THE DEVOPS 2.1 TOOLKIT

BUILDING, TESTING, DEPLOYING, AND MONITORING SERVICES INSIDE DOCKER SWARM CLUSTERS

DOCKER SWARM

VIKTOR FARCIC

The DevOps 2.1 Toolkit: Docker Swarm

Building, testing, deploying, and monitoring services inside Docker Swarm clusters

Viktor Farcic

This book is for sale at http://leanpub.com/the-devops-2-1-toolkit

This version was published on 2017-01-22

Tweet This Book!

Please help Viktor Farcic by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought The DevOps 2.1 Toolkit by @vfarcic

The suggested hashtag for this book is #devops2book.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#devops2book

Contents

Preface

At the beginning of 2016, I published **The DevOps 2.0 Toolkit**[1]. It took me a long time to finish it. Much longer than I imagined.

I started by writing blog posts in TechnologyConversations.com[2]. They become popular and I received a lot of feedback. Through them, I clarified the idea behind the book. The goal was to provide a guide for those who want to implement DevOps practices and tools. At the same time, I did not want to write a material usable to any situation. I wanted to concentrate only on people that truly want to implement the latest and greatest practices. I hoped to make it go beyond the "traditional" DevOps. I wished to show that the DevOps movement matured and evolved over the years and that we needed a new name. A reset from the way DevOps is implemented in some organizations. Hence the name, *The DevOps 2.0 Toolkit*[3].

As any author will tell you, technical books based mostly on hands-on material do not have a long time span. Technology changes ever so quickly and we can expect tools and practices that are valid today to become obsolete a couple of years afterward. I expected *The DevOps 2.0 Toolkit* to be a reference for two to three years (not more). After all, how much can things change in one year? Well, Docker proved me wrong. A lot changed in only six months since I made the book public. The new Swarm was released. It is now part of Docker Engine v1.12+. Service discovery is bundled inside it. Networking was greatly improved with load balancing and routing mesh. The list can go on for a while. The release 1.12 is, in my opinion, the most significant release since the first version that went public.

I remember the days I spent together with Docker engineers in Seattle during *DockerCon 2016*. Instead of attending the public sessions, I spent four days with them going through the features that will be released in version 1.12 and the roadmap beyond it. I felt I understood all the technical concepts and features behind them. However, a week later, when I went back home and started "playing" with the new *Docker Swarm Mode*, I realized that my brain was still wired to the way the things were working before. Too many things changed. Too many new possibilities emerged. It took a couple of weeks until my brain reset. Only then I felt I truly understood the scope of changes they introduced in a single release. It was massive.

In parallel with my discovery of the Swarm Mode, I continued receiving emails from *The DevOps 2.0 Toolkit*[4] readers. They wanted more. They wanted me to cover new topics as well as to go deeper into those already explored. One particular request was repeated over and over. "I want you to go deeper into clustering." Readers wanted to know in more detail how to operate a cluster and how to combine it with continuous deployment. They requested that I explore alternative methods for

[1] https://www.amazon.com/dp/B01BJ4V66M
[2] https://technologyconversations.com/
[3] https://www.amazon.com/dp/B01BJ4V66M
[4] https://www.amazon.com/dp/B01BJ4V66M

zero-downtime deployments, how to monitor the system more efficiently, how to get closer to self-healing systems, and so on. The range of topics they wanted me to cover was massive, and they wanted it as soon as possible.

Hence, I decided to start a new book and combine my amazement with Docker 1.12 with some of the requests from the readers. **The DevOps 2.1 Toolkit** was born, and, more importantly, **The DevOps Toolkit Series** came into existence.

Overview

This book explores the practices and tools required to run a Swarm cluster. We'll go beyond a simple deployment. We'll explore how to create a **continuous deployment** process. We'll set up multiple clusters. One will be dedicated to testing and the other for production. We'll see how to accomplish zero-downtime deployments, what to do in case of a failover, how to run services at scale, how to monitor the systems, and how to make it heal itself. We'll explore the processes that will allow us to run the clusters on a laptop (for demo purposes) as well as on different providers. I'll try to cover as many different hosting solutions as possible. However, time and space are limited, and the one you use (be it public or on-premise) might not be one of them. Fear not! The processes, the tools, and the practices can be easily applied to almost any. As always, this book is very hands-on oriented, but the goal is not to master a particular set of tools but to learn the logic behind them so that you can apply it to your job no matter whether your choice ends up being different.

This book does not make *The DevOps 2.0 Toolkit*[5] obsolete. I think that the logic behind it will continue being valid for a while longer. **The DevOps 2.1 Toolkit** builds on top of it. It goes deeper into some of the subjects explored before and explains others that could not fit in *2.0*. While I did my best to write this book in a way that prior knowledge is not required, I strongly recommend starting with *The DevOps 2.0 Toolkit*[6]. The first book in the series sets the stage for this one as well as those that will come afterward. Please consider this book as the second episode in *The DevOps Toolkit Series*.

