTING

Monitoring with Docker Tools

- The *stats* cover CPU and memory usage as well as network utilization.
- Note that unless you have set memory limits on the container, the limit you see on memory will represent the total amount of memory on the host, rather than the amount of memory available to the container.



Get Stats On All Running Containers

Most of the time, you will want to get stats from all the running containers on the host (in my opinion this should be the default). You can do this with a bit of shell script fu:

```
$ docker stats \
    $(docker inspect -f {{.Name}} $(docker ps -q))
CONTAINER                   CPU %   MEM USAGE/LIMIT     ...
/logging_dnmonster_1        0.00%   57.51 MB/2.099 GB
/logging_elasticsearch_1    0.60%   337.8 MB/2.099 GB
/logging_identidock_1       0.01%   29.03 MB/2.099 GB
/logging_kibana_1           0.00%   61.61 MB/2.099 GB
/logging_logspout_1         0.14%   1.7 MB/2.099 GB
/logging_logstash_1         0.57%   263.8 MB/2.099 GB
/logging_proxy_1            0.00%   1.438 MB/2.099 GB
/logging_redis_1            0.14%   7.283 MB/2.099 GB
```

The `docker ps -q` gets the IDs of all running containers, used as input to `docker inspect -f {{.Name}}`, which turns the IDs to names which are passed to `docker stats`.

MONITORING AND ALERTING

Monitoring with Docker Tools

- This is useful in as far as it goes, and also hints at the existence of a *Docker API* that can be used to get such data programmatically.
- This API does indeed exist, and you can call the endpoint at /containers/<id>/stats to get a stream of various statistics on the container, with more detail than the CLI.
- This API is somewhat inflexible; you can stream updates for all values every second, or only pull all stats once, but there are no options to control frequency or filtering.
- This means you are likely to find the *stats API* incurs too much overhead for continuous monitoring but is still useful for ad hoc queries and investigations.
- Most of the various metrics exposed by Docker are also available directly from the Linux kernel, through the CGroups and namespaces features, which can be accessed by various libraries and tools, including Docker's runc library.

MONITORING AND ALERTING

Monitoring with Docker Tools

- If you have a specific metric you want to monitor, you can write an efficient solution using runc or making kernel calls directly.
- You will need to use a language that allows you to make low-level kernel calls, such as Go or C.
- Once you've exposed the values you need, you may want to look into tools such as statsd for aggregating and calculating metrics, InfluxDB and OpenTSDB for storage, and Graphite and Grafana for displaying the results.
- In the majority of cases, you will want to use a pre-existing tool for gathering and aggregating metrics and producing visualizations.
- There are many commercial solutions to this, but we will look at the leading open source and container-specific solutions.

MONITORING AND ALERTING

Monitoring with Docker Tools

- **Monitoring and Alerting with Logstash**
- While Logstash is very much a logging tool, it's worth pointing out that you can already achieve a level of monitoring with Logstash, and that the logs themselves are an important metric to monitor.
- For example, you could check nginx status codes and automatically email or message an alert upon receiving a high volume of 500s.
- Logstash also has output modules for many common monitoring solutions, including Nagios, Ganglia, and Graphite.

MONITORING AND ALERTING

Monitoring with Docker Tools

- **cAdvisor**
- cAdvisor (a contraction of Container Advisor) from Google is the most commonly used Docker monitoring tool.
- It provides a graphical overview of the resource usage and performance metrics of containers running on the host.
- As cAdvisor is available as a container itself, we can get it up and running in a flash.

⇒ Just launch the cAdvisor container with the following arguments:

```
$ docker run -d \
  --name cadvisor \
  -v /:/rootfs:ro \
  -v /var/run:/var/run:rw \
  -v /sys:/sys:ro \
  -v /var/lib/docker/:/var/lib/docker:ro \
  -p 8080:8080 \
  google/cadvisor:latest
```

⇒ Once the container is running, point your browser at http://localhost:8080. You should see a page with a bunch of graphs, something like Figure.

