User Guide

Using Payara Server 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 that 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 the provided Payara Docker container images. This guide will demonstrate the basic usage of Docker, as well as some example configurations using the Payara Server Docker images.

At the time of writing this guide, the most recent Payara Server 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 Server with Docker, we'll explain what Docker is and how to use in most common scenarios.

Here's some terminology we'll be using:

Image An image is a snapshot of how a container should look 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 <container-id/alias>
```

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>
```

The command above 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 Server Docker Images

The Payara Platform provides the following distributions as Docker container images for both Zulu JDK-8 and Zulu JDK-11 that can be used as-is to run your applications on the Payara Platform:

- Payara Server Full
- Payara Micro
- Payara Server Node
- Payara Server Web

The Payara Server Web image is the same as the Full image but using Payara Server Web distribution. Instructions will be the same for both. The Payara Server Node image is a server-full image customised for use as an instance in a bigger domain, connecting to and registering itself against a separate DAS. Configuration instructions for this image are different to those of the server-full and server-web images.

You can also create your own Docker images based on the provided Payara Docker container images.

Docker images for Payara Platform Community Edition are available on DockerHub, while customers can access Payara Enterprise Docker Images in our Nexus repository under the repo 'payara-docker'. Instructions for using the private Payara Enterprise Docker repo are available in the Customer Hub Support Portal.

Using the Payara Server Full Docker Image

The Payara Server Full Docker Image can be found on DockerHub, see here for more details: https://hub.docker.com/r/payara/server-full/.

A container created from the Payara Server Full Docker image will run the Payara Server **production** domain as the main process. You can run the container with no configuration changes using the following command:

```
docker run -p 8080:8080 payara/server-full
```

The 'latest' tag is used by default.

This will simply run Payara Server inside the container, and map port 8080 to the host machine. You can visit localhost:8080 and see the request forwarded to the container. To visit the admin console on a localhost port you would also need to map port 4848. To see how to access Payara Server, see the networking section in the Docker container introduction above.

When using the administration console or asadmin utility, the default username and password are both 'admin'.

> ***When you run the Payara Server docker container with the default settings, Payara Server will assume all host resources are available to it and it will allocate all the available memory. To restrict how much memory and cpu can be used by the docker container, use the options --cpus (nnumber of CPUs allowed to use) and -m (amount of memory allowed to use), for example like this:***
>
> ```
> docker run --cpus=1.5 -m=1024m -p 8080:8080 payara/server-full
> ```

Deploying Applications

You can deploy via the admin console as with a non-containerized Payara Server instance (an instance not running in a Docker container). If you don't want to extend the Dockerfile with your own, the Payara Server Full Docker image also 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/server-full
```

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

Alternatively, you can build your own Docker image from a Dockerfile to do the same:

```
FROM payara/server-full
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 Lifecycle

With the default Docker entrypoint, Payara Server Docker container will do the following during start-up:

1. Search for a file with Payara Server commands stored in the file `$POSTBOOT_COMMANDS` (by default `/opt/payara/config/post-boot-commands.asadmin`)
   - if the file exists and contains commands, these commands will be executed when Payara Server starts up
   - you can copy a file with commands into this location when building a custom docker image
2. Search for a file with Payara Server commands stored in the file `$PREBOOT_COMMANDS` (by default /opt/payara/config/pre-boot-commands.asadmin)
   - if the file exists and contains commands, these commands will be executed as Payara Server starts up
   - you can copy a file with commands into this location when building a custom docker image
3. Search for all deployable packages in the directory `$DEPLOY_DIR` and generate a script with commands to deploy them when Payara Server boots
   - `DEPLOY_DIR` is an environment variable which can be modified to search for deployables in a different directory
   - all generated deploy commands are appended to the file `$POSTBOOT_COMMANDS` if it exists
4. If shell scripts with suffix `.sh` found in `$SCRIPT_DIR/init.d` directory (by default `/opt/payara/scripts/init.d`), they are executed
   - you can copy your own scripts into that directory when building a custom docker image or you can mount the directory to your local directory with scripts
5. Payara Server starts and executes the `$PREBOOT_COMMANDS` and `$POSTBOOT_COMMANDS` scripts, which by default contains commands to deploy found applications and any custom commands
   - Payara Server process listens to all signals sent to the Docker container and correctly shuts down when you stop the container with `docker stop`

So if you wanted applications to be deployed as usual but the container IP to be printed out initially, you might create a bash script (test.sh) with the following contents:

```
#!/bin/bash
ip a
```

then run the container as follows:

```
docker run -v /directory/with/script:/opt/payara/scripts/init.d payara/server-full
```

This would cause the container IP details to be printed out before starting the server.

