# Application Metrics Made Simple

IBM Runtime Tools, UK

## OBJECTIVES

This workshop will teach you how to deploy a web application with built-in performance metrics and a dashboard for Java, Swift or Node.js

Using tools created by the IBM Runtimes team you will see how to quickly get feedback about the state of your application. The workshop will also show you how to get started with driving workloads into a web application, and all from a Docker file.

**Open Source Software**

*The three 'Application Metrics' dashboards developed by IBM and used in this workshop are all Open Source, licensed under the Apache 2.0 license.*

## PREREQUISITES

- Docker: www.docker.com
- Docker-Compose: docs.docker.com/compose/install/
- Git (optional, you can also download the source as a zip)
- Source Files: either run
    - `git clone http://github.com/RuntimeTools/DockerDashboardsWorkshop`
    - Or open github.com/RuntimeTools/DockerDashboardsWorkshop in a browser and download the source in a zip file.

## EXERCISE 1: A SIMPLE NODE.JS WEB APPLICATION

Launch a simple Node.js web server in a Docker container and view the **Application Metrics for Node.js** dashboard.

```
cd DockerDashboardsWorkshop/NodeApp
```

There are three files in this directory that we need to get a Node.js web server running in a docker container.

- Dockerfile – tells Docker how to build the image
- app.js – Node.js main file
- package.json – Describes your package, including dependencies

Let's take a look at app.js

```
require('appmetrics-dash').attach()
const http = require('http');

module.exports = http.createServer((req, res) => {
    // Send "Hello World" to every request
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello World');
}).listen(3000, () => console.log("Node.js HTTP server listening on port 3000"));
```

`require('appmetrics-dash').attach()` is all that is needed to 'inject' **Application Metrics for Node.js** monitoring into your application and enable it. The rest of this file defines an extremely simple HTTP server which returns "Hello World" to every request.

What's in the Dockerfile?

Get running:

`cd DockerDashboardsWorkshop/NodeApp` (if you didn't do this earlier)

```
docker build -t node_app .
docker run -t -p 3000:3000 node_app
```

```
# Base image
FROM node:boron
# Copy app source
COPY . .
# Install dependencies
RUN npm install
# Expose port 3000 which the HTTP server runs on
EXPOSE 3000
# Command to run when this Docker image starts
CMD [ "npm", "start" ]
```

Passing `-t` to 'docker build' names the image you are building.

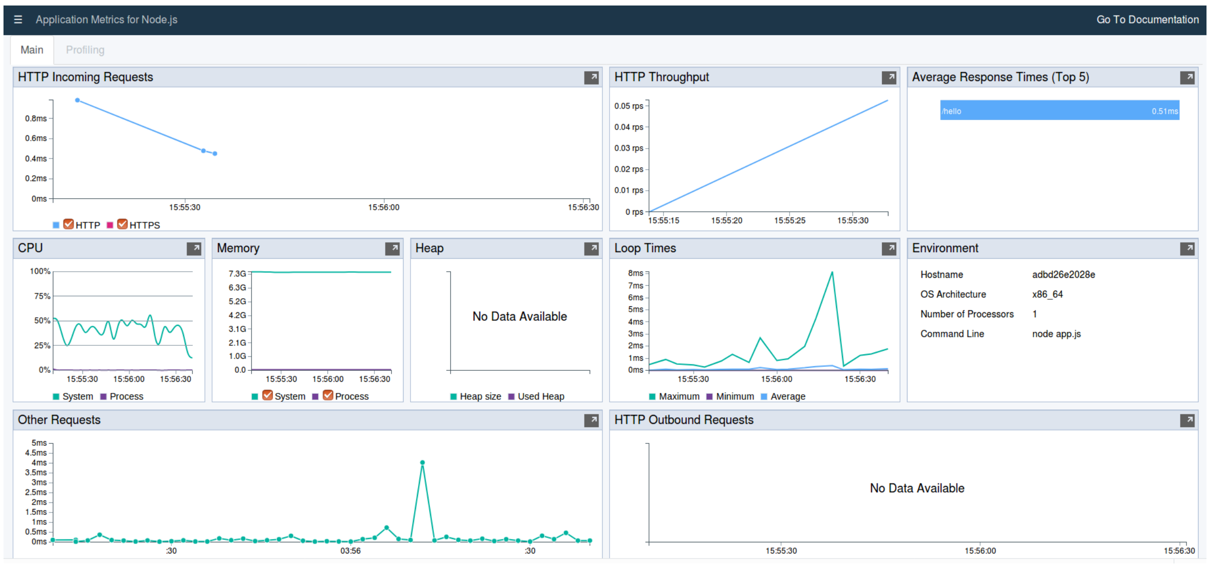
Passing `-t` to 'docker run' will allocate a pseudo-TTY, which allows us to use Ctrl+C to get out of the container logs without stopping the container.

`-p` maps ports from your docker image to your local machine. The Application Metrics dashboard is also served on port 3000.