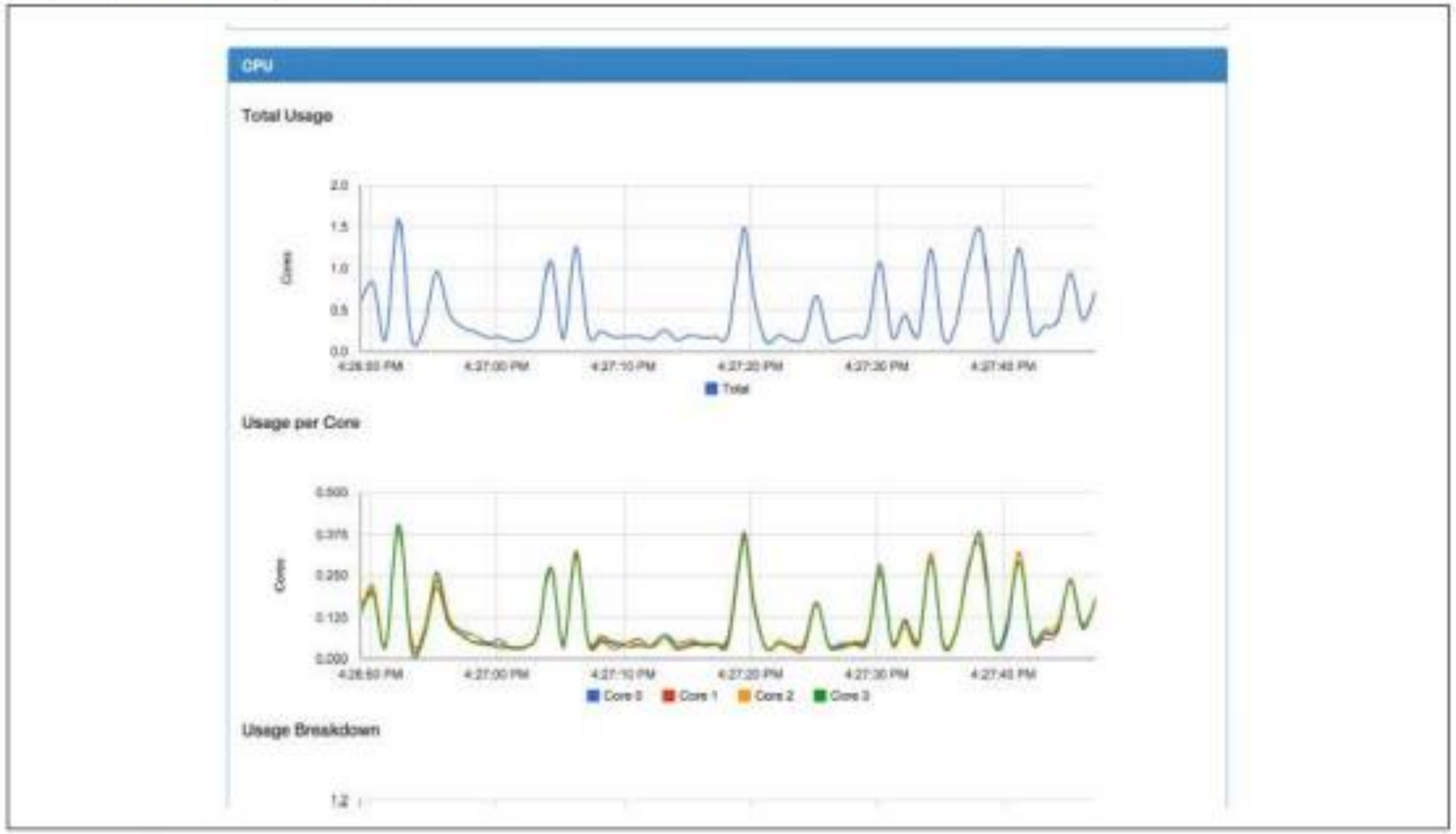


Figure 10-5. cAdvisor graph of CPU usage

MONITORING AND ALERTING

Monitoring with Docker Tools

- **cAdvisor**
- You can drill down to specific containers by clicking the “Docker Containers” link and then clicking the name of the container you’re interested in.
- cAdvisor aggregates and processes various stats and also makes these available through a REST API for further processing and storage.
- The data can also be exported to InfluxDB, a database designed for storing and querying time series data including metrics and analytics.
- The roadmap for cAdvisor includes features such as hints on how to improve and tune the performance of containers, and usage prediction information to cluster orchestration and scheduling tools.

MONITORING AND ALERTING

Monitoring with Docker Tools

- **Cluster Solutions**
- cAdvisor is great but is a per-host solution. If you're running a large system, you will want to get statistics on containers across all hosts as well on the hosts themselves.
- You will want to get stats on how groups of containers are doing, representing both subsystems as well as slices of functionality across instances.
- For example, you may want to look at the memory usage of all your nginx containers, or the CPU usage of a set of containers running a data analysis task.
- Since the required metrics tend to be application and problem specific, a good solution will provide you with a query language that can be used to construct new metrics and visualizations.

MONITORING AND ALERTING

Monitoring with Docker Tools

- **Cluster Solutions**
- Google has developed a cluster monitoring solution built on top of cAdvisor called Heapster, but at the time of writing, it only supports Kubernetes and CoreOS, so we won't consider it here.
- Instead, we'll take look at Prometheus, an open source cluster-monitoring solution from SoundCloud that can take input from a wide range of sources, including cAdvisor.
- It is designed to support large microservice architectures and is used by both SoundCloud and Docker Inc.

MONITORING AND ALERTING

Monitoring with Docker Tools

- **Prometheus**
- Prometheus is unusual in that it operates on a pull-based model.
- Applications are expected to expose metrics themselves, which are then pulled by the Prometheus server rather than sending metrics directly to Prometheus.
