User Guide

Using Payara Micro with Docker

The Payara® Platform - Production-Ready,
Cloud Native and Aggressively Compatible.

Contents

Docker® is an open-source tool used to create, deploy and manage small portable containers. Containers are similar to virtual machines (VM), but where VMs run an entire operating system inside, the host containers only package the required components. Because of this they are extremely lightweight; they start up in a fraction of the time of a regular VM and waste very little extra resources on top of the main process being run inside them. They are used primarily as a lightweight way to assure that the program will run the same regardless of host platform. Docker can also manage virtual networking between containers, as well as health checks to make sure each container is running properly.

Payara provides several Docker container images. These can be used as-is to run your applications on Payara Server or Payara Micro (the Payara Platform). Or you can create your own Docker images based on them. This guide will demonstrate the basic usage of Docker, as well as some example configurations using the Payara Micro Docker images.

At the time of writing this blog, the most recent Payara Micro release is 5.191. Some functionality may differ if you're using a different version.

Get Started with Docker

Before explaining how to use Payara Micro with Docker, we'll explain what Docker is and how to use it in most common scenarios.

Here's some terminology we'll be using further:

Image An image is a snapshot of how a container should look like before it starts up. It will define which programs are installed, what the startup program will be, and which ports will be exposed.

Container A container is an image at runtime. It will be running the process and the configuration defined by the image, although the configuration may differ from the image if any new commands have been run inside the container since startup.

Docker Daemon The Docker daemon is the service that runs on your host operating system. Containers are run from the Docker daemon. The Docker CLI (command line interface) just interacts with this process.

DockerHub DockerHub is a central repository where images can be uploaded, comparable to what Maven Central is for Maven artifacts. When pulling an image remotely, it will be pulled from DockerHub if no other repository is specified.

Repository A repository in the context of Docker is a collection of Docker images. Different images in the repository are labelled using tags.

Tag A tag distinguishes multiple images within a repository from one another. Often these tags correspond to specific versions. Because of this, when no tag is specified when running an image, the 'latest' tag is assumed.

Entrypoint The Entrypoint command is run when the container starts. Containers will exit as soon as the entrypoint process terminates.

Installation of Docker

To run Docker containers, Docker must first be installed on your operating system. To install Docker, you can follow the OS specific guide from the Docker documentation[1].

In case your system is Microsoft Windows® or Mac®, download and run the installation program for Docker Desktop according to the instructions.

In case your system is Linux®:

1. Install the Docker program according to the instructions for your Linux® distribution.
2. Start the Docker daemon - this is specific to your Linux distribution. On Ubuntu®, you would do it by `sudo systemctl start docker`.
3. By following the above steps, you will have Docker setup and ready to run. If however you want to perform steps further to the basic installation such as: enabling Docker on boot, running Docker without `sudo`, or allowing remote connections to your Docker daemon, then the post-install guide on the Docker website[2] details how to do these steps and more.
    - After installing Docker on Linux, you need to run all docker commands with `sudo`, e.g. `sudo docker info`. To run Docker without sudo, add your user into a group called `docker` and restart your computer.

Once you've got Docker installed, you can run the following command to verify your install (on Linux, you might need to prepend the command with `sudo`):

```
docker run hello-world
```

This will run the 'hello-world' Docker container in the foreground, which is a minimal image with a C program to print the message it does. Since the container then exits immediately, it won't block the terminal.

Using Docker Containers

The following sections will detail the basic container management required to use Docker. The full functionality is covered in the command guide on the Docker Website[3].

Checking Container Status

Running the following command will show all currently running containers:

```
docker ps
```

You'll see an output similar to the following:

```
CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS                      PORTS   NAMES
3dc34ab42b5c   hello-world   "/hello"   16 minutes ago   Exited (0) 16 minutes ago           gifted_noyce
```

Any container can be referenced in commands either by its ID directly, or one of its aliases listed on the right. If none were specified on the command line, it'll be assigned a random one.

