Scalable Hello World

Mike Sollanych - [mike@sollanych.com](mailto:mike@sollanych.com) – june 2017

# Overview

This document details a project developed to implement a scalable Ruby on Rails application for demonstration purposes. All source code used is available at: <https://github.com/mike-sol/scalable-hello>

Table of Contents

[Overview 1](#_Toc484460836)

[Requirements (as provided) 1](#_Toc484460837)

[Technology Overview 2](#_Toc484460838)

[Development Environment: 2](#_Toc484460839)

[Production Environment: 2](#_Toc484460840)

[Working with the Development Environment 2](#_Toc484460841)

[Components of the Solution 3](#_Toc484460842)

[Setting up the Production Environment 3](#_Toc484460843)

[Scaling The Solution 6](#_Toc484460844)

[Monitoring the Solution 7](#_Toc484460845)

# Requirements (as provided)

The system/service that we would like you to deploy is a web application that can be auto-scaled in the future. To this end:

* Provision server(s) to run a Rails application using MySQL as its database.
* The Rails application can just be the default “Hello World!” type.
* It should be able to easily and dynamically add and remove application servers that can communicate to an existing database server.

Additionally, please document how the following can be achieved

* Auto Scaling
* Monitoring/Alerting

# Technology Overview

Part of the fun of working on a demonstration project is taking on the challenge of learning new, cutting-edge technologies. To that end, I chose to work with Docker, intending to do as little configuration management as possible, aiming at an immutable infrastructure instead of a conventional, long-lived VM architecture.

Due to time constraints, it seemed wise to use off-the-shelf, ready-to-deploy components where possible, instead of reinventing the wheel. The resulting project fulfills the requirements above, but I will point out issues with this in the document below in red, along with my suggestions for how one would implement this more sustainably in a true production environment.

## Development Environment:

* Docker Compose running containers locally
* Ruby on Rails, with the Puma application server
* Locally hosted, containerized MariaDB

## Production Environment:

* Docker Swarm for AWS
* Ruby on Rails, with the Puma application server
* Amazon RDS-hosted MariaDB

# Working with the Development Environment

The below instructions are for Ubuntu, but can be adapted to work on Mac or PC.

Refer to the source instructions at <https://docs.docker.com/compose/rails/>

1. Install pre-requisites:
   1. Docker Engine: <https://docs.docker.com/engine/installation/linux/ubuntu/>
   2. Docker Compose: <https://docs.docker.com/compose/install/>
2. Clone the <https://github.com/mike-sol/scalable-hello> repo locally (or fork it and then clone)
3. Create an environment file by copying `.env.example` to `.env` and editing.  
   Instructions are within the file; at this point the only necessary step is to set a custom password for your local database instance.
4. Run `sudo docker-compose build` to run an initial build.   
    This will download the needed source images from Docker Hub and then prepare them with custom configuration.
5. Run 'sudo docker compose up' to start the database and Rails server.
6. If this is the first execution of this image, prepare the database:
   1. Open another terminal window
   2. Run `./prepare-db.sh`
7. Browse to the locally-running application on <http://localhost:3000>.

# Components of the Solution

Here are some of the key files, as seen in the source code at <https://github.com/mike-sol/scalable-hello>

* **Gemfile**: Describes the Ruby gems needed by this application
* **Rakefile**: Jumping off point for build-time jobs for Ruby on Rails
* **Dockerfile**: Describes the steps needed to build the `web` container, which runs Ruby on Rails. Inherits from a commonly available container (ruby:2.3.3) and installs gems from the Gemfile.
* **docker-compose.yml:**   
  describes the local development docker-compose environment, including the two containers needed (local `web` container built from the above Dockerfile, and imported `db` container based on the community MariaDB image)
* **docker-compose-production.yml:**   
  describes the production environment, which sources officially built and released versions of the `web` container from the Docker Hub container registry
* **scalablehello.dab**: A generated file made from the docker-compose-production.yml file for stack deployment
* **prepare-db.sh**: A script to run the Rails commands needed to set up the local database on first run