Open a browser at http://localhost:3000/ to see your "Hello World" message

Open a browser at http://localhost:3000/appmetrics-dash

Welcome to the **Application Metrics for Node.js** dashboard! It should look something like this:



| | |
|---|---|
| Hostname | adbd26e2028e |
| OS Architecture | x86_64 |
| Number of Processors | 1 |
| Command Line | node app.js |

Refresh http://localhost:3000/ a few more times to see HTTP Incoming Requests appear on the graph.

Most of the data is plotted as line graphs. **HTTP Incoming Requests**, **HTTP Outgoing Requests** and **Other Requests** show event duration against time. **HTTP Throughput** shows requests per second. **Average Response Times (top 5)** shows the 5 incoming HTTP requests that took the longest on average. **CPU** and **Memory** graphs show system and process usage over time. **Heap** shows the maximum heap size and used heap size over time. **Loop Times** shows samples taken at intervals from the Node.js event loop, with one point for the shortest latency, one for the average and one for the longest for each sample taken.

If a graph has points, hovering over one of these points will give additional information. For example 'HTTP Incoming Requests' will show the response time and the requested url.

A maximum of 15 minutes of data is shown across all graphs.

If a lot of data is being produced by the application being monitoring then the dashboard will automatically start to aggregate data. Looking at the HTTP Incoming Requests chart again, each point will represent all the requests for a 2 second period. The tooltip shows the total number of requests along with the average time taken and the longest time. The longest time is the value actually plotted.

Don't stop your application yet, we'll be sending requests to it in the next exercise.

**Further Work**: Can you edit app.js to send "Hello World" as JSON on the /json endpoint? I.e http://localhost:3000/json

## EXERCISE 2: DRIVING WORKLOAD

When developing and testing a web application it can be useful to use a workload or benchmarking driver to send HTTP requests to your application.  Today we will use 'wrk2' from https://github.com/giltene/wrk2 which is a fork of 'wrk' https://github.com/wg/wrk. Wrk2 enables a steady stream of HTTP requests while wrk generates significant load. Apache JMeter is also a popular option for performance testing, and is Open Source.

If you are running Linux it is simple to build and run wrk2 on your laptop. There are a couple of extra steps if you're on a Mac. If you don't have Linux, Mac OS or a Linux Virtual Machine then you may want to just skip this exercise. Later on we launch the workload driver in a Docker container so you won't be missing out.

To build on Linux (Ubuntu):

```
sudo apt-get install libssl-dev
sudo apt-get install build-essential
git clone https://github.com/giltene/wrk2
cd wrk2
make
```

To build on Mac:

```
brew install openssl
git clone https://github.com/giltene/wrk2
```

Open wrk2/Makefile and change the top 2 lines (the text you need to add is in bold):

```
CFLAGS  := -std=c99 -Wall -O2 -D_REENTRANT -I/usr/local/opt/openssl/include
LIBS    := -lpthread -lm -lcrypto -lssl -L/usr/local/opt/openssl/lib
```

```
cd wrk2
make
```

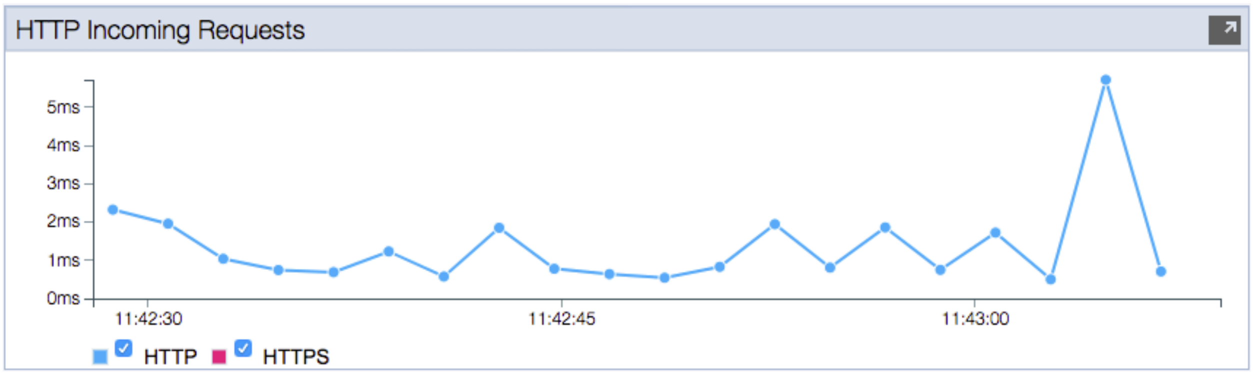
To run workload:

```
./wrk -t2 -c10 -d60s -R200 http://localhost:3000/
```

This runs a benchmark for a duration of 60 seconds (-d60) using 2 threads (-t2) and 10 open HTTP connections (-c10), sending 200 requests per second (-R200).

Take a look at your graph in the Application Metrics for Node.js dashboard.