If you don't see anything from here, it means that no containers are running. If you expect to see a process here but you don't, it likely means it errored out and stopped, since Docker containers terminate as soon as the entrypoint program exits. For scenarios like this, the Docker daemon usefully keeps containers stored after they've closed. You can run the following command to see all containers, even after they've finished:

```
docker ps -a
```

This will also list stopped containers. To use a container with the same alias, these stopped containers must be removed. See the Docker `rm` command later for details on how to do this.

Starting a Container

Using the hello-world image as a start point, we'll start a container:

```
docker run hello-world
```

This runs the 'hello-world' container, assuming the latest tag. You can specify a custom tag by appending the tag name after the image name, separating them with a colon. The container will run in the foreground but since the process exits almost immediately it won't block the terminal. If the image isn't found locally, it will be downloaded from DockerHub (as with the docker pull command). The following parameters are common when running containers:

Parameter Name	Parameter Value	Description
--rm	N/A	Denotes that once this container has finished running, it should be automatically removed from the list of closed processes, freeing up the name for reuse.
--name	Container name	Specifies a name for the container once running. This will replace the automatically assigned name e.g. gifted_noyce. These names are unique identifiers, and as such cannot be shared between containers, even stopped ones.
-p	<host_port>:<container_port>	Specifies a host port to map to a specific port on the container. For example, running HTTPD (which hosts an Apache HTTPD instance on port 80) with the parameter `-p 9000:80` will expose the HTTPD server on port 9000 of the host machine.
-d	N/A	Without this parameter the container is run in the foreground. It will block the terminal process, and print the container logs to the terminal. With this option, the process is forked to the background, and will simply print the container ID instead of the container logs.
-v	<local_dir>:<container_dir>	Allows a container to use the local filesystem, by mounting the specified local directory to the specified container directory.

For a full list of parameters, see the official documentation[4].

Basic Networking

When running a container, the -p option explained above maps a port on the host to a port on the container being run. If no port mappings are specified the container is still accessible, but only from the host running the Docker daemon. Docker handles a collection of networks; the default one is named 'bridge', and will allow containers running on the same machine to communicate. You can inspect this network by running the following command:

```
docker network inspect bridge
```

This will print out the details of the bridge network, and within that the IPs of containers running on it. You can read more about Docker networking here: https://docs.docker.com/network/.

Stopping a Container

When you've got a container running and it's visible in the output of `docker ps`, you can stop the container using the following syntax:

```
docker stop <container-id/alias>
```

This command will send a SIGTERM to the main process running inside the container in order to terminate it. If the main process doesn't handle SIGTERM signals properly, either because it's hanging, or it hasn't been designed with that in mind, then a SIGKILL will be sent after a grace period (default 10 seconds) if the container hasn't terminated.

If you need to forcibly stop a container, you can run the following command instead:

```
docker kill <container-id/alias>
```

This command kills the container immediately with a SIGKILL. You can change the signal sent by this command with the --signal option.

Removing Old Containers

If you try and run a container with the same name twice, you'll get an exception that looks similar to the following:

```
docker: Error response from daemon: Conflict. The
container name "/hello" is already in use by container
"f122e1365f9b545a034676c2764de4bad431466866a6295b6ba8cd1965704b5a". You have to
remove (or rename) that container to be able to reuse that name.
See 'docker run --help'.
```

To fix this, the old container (that appears with docker ps -a) will need removing. You can remove this container using it's name or any aliases assigned to it like so:

```
docker rm <image-name>
```

Checking Container Logs

When a container is either running in the background (it has been forked with `docker run -d`), or it has exited early, it can be useful to see the container logs.

```
docker logs <container-id/alias>
```

This will print out the current content of the container logs to the terminal. To follow these logs, add the -f parameter. To see timestamps with the logs, use the -t parameter.

