# Dockerunit: Multi-replica Ephemeral Deployments for Advanced Testing of Microservices

Quirino Zagarese, Pedro Nuno Santos,
Vincenzo Candela, and Francesco Sorice

Dockerunit, https://dockerunit.github.io/
quirino.zagarese@gmail.com pnuno.santos@pm.me
irotech@gmail.com fsorice89@gmail.com

## 1 Introduction

Testing Microservices can be a challenging task, especially when several services and multiple replicas are involved. Although software testing best practices suggest that one should follow the Testing Pyramid approach[5] and focus most of the effort on unit testing, the distributed nature of Microservices is shifting most of the complexity towards the integration phase and the emulation of those failures that characterize distributed systems. Modern enterprises are taking advantage of public cloud infrastructure to build testing environments, however, this approach can be very expensive and difficult to scale, as it puts considerable pressure on infrastructure teams and often leads to unnecessarily large bills. In reality, testing environments are usually shared across teams, but this can be a source of frustration as the access needs to take place in mutual exclusion and this prevents teams from delivering at their maximum capacity.

These limitations, along with the proliferation of containerization[8], have led to the creation of few useful tools that simplify the testing of containerized services. TestContainers[2] provides a Java builder API to create Docker[7] containers from within tests, however, it has no support for multiple service replicas. Docker-compose-rule[3] by Palantir provides a similar API, but it builds on top of docker-compose[1] and, as such, it doesn't foster descriptors reusability.

We present Dockerunit: a Java framework that enables declarative tests specifications and manages the full test lifecycle for dozens of multi-replica Docker services, without relying on complex orchestration platforms like Kubernetes[4].

## 2 Reusable descriptors and service discovery

Dockerunit provides Java annotations to (i) create service-descriptor classes and (ii) tie multiple services together into a testing deployment configuration. A descriptor class declares a service name, a Docker image, a health-check definition that is leveraged by the framework to implement reliable service discovery, and further optional details like port-mapping, volume binding and environment variables. Listing 1 shows an example of a descriptor class.

```java
@Svc(name="my-microservice", image="my-microservice-image:latest")
@WebHealthCheck(endpoint="/health-check", port=8080)
@PublishPorts // maps container level ports to random host ports
@Volume(host="application.properties", container="/usr/bin/application.
    properties", useClasspath=true)
@UseConsulDns
public class MyMicroserviceDescriptor {}
```

Listing 1: Example of a descriptor class

Tests can import multiple descriptors, define the starting order and the number of replicas for each service as shown in listing 2.

```
@WithSvc(svc=MySQLDescriptor.class, priority=10, replica=1)
@WithSvc(svc=MyMicroserviceDescriptor.class, priority=1, replicas=3)
public class MyIntegrationTest {
  @Rule public DockerUnitRule rule = new DockerUnitRule();
        private ServiceContext context;
  @Before public void setup() {
    context = DockerUnitRule.getDefaultServiceContext();
```

Listing 2: Test including multiple services and replicas

Before tests are executed, Dockerunit creates the necessary Docker containers based on the specified ordering (i.e. inverse of "priority") and replica parameters. Afterwards, it performs service discovery by leveraging Consul[6] and the health-check configuration (@WebHealthCheck). Once all the services are running and healthy, Dockerunit gives control to the tests and provides a *ServiceContext* instance that can be used to get access to each instance of each service, with the aim to call any of the endpoints of the Microservices under test. Consul is not only used for discovery: it works as a DNS resolver and load-balancer. By using the *@UseConsul* annotation (see listing 1), services can resolve each other by using the name that is specified in *@Svc*. If, for instance, we set the name in the *MySQLDescriptor* to "mysql", then "my-microservice" would be able to resolve it by using the name "mysql.service.consul". This name could be specified in the "application.properties" configuration file that has been mounted inside the each replica of "my-microservice" using *@Volume* (listing 1), which allows developers to test service-to-service communication without interfering with the networking layer as docker-compose would. Finally, if another microservice needs to communicate with "my-microservice", it would use the corresponding name and Consul would perform DNS load-balancing across the available replicas. Developers can easily extend Dockerunit by creating new annotations and providing the necessary logic that interprets such annotation. By implementing a specific interface, developers can access low level Docker APIs and cater for any kind of container customisation.

## 3 Conclusion and future work

We have provided a quick overview of Dockerunit[1]. Although this example looks fairly simple, Dockerunit is currently the main Microservices testing tool at Yoti, a Digital Identity company, where it is used to test dozens of Microservices on three global products. In the future, we aim to evolve the framework by providing high level APIs for fault injection and chaos engineering.

## References

[1] Docker compose. https://docs.docker.com/compose/, 2013.

[2] Test containers. https://www.testcontainers.org/, 2015.

[3] Docker compose rule. https://github.com/palantir/docker-compose-rule, 2016.

[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):70–93, 2016.

[5] M. Fowler. Testpyramid. https://martinfowler.com/bliki/TestPyramid.html, 2011.

[6] Hashicorp. Modern service networking for cloud and microservices. https://www.hashicorp.com/resources/modern-service-networking-cloud-microservices/, 2019.

[7] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[8] C. Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[1]Dockerunit is available at https://dockerunit.github.io