You should see a lot of points appearing on the HTTP graph:



***Note:*** HTTP data is aggregated. Try hovering over one of the points on the graph to see its tooltip.

Next try changing some of the arguments to the 'wrk' command e.g. `-t 2m` will run for 2 minutes

## EXERCISE 3: A JAVA WEB APPLICATION

Launch a simple Java web server in a Docker container and view the **Application Metrics for Java** dashboard

```
cd ../JavaApp
```

We have included the **Application Metrics for Java** packages here for ease, but they are published on Maven Central: https://mvnrepository.com/artifact/com.ibm.runtimetools

Let's have a look at the Dockerfile:

```
# Base Image
FROM websphere-liberty:javaee7

# Specify a working directory (need to refer to this in jvm.options file)
WORKDIR /usr/javametrics

# Expose port 9080 which the HTTP server runs on
EXPOSE 9080

# Copy sample app into Liberty's dropins directory
ADD JavaSampleProject.war /config/dropins/

# Copy Web Dashboard into Liberty's dropins directory
ADD javametrics-dash-1.0.1.war /config/dropins/

# Add config files and other dependencies
ADD server.xml /config/
ADD jvm.options /config/
ADD asm /usr/javametrics/asm
ADD javametrics-agent-1.0.1.jar /usr/javametrics/
```

This sample Java Web Application uses Websphere Liberty as the Web Server. The example app is implemented as a Servlet. It has been packaged into JavaSampleProject.war for ease of deployment but the source can be found under:

DockerDashboardsWorkshop/JavaApp/JavaSampleProject/

You might notice that there's no CMD section. Starting the Web server in this case is taken care of by the base image.

Get running:

```
docker build -t java_app .

docker run -t -p 9080:9080 java_app
```

Open a browser at http://localhost:9080/JavaSampleProject/json to see your "Hello World" message

Open a browser at http://localhost:9080/javametrics-dash to open the dashboard

The Application Metrics for Java Dashboard is slightly different to the Dashboard for Node.js. The **Heap** graph shows four different measurements of memory over time and the **Garbage Collection Time** graph shows the amount of time spent in GC as a percentage.

## EXERCISE 4: A SWIFT WEB APPLICATION

Launch a simple Swift Kitura web server in a Docker container and view the Swift Application Metrics Dashboard

```
cd ../SwiftApp
```

Get running:

```
docker build -t swift_app .

docker run -t -p 8080:8080 swift_app
```

Open a browser at http://localhost:8080/swiftmetrics-dash to open the dashboard

If you are new to Swift it would be worth familiarising yourself with the Dockerfile, Package.swift and the contents of SwiftDashDemo/Sources/main.swift.

Then try sending some workload to the Swift server on http://localhost:8080/json, which is the endpoint for the test application. If you were unable to compile wrk2 in Exercise 2 then just open the URL in your browser and refresh a few times to see HTTP requests appearing in the dashboard.

## EXERCISE 5: DOCKER-COMPOSE

Compose all three dashboards using Docker compose and launch them all with one command.
Docker compose allows you to compose together multiple docker containers and start them simultaneously. We're going to compose our 3 HTTP servers and then add a 4th container to run the workload driver.
To configure docker-compose all we need is 1 YAML file. You will find it in ComposedApp/docker-compose.yml. Please note that the names of the images are important here. In exercise 1,2 and 3 we called them node_app, java_app and swift_app. If you used different names you will need to edit the docker-compose file accordingly.

At this point it would be good to make sure you've killed all of the docker containers you launched previously. Open another console and run:

```
docker stop $(docker ps -q)
```

(Note: this stops all running Docker containers. Run `docker ps` if you want to find the ID of a specific image.)

To start the composed docker containers:

```
cd ComposedApp
docker-compose up
```

Now all 3 dashboards will be running in your browser
http://localhost:3000/appmetrics-dash
http://localhost:8080/swiftmetrics-dash
http://localhost:9080/javametrics-dash

Try running some workloads.
Use Ctrl+C to stop all the containers

## EXERCISE 6: A DOCKER IMAGE FOR WORKLOADS

In this exercise we will add a 4th Docker image to our composed app to run the workload driver.

Change directory:

```
cd DockerDashboardsWorkshop/WorkloadDriver
```

Get the source for wrk2:

```
git clone https://github.com/giltene/wrk2
```

(or copy the source you cloned in Exercise 2).

Build the Docker image:

```
docker build -t workload_driver .
```

Go back to our docker-compose.yml and uncomment the 4th section :

```
wrk2:
  image: workload_driver:latest
  depends_on:
    - swiftmetrics
    - appmetrics
    - javametrics
```

Launch the containers again:

```
docker-compose up
```

Open the 3 dashboards again or refresh the web pages in your browser. Notice there are HTTP requests being sent to all 3 servers this time by the workload driver.

## FURTHER WORK

- Interested in Prometheus? Try appmetrics-prometheus, or one of the built in /metrics endpoints in Application Metrics for Java or Application Metrics for Swift.
- Explore the dashboards, hover over the points of the HTTP graph, maximize and minimize graphs, try the profiling view for Node.js
- Change the amount of workload by editing WorkloadDriver/workload.sh
- Get the web servers to do something more interesting than just return Hello World
- Try the dashboard with your own Java, Swift or Node.js web application