Executing Commands In Containers

When a container is running, you can start an interactive shell inside the container to execute commands at runtime. This is useful if for example, you have an image that isn't quite right and you want to test exactly what commands need to be run next in the Dockerfile. Alternatively, you can commit the current status of a container to a new named image. For more details on this, see the documentation for docker commit[5].

```
docker exec -it <container-id/alias> <command>
```

That will execute the given command on the given container. The -it options are present if you specify the command as being `sh` for example. This makes the command interactive, which allows an interactive shell process to be run inside the container.

Writing Docker Images

Docker images are written using Dockerfiles. These are text files that describe the building blocks for any image. They support a simple set of commands that will be run in order to build the image. Each Dockerfile is built starting from a base image (which is any other image), and each command in the Dockerfile will create a new image layer. Because of this image layering, downloading new images is really quick since the same previous layers can be reused. For example, if you download multiple images built from the Ubuntu image, the Ubuntu image will only be downloaded once. This

section will cover the basics of Dockerfile authoring. For more information on this, see the Docker official documentation[6].

Below is an example of a Dockerfile:

```
FROM alpine:3.8
ENV test world
CMD echo "Hello $test!"
```

This image is built from the Alpine image (a minimal Linux distribution made for Docker), and will print “Hello world!” when run. Dockerfiles will be written in the same format as above, using the Dockerfile command specified at the beginning of any line. Some basic commands are covered below.

FROM

The FROM command specifies a base image for your custom image. As such it should usually be the first command in your Dockerfile. Each image is expected to build from another, unless you want to build every image you write from scratch[7] (a 0 byte image that doesn’t even contain a filesystem). The FROM command follows the following syntax:

```
FROM <image>[:<tag>]
```

If no tag is specified, the ‘latest’ tag is assumed. So for example for the Alpine image, FROM alpine:3.8 will run the version 3.8 image, and FROM alpine will use the latest tag, which could be equal to another tag.

ENV

The ENV command sets an environment variable <key>=<value>. These environment variables will then be available in all future layers. This command can use either of the following syntax:

```
ENV <key> <value>
ENV <key>=<value> ...
```

RUN

The RUN command is used to run commands on the intermediary image. This is used in setup to run commands that can be run before the container needs to start, for example creating directories or running CURL to fetch an online resource.

The RUN command, along with some other commands such as ENTRYPOINT or CMD, accept two forms of syntax since they are used to pass commands to the container. These forms are referred to as exec form and shell form. The differences are covered below:

Shell Format

The shell format looks as follows:

```
RUN executable param1 param2
```

This format will invoke a shell to run the command. This means that the image must contain a shell to run the command, and that if you do, environment variable substitution will be performed.

Exec Format

The exec format looks as follows:

```
RUN ["executable", "param1", "param2"]
```

This format will run the executable directly. No variable substitution will be performed, which can sometimes be useful in preventing strings being changed by the shell.

The RUN command can use either of these forms. When running in shell form, a backslash must be used to continue the RUN instruction onto the next line. To reduce the number of layers produced it's recommended to run as many RUN commands as possible in the same layer, for example like so:

```
RUN mkdir -p /opt/test/config && \
    mkdir -p /opt/test/lib
```

There is no functional reason to have a layer containing only one of these commands, so they are merged to reduce layer complexity.

ADD

The ADD command is used to add a file from a local directory or remote URL to the fileystem of an image, so that it can be used at runtime.

```
ADD [--chown=<user>:<group>] <src> <dest>
```

If a local file in a recognised compression format is used, it is automatically unpacked as a directory. If the chown parameter is used, the file will be owned by the specified user at runtime.

While the ADD command can specify a remote URL as a source, this is discouraged in favour of doing so from a RUN command:

```
RUN curl -Sl http://test.com/big.tar.gz && \
    tar -xzC /big.tar.gz && \
    rm -f /big.tar.gz
```

This is because the files not needed after extraction can be deleted as shown above.

Because of the automatic unpacking of the ADD command, the COPY command is recommended when this is not required. The COPY command functions in the same way as ADD, but without support for remote files or compressed file unpacking.