# Setting up the Production Environment

1. Basic AWS setup
   1. Sign up for AWS, or use an existing account if you have one.
   2. Create an EC2 Keypair named "docker" and retain the resulting 'docker.pem' file.
2. Deploy Docker Community Edition for AWS
   1. Navigate to <https://docs.docker.com/docker-for-aws/#quickstart>
   2. Essentially, this deploys a CloudFormation template to spin up a VPC and a Docker Swarm.  
      Use the following parameters, leaving the rest as default:
      1. Stack Name: Docker
      2. Number of Managers: 1
      3. Number of Workers: 2
      4. SSH key to use: docker

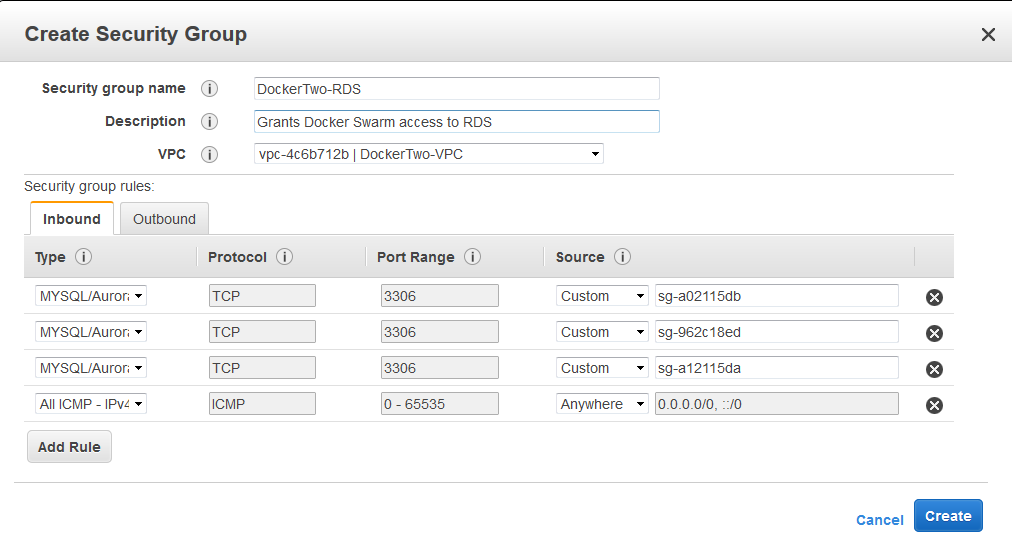
This will create a new VPC, and within it a Docker Swarm composed of two Amazon ASGs and an Elastic Load Balancer. This can be scaled out to 3 or 5 managers and hundreds of workers.

In real production use, one would deploy at least 3 managers and probably 5 workers to start, depending on capacity analysis.

1. Deploy a MySQL-compatible database on AWS.

In real production, you may want to use Amazon Aurora for a scalable MySQL database with high performance. Significantly more capacity analysis would be needed during this step, as well as planning for backups / snapshots, potentially sharding, replicas, and more.   
  
For my test, I deployed a MariaDB instance in the Free Tier.

* 1. **Important:** Create a RDS security groups to enable the Docker nodes access. Create a new security group like the below, using the security group IDs of the automatically-created Docker security groups:



* 1. Deploy a new RDS instance with the following:
     1. DB instance identifier: rails
     2. Master username: rails
     3. Password: Generate a secure string and save for later
     4. VPC: Use the newly created Docker-VPC
     5. Security group: Use the one created above
     6. Database name: rails
  2. When the RDS instance finishes deploying, you will need to put the URL for it into `docker-compose-production.yml` so that the Rails application can find its database instance.   
     In actual production, a DNS CNAME or a service discovery layer like Consul would be used for this instead of hard-coding the RDS name.

1. Take a look at the Docker Swarm and prepare it for use.
   1. SSH to one of the Docker Manager nodes.   
      To do this, look in the EC2 dashboard for the list of currently running nodes, and get the Public IP for any of the Docker-Manager instances.  
        