- The Prometheus UI can be used to query and graph data interactively, and the separate PromDash can be used to save graphs, charts, and gauges to a dashboard.
- Prometheus also has an Alert manager component that can aggregate and inhibit alerts and forward to notification services, such as email, and specialist services, such as PagerDuty and Pushover.

CONTROLLING RUNNING CONTAINERS

- In this section, let us introduce a few basic as well as a few advanced command structures for meticulously illustrating how the Docker containers can be managed.
- The Docker engine enables you to start, stop, and restart a container with a set of docker subcommands.
- Let's begin with the *docker stop* subcommand, which stops a running container.
- When a user issues this command, the Docker engine sends SIGTERM (-15) to the main process, which is running inside the container. The SIGTERM signal requests the process to terminate itself gracefully.
- Most of the processes would handle this signal and facilitate a graceful exit.
- However, if this process fails to do so, then the Docker engine will wait for a grace period.

CONTROLLING RUNNING CONTAINERS

- Even after the grace period, if the process has not been terminated, then the Docker engine will forcefully terminate the process.
- The forceful termination is achieved by sending SIGKILL (-9). The SIGKILL signal cannot be caught or ignored, and so it will result in an abrupt termination of the process without a proper clean-up.
- Now, let's launch our container and experiment with the docker stop subcommand, as shown here:

```
$ sudo docker run -i -t ubuntu:14.04 /bin/bash
root@da1c0f7daa2a:/#
```

CONTROLLING RUNNING CONTAINERS

- Having launched the container, let's run the *docker stop* subcommand on this container by using the container ID that was taken from the prompt.

```
$ sudo docker stop da1c0f7daa2a
da1c0f7daa2a
```

- Now, we will notice that the container is being terminated. If you observe a little more closely, you will also notice the text *exit* next to the container prompt.
- This has happened due to the SIGTERM handling mechanism of the bash shell, as shown here:

```
root@da1c0f7daa2a:/# exit
```

- If we take it one step further and run the *docker ps* subcommand, then we will not find this container anywhere in the list.
- Since our container is in the stopped state, it has been comfortably left out of the list.