ENTRYPOINT

The ENTRYPOINT command specifies the executable to run on container start. It supports the same exec and shell forms supported by the run command. When used in shell form (not recommended), the ENTRYPOINT is run using /bin/sh -c. Because of this, signals will not be passed to the subprocess. **This means that your application will not respond to docker stop.** There is no default ENTRYPOINT, meaning that the CMD is used as the entrypoint unless one is specified.

CMD

The CMD command also specifies a command to run on container startup, but behaves depending on how it's specified. The CMD command has 3 forms. The first two are the same shell and exec forms as ENTRYPOINT and RUN. The third form is when the exec form is while the ENTRYPOINT is specified in exec form. In this use case, the CMD command will specify default parameters to the ENTRYPOINT process. In all other cases, the CMD command sets the command to be executed on container start.

Building a DockerFile

Once you've got a Dockerfile written, you must build the image to run it. Building the image will incrementally build images by running the commands contained in the Dockerfile. You can build an image using the following command:

```
docker build [OPTIONS] PATH
```

To see the full syntax, see the official reference page[8]. An example usage would be building an image called 'test' from the directory containing the Dockerfile using the following command:

```
docker build -t test .
```

The command will look for the default Dockerfile called 'Dockerfile' by default. You can then run the container as with a remote image with docker run.

Payara Micro Docker Image

Payara provides a Docker image for Payara Micro which is tuned for production usage but is also very useful in development. It's pre-configured in a way that makes it easy to start and use without a custom Dockerfile. You can also build your own images using the Payara Micro image as a base to remove several required steps from your Dockerfile.

The Payara Micro Docker image can be found on DockerHub: payara/micro[9]

Using the Payara Micro Docker Image

The Payara Micro Docker Image can be found on DockerHub, see here for more details: https://hub.docker.com/r/payara/micro/.

A container created from the Payara Micro Docker image will run a Payara Micro instance as the main process. You can run a container with no configuration changes using the following command:

```
docker run -p 8080:8080 payara/micro
```

The 'latest' tag is used by default.

This will simply run Payara Micro inside the container, with no applications deployed, and map port 8080 to the host machine. You can visit localhost:8080 and see the request forwarded to the container.

Deploying Applications

f you don't want to extend the Dockerfile with your own, the Payara Micro Docker image provides a directory from which applications will be deployed on startup. To utilise this, mount a directory using the following command:

```
docker run -p 8080:8080 -v /local/application/directory:/opt/payara/deployments
payara/micro
```

The default entrypoint for the container will deploy all deployment files and directories found in /opt/payara/deployments, so mounting a directory there that contains an application will deploy it on startup.

Alternatively, you can create your own Dockerfile to do the same:

```
FROM payara/micro
COPY application.war $DEPLOY_DIR
```

Running the container produced from this image will also deploy the application on startup. The image will need rebuilding when the application changes.

The Container Entrypoint

The Payara Micro Docker image runs Java as PID 1, with an entrypoint of the Payara Micro jar. By default, the CMD arguments to the entrypoint are the deployment directory. This can be overridden from the command line or using the CMD instruction in Dockerfile, but the deployment directory would need to be specified again. For example:

```
docker run payara/micro --nocluster --deploymentDir /opt/payara/deployments
```

The same thing in Dockerfile:

```
FROM payara/micro
CMD ["--nocluster", "--deploymentDir", "/opt/payara/deployments"]
```

This will run Payara Micro without starting Hazelcast®. See the Payara Micro documentation[10] for a list of all possible arguments.

Configuring the Root Directory in the Image

The standard Docker image will extract its root configuration directory into a temporary directory before the server starts and then begins configuring itself by interpreting the command line option. It is possible to 'bake' these changes into the Docker image itself by running Payara Micro with command line switch --warmup during image build. The switch will cause the instance to shutdown as soon as it finishes its startup, which includes executing pre-boot and post-boot scripts and deployment of all applications and executing post-deploy scripts.