      Connect to it with: `ssh -i <path-to-your-saved-docker.pem> docker@<public-IP>`
   2. Take a look at what's running now:
      1. `docker ps` shows you all of the Docker containers on the current host
      2. `docker node ls` shows you all of the Docker nodes in the swarm
      3. `docker node ps $(docker node ls -q)` will show you all running containers across all of your nodes.   
         There shouldn't be anything running in the latter right now!
   3. Configure the needed Docker Secrets for the application.
      1. Put the database password into place as a Docker Secret.  
         `echo "<RDS database password>" | docker secret create rails\_database\_password -`
      2. Generate a 64-character password to use for the Rails secret key base, and similarly install it.  
         `echo "<Rails secret key base>" | docker secret create rails\_secret\_key\_base - "`
2. Deploy the application to the Docker Swarm.

In actual production use, you would use a CI tool like Jenkins, GoCD, or Bamboo with an SSH key to the Docker Manager nodes in order to run the deploy in an automated, controlled fashion.   
For the sake of this test, we'll perform those steps manually.

You'd also likely be running your own Docker container registry, whether self-hosted, using Amazon ECR, Azure ACS, or some other mechanism. In this case, we're using the Docker Hub and simply posting the image publicly because there's nothing sensitive about it.  
  
Steps 5a and 5b are only needed if the application has changed.  
If you just want to deploy the existing version, start at step 5c.

* 1. Publish the newest version of the container to Docker Hub.  
     See <http://ropenscilabs.github.io/r-docker-tutorial/04-Dockerhub.html>
     1. Log into Docker Hub:  
        `sudo docker login --username=yourhubusername [--email=youremail@company.com`](mailto:--email=youremail@company.com%60)
     2. Tag the image with a version string:  
        `sudo docker images` to see the Image ID  
        `sudo docker tag <imageID> mikesol/scalablehello\_web:<version>`
     3. Push the image to Docker Hub  
        `sudo docker push mikesol/scalablehello\_web`
  2. Bundle the application to produce a Distributed Application Bundle (.dab) file
     1. Make sure the version tag in docker-compose-production.yml lines up with the one you used above
     2. Ensure the image has been pulled down, and then bundle it:  
        `sudo docker-compose -f docker-compose-production.yml pull web`  
        `sudo docker-compose -f docker-compose-production.yml bundle`

This produces `scalablehello.dab`.

* 1. Copy the resulting application bundle up to the Docker Swarm.  
     `scp -i ~/.ssh/docker.pem scalablehello.dab docker@<swarm-manager-ip>:`
  2. SSH into the Docker Swarm manager node and deploy the stack:

`docker deploy --bundle-file scalablehello.dab scalablehello`

* 1. Now, `docker service ls` should return like below:

ID NAME MODE REPLICAS IMAGE PORTS

od43899xma5c scalablehello\_web replicated 1/1 mikesol/scalablehello\_web@sha256:7ad145fbce1190ea062dcd41a52600be45b0c004a3bd35c9cbb7f29b380cf224 \*:0->3000/tcp

* 1. `docker service ps scalablehello\_web` should return like below:

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

x1zl63ovpru7 scalablehello\_web.1 mikesol/scalablehello\_web@sha256:69e8f3f306935aab186eac860efbf021565f80ed64f955f2f895a1fd2e08bdae ip-172-31-9-183.us-west-2.compute.internal Running Running 43 seconds ago

* 1. Give the service access to the secrets:
     1. `docker service update --secret-add="rails\_database\_password" scalablehello\_web`
     2. `docker service update --secret-add="rails\_secret\_key\_base" scalablehello\_web`

This will cause initially-started containers to be killed and new ones launched in their place.  
  
*It looks like it should be possible to specify the secrets an app has access to via the docker-compose file, but this didn't work well in my testing, and documentation on this is spotty.*

* 1. First run only:  
     Run the database migrations to prepare the table schema and seed initial data.

In a normal production environment, this would make more sense to execute either manually from a management host, or automatically via CI if there’s a high degree of confidence. In our case, we’ll actually use one of the running Docker containers to do this, so that we don’t have to deploy another instance.