Environment Variables

The following environment variables are available for use. When edited either in a `Dockerfile` or before the `startInForeground.sh` script is ran, they will change the behaviour of the Payara Server instance.

- `JVM_ARGS` - Specifies a list of JVM arguments which will be passed to Payara Server in the `startInForeground.sh` script.
- `MEM_MAX_RAM_PERCENTAGE` - Specifies the maximum percentage of container RAM that Payara Server will be able to use.
- `MEM_XSS` - Specifies the size of the JVM Thread Stack Size.
- `DEPLOY_PROPS` - Specifies a list of properties to be passed with the deploy commands generated in the `generate_deploy_commands.sh` script, For example '`--properties=implicitCdiEnabled=false`'.
- `POSTBOOT_COMMANDS` - The name of the file containing post boot commands for the Payara Server instance. This is the file written to in the `generate_deploy_commands.sh` script.
- `PREBOOT_COMMANDS` - The name of the file containing pre boot commands for the Payara Server instance.

The following environment variables shouldn't be changed, but may be helpful in your Dockerfile.

Variable Name	Value	Description
HOME_DIR	/opt/payara	The home directory for the payara user.
PAYARA_DIR	/opt/payara/appserver	The root directory of the Payara installation.
SCRIPT_DIR	/opt/payara/scripts	The directory where the generate_deploy_commands.sh and startInForeground.sh scripts can be found.
CONFIG_DIR	/opt/payara/config	The directory where the post and pre boot files are generated to by default.
DEPLOY_DIR	/opt/payara/deployments	The directory where applications are searched for in generate_deploy_commands.sh script.
DOMAIN_NAME	production	The name of the Payara Server domain.
PASSWORD_FILE	/opt/payara/passwordFile	The location of the password file for asadmin. This can be passed to asadmin using the --passwordfile parameter.

If you want to change the admin password instead of using the default one, it's not enough to change the password stored in the file **$PASSWORD_FILE**. *You also need to use the following command to change the password before starting Payara Server:*

```
${PAYARA_DIR}/bin/asadmin --user ${ADMIN_USER} --passwordfile=/
mypasswordfile change-admin-password --domain_name=${DOMAIN_NAME}
```

And you also need to supply the file **/mypasswordfile** *with the current admin password (in the* **AS_ADMIN_PASSWORD** *variable) and the new password (in the* **AS_ADMIN_NEWPASSWORD** *variable). Then you need to change the password in the file* **$PASSWORD_FILE** *so that Payara Server starts with the correct password.*

You can add this command in your custom Docker image (using the **RUN** *Dockerfile command) or you can place it in a shell script in the* **init.d** *directory inside the Docker container.*

Payara Server also allows you to externalise various aspects of configuration through the usage of variable replacement, utilising environment variables, system properties, and MicroProfile Config

properties. This allows you to, for example, extend the image with additional environment variables and reference them within your application like so:

```
@DataSourceDefinition(name="java:global/ExampleDataSource",
      className="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource",
      user="${ENV=mysql.user}",
      password="${ENV=mysql-db-password}",
      databaseName="${ENV=MYSQL_DBNAME}",
      serverName="localhost"
   )
```

Running Commands

If you need to modify the configuration of Payara Server in your custom Docker images, you can do it by adding asadmin commands into the following files:

- `$POSTBOOT_COMMANDS` (by default /opt/payara/config/post-boot-commands.asadmin) - these commands will be executed after Payara Server is started and ready to accept requests but before applications are deployed
- `$PREBOOT_COMMANDS` (by default /opt/payara/config/pre-boot-commands.asadmin) - these commands will be executed before Payara Server engine is started. This mode usually works only for the set asadmin command to modify the configuration using dotted configuration names