This can be used for variety of use cases, including:

- Preparing resources like database connection pools separately from deployment, simplifying number of command line arguments necessary for productive run
- Preparing Docker image with applications already deployed into runtime directory
- Collecting class loading information for preparing class data sharing archive

In Dockerfile it will look similar to this:

```
FROM payara/micro

# Copy prepared scripts to configure the instance
COPY scripts/set-thread-pools .
COPY target/app.war .

RUN java -jar payara-micro.jar --rootdir rootdir/ --postbootcommandfile set-
thread-pools --deploy app.war:/ --warmup \
# Files are already applied and present in root directory
    && rm set-thread-pools app.war
# rootdir needs to be specified so that configuration is picked up
CMD ["--rootdir", "rootdir/"]
```

Launching Micro with flat classpath

In order to utilize JVMs class sharing option on OpenJDK-based virtual machines, Payara Micro needs to launch with simpler classloading configuration. This is achieved by creating special launcher entrypoint in root configuration directory.

```
FROM payara/micro

FROM payara/micro

# Copy prepared scripts to configure the instance
COPY scripts/set-thread-pools .
COPY target/app.war .

# Creates rootdir/launch-micro.jar
RUN java -jar payara-micro.jar --rootdir rootdir/ --outputlauncher \
 # the original launcher is no longer necessary
 && rm payara-micro.jar  \
 && java -jar rootdir/launch-micro.jar --postbootcommandfile set-thread-pools
--deploy app.war:/ --warmup \
 && rm set-thread-pools app.war

ENTRYPOINT [“java”, “-XX:+UseContainerSupport”, “-XX:MaxRAMPercentage=90.0”,
“-jar”, “rootdir/launch-micro.jar”]
# rootdir is hardcoded in launcher
CMD [“--nocluster”]
```

There are few more steps involved in enabling class data sharing , which are explained in blog article Warming Up Payara Micro Container Images[11].

Clustering Payara Micro Docker Containers

One method of connecting multiple Payara Micro Docker containers into a cluster is using TCP/IP Hazelcast discovery. This will allow multiple Payara Micro Docker containers to join the same data grid, although they will not be able to change the configuration of each other or join deployment groups etc. This method therefore relies on the configuration and application being contained entirely in your Docker image.

To cluster using this method, first find the IPs that will be used by your Docker containers. This is explained in the earlier section ‘Basic Networking’. Knowing this, you must enable TCP/IP discovery on the instances. This can be done either using your Dockerfile, or from the command line making use of the CMD overwriting. Assuming that your Docker containers are assigned IPs of between 172.17.0.2 and up, the following command will run Payara Micro Docker containers that cluster in the aforementioned fashion.

```
docker run payara/micro --clusterMode tcpip:172.17.0.1-50:6900
```

Note that this can also be configured in a Dockerfile by overwriting the CMD command.

Payara Micro also support DNS and Kubernetes® discovery modes which will be more suitable for docker-compose and Kubernetes environment respectively. More information on discovery modes can be found here: https://docs.payara.fish/documentation/payara-micro/clustering/autoclustering.html.

References

1. https://docs.docker.com/install/
2. https://docs.docker.com/install/linux/linux-postinstall/
3. https://docs.docker.com/engine/reference/commandline/docker/
4. https://docs.docker.com/engine/reference/commandline/run/
5. https://docs.docker.com/engine/reference/commandline/commit/
6. https://docs.docker.com/engine/reference/builder/
7. https://hub.docker.com/_/scratch
8. https://docs.docker.com/engine/reference/commandline/build/
9. https://hub.docker.com/r/payara/micro/
10. https://docs.payara.fish/documentation/payara-micro/appendices/cmd-line-opts.html
11. https://blog.payara.fish/warming-up-payara-micro-container-images-in-5.201

sales@payara.fish **+44 207 754 0481** **www.payara.fish**