CONTROLLING RUNNING CONTAINERS

How do we see the container that is in the stopped state?

- Well, the *docker ps* subcommand takes an additional argument *-a*, which will list all the containers in that Docker host irrespective of its status. This can be done by running the following command:

```
$ sudo docker ps -a
```

```
CONTAINER ID |IMAGE |COMMAND |CREATED |STATUS |PORTS |NAMES
da1c0f7daa2a |ubuntu:14.04 |"/bin/bash" |20 minutes ago |Exited (0) 10 minutes ago| |
desperate_engelbart
```

CONTROLLING RUNNING CONTAINERS

- Next, let's look at the *docker start* subcommand, which is used for starting one or more stopped containers.
- A container could be moved to the stopped state either by the *docker stop* subcommand or by terminating the main process in the container either normally or abnormally.
- On a running container, this subcommand has no effect.
- Let's start the previously stopped container by using the *docker start* subcommand, as follows:

```
$ sudo docker start da1c0f7daa2a
da1c0f7daa2a
```

CONTROLLING RUNNING CONTAINERS

- The *restart* command is a combination of the stop and the start functionality.
- In other words, the *restart* command will stop a running container by following the precise steps followed by the *docker stop* subcommand and then it will initiate the start process.
- This functionality will be executed by default through the *docker restart* subcommand.
- The next important set of container-controlling subcommands are *docker pause* and *docker unpause*.
- The *docker pause* subcommands will essentially freeze the execution of all the processes within that container.
- Conversely, the *docker unpause* subcommand will unfreeze the execution of all the processes within that container and resume the execution from the point where it was frozen.

CONTROLLING RUNNING CONTAINERS

- Having seen the technical explanation of pause and unpause, let's see a detailed example for illustrating how this feature works.
- We have used two screen or terminal scenarios.
- On one terminal, we have launched our container and used an infinite while loop for displaying the date and time, sleeping for 5 seconds, and then continuing the loop. We will run the following commands:

```
$ sudo docker run -i -t ubuntu:14.04 /bin/bash
root@c439077aa80a:/# while true; do date; sleep 5;
done
```

CONTROLLING RUNNING CONTAINERS

- Thu Oct 2 03:11:19 UTC 2014
- Thu Oct 2 03:11:24 UTC 2014
- Thu Oct 2 03:11:29 UTC 2014
- Thu Oct 2 03:11:34 UTC 2014
- Thu Oct 2 03:11:59 UTC 2014
- Thu Oct 2 03:12:04 UTC 2014
- Thu Oct 2 03:12:09 UTC 2014
- Thu Oct 2 03:12:14 UTC 2014
- Thu Oct 2 03:12:19 UTC 2014
- Thu Oct 2 03:12:24 UTC 2014
- Thu Oct 2 03:12:29 UTC 2014
- Thu Oct 2 03:12:34 UTC 2014

CONTROLLING RUNNING CONTAINERS

- Our little script has printed the date and time every 5 seconds with an exception at the following position:

```
Thu Oct 2 03:11:34 UTC 2014
Thu Oct 2 03:11:59 UTC 2014
```

- Here, we encountered a delay of 25 seconds, because this is when we initiated the docker pause subcommand on our container on the second terminal screen, as shown here:

```
$ sudo docker pause c439077aa80a
c439077aa80a
```

CONTROLLING RUNNING CONTAINERS

- When we paused our container, we looked at the process status by using the docker ps subcommand on our container, which was on the same screen, and it clearly indicated that the container had been paused, as shown in this command result:

```
$ sudo docker ps
CONTAINER ID |IMAGE |COMMAND |CREATED |STATUS |PORTS |NAMES
c439077aa80a |ubuntu:14.04 |"/bin/bash" |47 seconds ago |Up 46 seconds (Paused) ||ecstatic_torvalds
```