For example, the following Dockerfile would instruct Payara Server to enable the HealthCheck Service after the core server is booted:

Dockerfile

```
FROM payara/server-full

RUN echo 'healthcheck-configure --dynamic=true --enabled=true' > $POSTBOOT_
COMMANDS
```

The Default Entrypoint

If a JVM runs as PID 1 it will not reap zombie processes correctly. Because of this, the Payara Docker image uses Tini as an init: https://github.com/krallin/tini. This is a simple init script that responds to signals and reaps zombie processes correctly. The default `CMD` argument for `tini` runs the `${SCRIPT_DIR}/entrypoint.sh` (default SCRIPT_DIR is /opt/payara/scripts) script in *exec* mode, which in turn runs the following:

- `${SCRIPT_DIR}/init_1_generate_deploy_commands.sh`. This script outputs deploy commands to the post boot command file located at `$POSTBOOT_COMMANDS` (default `$CONFIG_DIR/post-boot-commands.asadmin`). If the deploy commands are already found in that file, this script does nothing.
- `${SCRIPT_DIR}/init_*.sh` scripts that you may provide for custom use as waiting or initializing during startup, before Payara starts up.
- `${SCRIPT_DIR}/init.d/*.sh` scripts that function the same as the above scripts, but is initially an empty directory to allow volume mounting to provide these scripts.
- `${SCRIPT_DIR}/startInForeground.sh`. This script starts the server in the foreground, in a manner that allows the Payara instance to be controlled by the docker host. The server will run the pre boot commands found in the file at $PREBOOT_COMMANDS, as well as the post boot commands found in the file at `$POSTBOOT_COMMANDS`.

Under usual circumstances, you shouldn't modify the entrypoint or the CMD argument for a Payara Docker container. Instead, specify additional shell scripts in the init.d directory or Payara Server commands in the $POSTBOOT_COMMANDS file, or modify provided environment variables.

Clustering Payara Server Docker Containers

One method of connecting multiple Payara Server Docker containers into a cluster is using TCP/IP Hazelcast® discovery. This will allow multiple Payara Server 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 an init script dropped in /opt/payara/scripts/init.d. Assuming that your Docker containers are assigned IPs of between 172.17.0.2 and up, the following Dockerfile will produce containers that cluster in the aforementioned fashion:

```
FROM payara/server-full:5.184

RUN echo 'set configs.config.server-config.hazelcast-config-specific-
configuration.enabled=false' > $PREBOOT_COMMANDS
RUN echo 'set-hazelcast-configuration --clusterMode tcpip --tcpipmembers
172.17.0.1-99:4900' > $POSTBOOT_COMMANDS
```

First, in order to avoid starting Hazelcast with the default configuration, Hazelcast is disabled by executing the `set` command before server is booted. Then, after the server is booted, Hazelcast data grid is enabled and configured to use the `tcpip` cluster mode.

Payara Server 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-server/hazelcast/discovery.html.

For more details on all methods of clustering Payara Server Docker containers, see our guide *Clustering Payara Server in Docker*.

Using the Payara Server Node Docker Image

The Payara Server Node Docker Image can be found on DockerHub, see here for more details: https://hub.docker.com/r/payara/server-node

A container created from the Payara Server Node Docker image will create and start an Payara Server full instance as the main process, registering itself to a separate Payara Server DAS (containerised or non-containerised). Containers using this image can either be started manually via the Docker CLI or from the Payara Server DAS, and can be used in one of two ways: as a managed container or as an unmanaged container.

Regardless of using containers based on this image in a managed or unmanaged capacity, this image requires *secure administration* to be enabled:

Enable Secure Admin

```
> asadmin change-admin-password
> asadmin enable-secure-admin
> asadmin restart-domain
```

Managed Container Usage

This usage describes the ability for the Payara Server DAS to control and manage the Docker containers themselves, not just the server instances running within them.

This requires the Docker REST API endpoint to be exposed, which can be done by adding the desired host and port settings to the `DOCKER_OPTS` environment variable (or the command that starts the Docker server):

```
DOCKER_OPTS="-H=0.0.0.0:2376"
```

This usage scenario is designed to mirror the traditional usage of SSH nodes, allowing a user to create, use, and delete Docker Containers as if they were simply Payara Server instances.