* + 1. SCP the docker.pem key up onto the swarm manager node. This will enable you to SSH to the swarm cluster members, which aren’t reachable from outside.
    2. On the docker swarm manager, use:  
       `docker service ps scalablehello\_web` to find the name of one of the nodes that the service is presently running on.
    3. SSH into the node in question: `ssh -i docker.pem <nodename>`
    4. Type `docker ps` to get a list of running containers.
    5. Use `docker exec -it <containerID> bash` to open a shell inside the container.
    6. Inside the container, run:  
       `RAILS\_ENV=production bundle exec rake db:migrate` and

`RAILS\_ENV=production bundle exec rake db:seed`

* + 1. Exit the container, and then disconnect from the node.
  1. Scale the service to two or more instances:  
     `docker service scale scalablehello\_web=2`

`docker service ps scalablehello\_web | grep -vi shutdown`

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

mpj0wc0w7wmo scalablehello\_web.1 mikesol/scalablehello\_web@sha256:3492417d02377ab0f37b2d61b9bbf203bc5854ef0c08345e89de883fae70337c ip-172-31-18-27.us-west-2.compute.internal Running Running 18 minutes ago

5uj9wxcump9y scalablehello\_web.2 mikesol/scalablehello\_web@sha256:3492417d02377ab0f37b2d61b9bbf203bc5854ef0c08345e89de883fae70337c ip-172-31-29-127.us-west-2.compute.internal Running Running 13 seconds ago

* 1. Publish a port for the service:  
     `docker service update --publish-add 80:3000 scalablehello\_web`  
     This will again cause the currently-running containers to be tossed and re-started. Luckily, it all happens within a minute.
  2. It should now be running and accessible!   
     To access it, since we haven't configured a vanity DNS name for the load balancer, navigate to the EC2 dashboard, look under Load Balancers, and get the name (e.g. Docker-ExternalLoa-1NKWM0E04W0X5-692705370.us-west-2.elb.amazonaws.com) and then just browse to that address.

### Some final considerations:

* This application is being hosted with the Puma webserver, which is normally intended only to serve dynamic Ruby content and not static assets, which should be served by Nginx or Apache.   
  **Due to time constraints,** a full webserver was not implemented, so the pretty static assets (e.g. Bootstrap CSS) which appear when running locally will not appear in production.
* The workflow for development and upload involves a fair number of manual steps, most of which would be scripted away behind CI or manual tooling.

# Scaling The Solution

Three levels of scaling can be configured with this configuration

* **The size of the Docker Swarm instances**; this defaults to t2.micro, but for a production deployment a larger instance would be appropriate. Ideally, the instance size is set at a sweet spot for price/performance; determining the exact size to use is likely a matter of tuning in production.  
    
  With the current solution, the instance size is fixed at stack creation time. It is possible to dynamically update an AWS launch configuration in order to change this, however, this may damage compatibility with Docker for AWS CloudFormation stack updates in the future, so it’s not recommended. Better to deploy a new swarm if the size needs to change.
* **The number of Docker Swarm members**; this can be done by editing the AWS auto-scaling group and simply setting the *Desired* value to any number.

Conditions can be established for automatically scaling this on demand based on the CPU and memory load of the machines in question; for a highly elastic workload, these conditions should be set at an appropriate threshold so that new swarm members will be ready to go before current ones are overloaded.

* **The number of container instances per service**; this is presently handled manually by using the `docker service scale` command on a manager node. This is clearly less than ideal, but it stands to reason that the Docker project – which moves quickly! – will be adding functionality to specify things like “number of instances to run per swarm node” which will make auto-scaling easy.   
    
  To implement this on the current solution, it would be necessary to use an external orchestration service that polls AWS to determine the size of the ASG and then adjusts the number of expected replicas as appropriate.

After implementing this, it is clear to me that it would be worthwhile to investigate the use of a managed container service, such as Azure ACS, or to look into different cluster schedulers with more functionality like Kubernetes or Nomad. Docker Swarm is likely to be a first class solution eventually, but is somewhat lacking at present.

# Monitoring the Solution

Typically, we’ll want to gather three kinds of information:

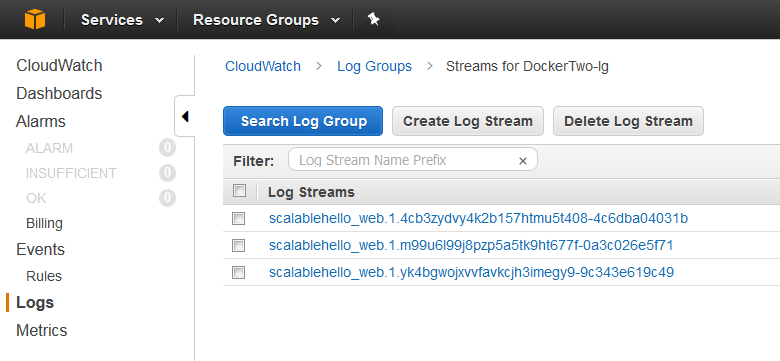
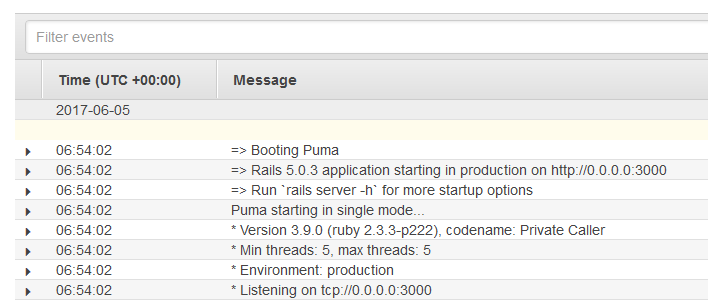
* Health checks, which test various aspects including:
  + Whether a host is up, and reachable over the network
  + Needed services are running
  + Load and free disk space are reasonable
  + Ports are listening and return as expected   
    (e.g. HTTP response codes for testing endpoints, valid SSL certs)
* Performance metrics, including:
  + Number of containers running
  + Requests / queries per second
  + HTTP responses broken down by code
  + Queue depths and input/output rate
* Logs:
  + HTTP, application, and database server logs
  + Container host and cluster logs
  + Network logging

Typical stacks for the above include pieces like Nagios (Opsvew / Icinga), Consul, Solarwinds, ELK stack, TICK stack, DataDog, Sumo Logic, Graphite, and more. Managing these systems can be a job in itself, leaving the decision as to whether to pay for them in hosted SaaS form, or pay for the time to run your own.

It is critical that these three classes of monitoring are applied at both the host and the container level. In production, it may be necessary to deploy agents within containers and to hosts. Most monitoring systems are not yet container-aware, so it may be necessary to generate unique, informative host names based on factors like the swarm, host, and container ID.

## Monitoring within AWS

Docker for AWS automatically sends container logs to CloudWatch:

Amazon’s own built in monitoring may be sufficient to some extent, but it’s really seen best more of as a source of data to load into other services. On the upside, services like SQS and Kinesis can be used to facilitate putting logs onto a stream or queue in order to consume them with other services.

# Stretch Goals

Some thoughts on the suggested stretch goals, which I unfortunately do not have time to implement:

* **Database redundancy**: this can either be accomplished using AWS’s own service functionality, or by running the database within a cluster of VMs. There are tradeoffs to both models; as with logging, effectively running a stable clustered database may still require a DBA level skillset.
* **Blue/Green Deployment:** this is very easy to accomplish by creating a new Docker image with the modified application code in it, deploying it with a service name including a \_blue or \_green suffix, and simply unpublishing the bound ports from the outgoing instance and assigning them to the new. In the event of failure, rollback is as easy as switching the ports back.
* **Testing:** Three kinds of testing would be appropriate for this application:
  + Unit testing, which could be used to test base model functionality for the individual ‘hello’ items
  + Functional testing, which is used at a higher level to ensure that appropriate logic is executing correctly when interacting with the ‘hello’ items
  + Integration testing, which is used to test workflows end-to-end, including all components of the system (database and front-end).

As the application itself is a scaffolded Rails app with little customization, default unit tests are included that execute successfully. Run `sudo docker-compose run web rake test`:

Starting scalablehello\_db\_1 ... done

Run options: --seed 8373

# Running:

.......

Finished in 8.217298s, 0.8519 runs/s, 1.0953 assertions/s.

7 runs, 9 assertions, 0 failures, 0 errors, 0 skips

# Conclusions

This project was a fascinating exercise. I had been meaning to delve into the world of Docker more deeply than before, and it was nice to have an external impetus to spend free time on this.

While there are some remaining issues with the application that could be sorted out given more time, I am pretty happy with the results, and believe it to be a strong foundation that a more advanced application could be built on. More time reading about best practices for this very young set of technologies would serve well to help ensure clean and efficient building and deployment.

Thanks for reading! - Mike Sollanych