CONTROLLING RUNNING CONTAINERS

- We continued on to issuing the docker unpause subcommand, which unfroze our container, continued its execution, and then started printing the date and time, as we saw in the preceding command, shown here:

```
$ sudo docker unpause c439077aa80a
c439077aa80a
```

- The container and the script running within it had been stopped by using the docker stop subcommand, as shown here:

```
$ sudo docker stop c439077aa80a
c439077aa80a
```

CONTAINERS IN A BUSINESS CONTEXT

- Containers **allow software developers to package everything an application needs into its own, self-contained bundle**. Each bundle, or container, includes the application itself, plus any dependencies, libraries and configuration files it needs to run.
- Containers **let developers create applications using small services accessible through an API** rather than a monolithic application. When using this type of microservices architecture, developers can easily make a small change to an application and push it out immediately--without affecting other microservices.

CONTAINERS IN A BUSINESS CONTEXT

We will examine the reasons that businesses are turning to containers and how they benefit businesses.

Velocity

Containers are lightweight and immutable, which means they lend themselves to increasing the speed of software delivery. Containers package all the applications dependencies and configurations within the container image, which allows for the same image to be used across all environments without modification. This eliminates many inconsistencies and speeds up defect resolution. It also supports a more agile and DevOps oriented approach, improving the development, test, and production cycles of an application. The velocity of application delivery has evident benefit to the business as it supports the ability to deliver new value and capabilities to customers more quickly at scale.

CONTAINERS IN A BUSINESS CONTEXT

Portability

Software containers are a means for distribution. They allow applications to be run safely and confidently in multiple places. Containers are packaged and transferred to servers directly via registries using a simple tag, push, and pull model that's easily automated. OCI has brought on increased confidence and guarantees around compatibility thanks to their container image and runtime specifications. Unlike virtual machine images, container images can easily be migrated between clouds without a lot of rework. Portability and repeatability is a common struggle for software teams, and containers are a means to largely overcome that friction.

CONTAINERS IN A BUSINESS CONTEXT

Isolation

The nature of software containers is that they have limited impact on other applications running on the same node (node = single physical machine). This increases simplicity, security, and stability for all apps/services on the node. The lighter footprint that containers have compared to VMs allows you to potentially run thousands of containers on one node – all in isolation. Application density increases. From a business standpoint, this enables tremendous flexibility around infrastructure use and can dramatically reduce overall resource consumption.

CONTAINERS IN A BUSINESS CONTEXT

Availability

Since containers are more lightweight and their contents are designed to be ephemeral (meaning the critical data is stored outside the container then mounted as a volume), containers can be restarted quickly and seamlessly if your application allows for this. If a container fails to start properly or stops responding, a new instance of the container can be scheduled by an orchestrator, helping to ensure high-availability. The ability to ship containers between different clouds and infrastructure providers can also be a factor in maintaining constant uptime.

CONTAINERS IN A BUSINESS CONTEXT

Simplicity

The container packaging model aligns well with modern, distributed application architectures that consist of different microservices. Once past the learning curve, one should work to decompose existing apps and package them more simply as immutable images. This pattern is proven to streamline operations via reduction of onerous tasks such as operating system stabilization, runtime provisioning, and other configuration. Deployment of an individual application or service is complete, easily repeatable, and fast.

CONTAINERS IN A BUSINESS CONTEXT

The overall benefits

The combined benefits of containers enable an accelerated pace of software delivery, which at the same time lowers the cost of development and operations. Businesses are leveraging containers to build new cloud-native applications, re-architect existing applications, and as a lift-and-shift strategy to move COTS or legacy applications to the cloud. Containers are used everywhere: private data centers, public cloud, and are commonly used to enable seamless portability between environments.

As the number of organizations using containers continues to grow, there is a movement towards management of containerized infrastructure and applications at a strategic level that’s growing even faster.