This entails setting up a *Docker Node*. From the admin console, this can be achieved from *Nodes > New...* and selecting a type of *Docker*. You will then be prompted to provide:

- A node name
- The host name or IP address of the machine you wish to create the Docker containers on
- The directory of the nodes within the Docker container (safe to leave blank unless you're using a custom Docker image)
- The installation directory of Payara Server within the Docker container (safe to leave as default unless you're using a custom Docker image)
- The docker image name and tag
- The port on the remote machine that the Docker REST API is exposing
- The fully qualified path to the password file on the remote machine to bind to the Docker containers and use for accessing the DAS
- Please note this is the path on the remote machine, not the path within the Docker container or on the machine the DAS is running on - uploading a file local to the DAS is not yet supported.
- Whether to use TLS to talk to the Docker REST API

It is highly recommended outside of development environments that you secure the Docker REST API endpoint with a TLS certificate. Help on how to do this can be found here.

It is also highly recommended that you use the same Docker image tag as that of your Payara DAS so as to avoid any Payara Server version conflicts (e.g. specify the 5.2020.4 tag if running Payara Server 5.2020.4).

Once you have created your node, you can create instances as you would normally if running in a non-containerised environment - any instance created using your Docker node as a target will be created within a container.

The instances are controlled in the same way as non-containerised instances, with the instance state mirroring that of the Docker container:

- Starting a stopped instance will start the Docker container and the instance within it
- Stopping a running instance will stop the instance and the Docker container hosting it
- Deleting an instance will delete the instance and the Docker container hosting it

If you wish to create, start, and stop containers using the Docker CLI rather than from the DAS, you can still do so, though you will need to specify a number of configuration options when creating

the container (described in the *Environment Variables* section below). The mandatory ones are the Payara DAS host and port, the mount for the password file, and the Managed Docker Node name. Additionally you can also specify the config, deployment group, and instance names to use.

```
docker container create --mount 'type=bind,source=${path}/passwordfile.
txt,target=/opt/payara/passwords/passwordfile.txt,readonly=true' -e PAYARA_DAS_
HOST=${PAYARA_DAS_HOST} -e PAYARA_DAS_PORT=${PAYARA_DAS_PORT} -e PAYARA_NODE_
NAME=${PAYARA_NODE_NAME} payara/server-node:${PAYARA_VERSION}

docker container start ${CONTAINER_NAME}
docker container stop ${CONTAINER_NAME}
```

> ***If you delete the container directly from the Docker CLI, the instance that was contained within that container will remain registered to the DAS until you remove it using the*** `delete-instance` ***command.***

Unmanaged Container Usage

This usage describes the Payara Server instances started within the container registering *themselves* to the DAS.

In contrast to Managed Container Usage, you do not need to expose the Docker REST API to use this image in this manner.

> ***As with the*** *Managed Container* ***usage, it is highly recommended that you use the same Docker image tag as that of your Payara DAS so as to avoid any Payara Server version conflicts (e.g. specify the 5.2020.4 tag if running Payara Server 5.2020.4).***

This usage scenario is intended for cases where you do not want the Payara Server DAS to have control over the machine running Docker itself, such as if running in a Kubernetes environment.

Since the Payara Server DAS has no control over the machine the Docker Containers are created on, the containers are typically created from the Docker CLI directly or via an external orchestrator such as Kubernetes. When creating the containers, you must provide the Payara Server DAS host and ports, as well as the mount information for the password file, as environment variables. Additional information such as deployment group, config, and instance name can also be provided via environment variable:

```
docker container run --mount 'type=bind,source=${path}/passwordfile.
txt,target=/opt/payara/passwords/passwordfile.txt,readonly=true' -e PAYARA_DAS_
HOST=${PAYARA_DAS_HOST} -e PAYARA_DAS_PORT=${PAYARA_DAS_PORT} payara/server-
node:${PAYARA_VERSION}
```

Instances registered to this DAS in this way only exist for as long as the container is active for - when the instance or container is shutdown the instance will also be deregistered from the DAS and forgotten about.

