So I recently dived into container schedulers (like **Kubernetes**, **DC/OS** (**Mesos**) or **Docker Swarm Mode**). You know, all the stuff the cool kids talk about.

And truth be told, I really liked the ephemeral nature of it all – and using things like **Terraform** really makes setup a breeze (once you leaped over the „fundamentals“ learning curve). However, I found a small hole in the concept: Configuration management inside containers. Like, how do you configure the application, that is inside the container? For example, take an Apache http server. How do you easily and effectively configure something like `RewriteRule`s, which can become quite complex in the marketing-inspired e-commerce world?

Walking the web, I found these approaches to the problem:

Environment variables

This is something like the „golden path“ for container configuration management. The way that works is, that you specify the values of environment variables, which are transported into the container and evaluated there. For example, the **TeamCity-Agent image** does that by using `SERVER_URL` to specify the URL to the teamcity server.

That approach is *okay* for simple configurations, but useless for advanced configurations with complex values.

Baked configurations

Another approach builds a new docker image with the configuration files baked into the image itself. This results in a very static configuration. Configuration changes require the complete rebuild and publish of the image. Also, this method needs a private image registry which, at least in case of Kubernetes, is **tied to the cloud provider you’re using**.

Scheduler-centric configuration

This involves using things like Kubernetes‘ **ConfigMap** or Docker Swarm mode’s **Configs**, that injects defined configurations as files into the container. This makes the code for clusters really hard to maintain and is dependent on the used scheduler solution.

Configuration storage

Then, there are things like **etcd** and **Consul**, which can store configuration in a distributed database across your nodes. Obviously, this makes the setup more complex and also requires a solution to fetch these values and use them inside the container. Also, managing these configurations, especially with complex values, can be cumbersome.

Configuration volumes

Docker can **manage data volumes** outside the scope of the container and mount them during runtime. So the configuration can be managed separately from the container and the configuration volumes can also be shared across containers (say, for load balancer nodes which share the same configuration).

I find this solution the best, as it is the most flexible to work with.

Why I care

However, even with configuration volumes you'll have to generate the configuration somehow. Alexey Melezhik created **Sparrowdo** for this task, which is nice, but I came up with another idea.

I did tell you about **SOCKO!**, right? SOCKO! is a configuration file generator, but with hierarchical capabilities. With SOCKO! you'll only have to set up a skeleton of your configuration and can specify the details in nodes of a hierarchy (imagine a directory tree). You then generate the configuration files with one of these node as a target. Every snippet (called a *cartridge*), that is found inside the node is used. If no matching one is found, the lower branches of the hierarchy are searched for them and so on.

This way you can break up your configuration into logical bits and avoid code duplication. There are other features currently available involving exchanging complete files and filling directories.

SOCKO! is currently undergoing a **rewrite as SOCKO 2.0.0**, which will result in a complete and extensible framework. While elaborating the thoughts of this post, I quickly wrapped up a Docker image with the current SOCKO 2.0.0 code, that you can effectively use for container configuration management.

Simply create a configuration volume for your container and let SOCKO fill it:

```
docker volume create loadbalancer-config
dockr run -v loadbalancer-config:/output -v `pwd`:/socko run dodevops/socko generate
--input /socko/input --hierarchy /socko/hierarchy --output /output
```

This will take the input and hierarchy directory from your current working directory and write the output into your configuration volume.

For more details on how SOCKO works, refer to the (alpha) **README** in the 2.0.0 branch.