> ***Currently if the Docker container is stopped manually or via an external orchestrator, the instance will show as** Stopped **in the DAS until it is restarted, at which point it will be cleared away; only instances stopped from the DAS are immediately cleared away.***

Deploying Applications

You can deploy via the admin console to any running Payara Server instance as with a non-containerized Payara Server instance (an instance not running in a Docker container), simply targeting the instances or deployment groups you wish to deploy your application.

If you wish to have the application deploy upon creation and startup of the Payara Server instances within the containers, you must create a deployment group and deploy the application to this. You must then pass this deployment group as a parameter to the Docker container to add the instance to it upon startup:

```
docker run --mount 'type=bind,source=${path}/passwordfile.txt,target=/opt/
payara/passwords/passwordfile.txt,readonly=true' -e PAYARA_DEPLOYMENT_GROUP=DG1
payara/server-node:${version}
```

The Container Lifecycle

With the default Docker entrypoint, Payara Server Docker container will do the following during start-up:

1. Determine the Docker Container IP if it hasn't already been provided. If an override hasn't been provided using the `DOCKER_CONTAINER_IP` container environment variable, the container will resolve the variable itself using the hostname -I command.
    - The container will always print out what hostname or IP address it is using.

2. Check if an instance name has been provided using the `PAYARA_INSTANCE_NAME` container environment variable.
    - If no name has been given the container will always create a new Payara Server instance with an autogenerated name.
        - If a node name has been provided using the `PAYARA_NODE_NAME` container environment variable, the Docker container will check against the DAS if the details of said node match the local Docker container environment (e.g. host name), creating a new node with an autogenerated name if they don't.
    - If a name has been provided, the container will check to see if the instance has already been created (in case a container is being reused), creating and registering it to the DAS if it hasn't.
3. The Payara Server instance will be started in the foreground.

Environment Variables

The following environment variables are available for use in either usage scenario. Configuration of instance settings and deployments is expected to be configured beforehand on the DAS.

Variable Name	Default Value	Description
`HOME_DIR`	`/opt/payara`	The home directory for the `payara` user.
`PAYARA_DIR`	`/opt/payara/appserver`	The root directory of the Payara installation.
`PAYARA_PASSWORD_FILE_DIR`	`/opt/payara/passwords`	The directory to store password files in.
`PAYARA_PASSWORD_FILE`	`/opt/payara/passwords/passwordfile.txt`	The location of the password file for asadmin. This can be passed to asadmin using the `--passwordfile` parameter.
`PAYARA_DAS_HOST`	`localhost`	The IP address or DNS name of the server hosting the Payara Server DAS (relative to the container).
`PAYARA_DAS_PORT`	`4848`	The admin port of the Payara Server DAS.
`PAYARA_NODE_NAME`		The name of the node to register this instance to - this is typically only used by when setting up and using managed Docker containers. If not specified, one will be generated and the container will be used in an unmanaged capacity.

Variable Name	Default Value	Description
PAYARA_CONFIG_NAME	default-config	The name of the config to attach to the Payara Server instance. If not specied, the default behaviour of Payara Server instances of making a copy of the *default-config* config will be created.
PAYARA_INSTANCE_NAME		The name of the instance to create. If not specified, one will be autogenerated.
PAYARA_DEPLOYMENT_GROUP		The deployment group for the instance to join upon startup.
DOCKER_CONTAINER_IP		The IP address of the Docker container. This only needs to be provided if the DAS would not be able to contact the Docker container at the address the container itself contacts itself on.

Payara Server also allows you to externalise various aspects of configuration through the usage of variable replacement, utilising environment variables, system properties, and MicroProfile Config properties. This allows you to, for example, extend the image with additional environment variables and reference them within your application like so:

```
@DataSourceDefinition(name="java:global/ExampleDataSource",
      className="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource",
      user="${ENV=mysql.user}",
      password="${ENV=mysql-db-password}",
      databaseName="${ENV=MYSQL_DBNAME}",
      serverName="localhost"
   )
```

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/server-full/
10. https://hub.docker.com/r/payara/server-web/
11. https://hub.docker.com/r/payara/server-node

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