The first few chapters might, on the first look, seem repetitive to the readers of *The DevOps 2.0 Toolkit*. Never the less, they are important because they unravel some of the problems we'll face when working inside a cluster. They set the stage for the chapters that follow. While you might be tempted to skip them, my recommendation is to read everything. As in any good story, the start sets the stage, the middle develops the story, and the end reveals an unexpected outcome. Some of the theory will be the same as described in *The DevOps 2.0 Toolkit*. In such cases, I will clearly indicate it and let you choose whether to skip it or read it as a way to refresh your memory.

While there will be a lot of theory, this is a hands-on book. You won't be able to complete it by reading it in a metro on the way to work. You'll have to read this book while in front of a computer getting your hands dirty. Eventually, you might get stuck and in need of help. Or you might want to write a review or comment on the book's content. Please post your thoughts on The DevOps 2.1 Toolkit channel in Disqus[7]. If you prefer a more direct communication, please join the DevOps20[8] Slack channel. All the books I wrote are very dear to me, and I want you to have a good experience reading them. Part of that experience is the option to reach out to me. Don't be shy.

[5]https://www.amazon.com/dp/B01BJ4V66M

[6]https://www.amazon.com/dp/B01BJ4V66M

[7]https://disqus.com/home/channel/thedevops21toolkit/

[8]http://slack.devops20toolkit.com/

Please note that this, just as the previous book, is self-published. I believe that having no intermediaries between the writer and the reader is the best way to go. It allows me to write faster, update the book more frequently, and have a more direct communication with you. Your feedback is part of the process. No matter whether you purchased the book while only a few or all chapters were written, the idea is that it will never be truly finished. As time passes, it will require updates so that it is aligned with the change in technology or processes. When possible, I will try to keep it up to date and release updates whenever that makes sense. Eventually, things might change so much that updates are not a good option anymore, and that will be a sign that a whole new book is required. **As long as I continue getting your support, I will keep writing**.

Audience

This book is for professionals interested in the full microservices lifecycle combined with continuous deployment and containers. Due to the very broad scope, target audience could be *architects* who want to know how to design their systems around microservices. It could be *DevOps* wanting to know how to apply modern configuration management practices and continuously deploy applications packed in containers. It is for *developers* who would like to take the process back into their hands as well as for *managers* who would like to gain a better understanding of the process used to deliver software from the beginning to the end. We'll speak about scaling and monitoring systems. We'll even work on the design (and implementation) of self-healing systems capable of recuperation from failures (be it of hardware or software nature). We'll deploy our applications continuously directly to production without any downtime and with the ability to rollback at any time.

This book is for *everyone wanting to know more about the software development lifecycle* starting from requirements and design, through development and testing all the way until deployment and post-deployment phases. We'll create the processes taking into account best practices developed by and for some of the biggest companies.

About the Author

Viktor Farcic is a Senior Consultant at CloudBees[9], a member of the Docker Captains[10] group, and books author.

He coded using a plethora of languages starting with Pascal (yes, he is old), Basic (before it got Visual prefix), ASP (before it got .Net suffix), C, C++, Perl, Python, ASP.Net, Visual Basic, C#, JavaScript, Java, Scala, etc. He never worked with Fortran. His current favorite is Go.

His big passions are Microservices, Continuous Deployment and Test-Driven Development (TDD).

He often speaks at community gatherings and conferences.

He wrote The DevOps 2.0 Toolkit[11] and Test-Driven Java Development[12].

His random thoughts and tutorials can be found in his blog TechnologyConversations.com[13].

[9]https://www.cloudbees.com/
[10]https://www.docker.com/community/docker-captains
[11]https://www.amazon.com/dp/B01BJ4V66M
[12]https://www.packtpub.com/application-development/test-driven-java-development
[13]https://technologyconversations.com/

Continuous Integration With Docker Containers

It is paradoxical, yet true, to say, that the more we know, the more ignorant we become in the absolute sense, for it is only through enlightenment that we become conscious of our limitations. Precisely one of the most gratifying results of intellectual evolution is the continuous opening up of new and greater prospects.

– Nikola Tesla

To fully understand the challenges and benefits that Docker Swarm brings, we need to start from the beginning. We need to go back to a code repository and decide how are we going to build, test, run, update, and monitor the services we're developing. Even though the objective is to implement continuous deployment to a Swarm cluster, we need to step back and explore *continuous integration (CI)* first. The steps we'll define for the CI process will dictate how we proceed towards *continuous delivery (CD)*, from there towards *continuous deployment (CDP)*, and, finally, how we ensure that our services are monitored and able to self-heal. This chapter explores continuous integration as a prerequisite for the more advanced processes.

A note to *The DevOps 2.0 Toolkit* readers

The text that follows is identical to the one published in *The DevOps 2.0 Toolkit*. If it is still fresh in your mind, feel free to jump to the Defining a Fully Dockerized Manual Continuous Integration Flow sub-chapter. Since I wrote the *2.0*, I discovered a few better ways to implement CI processes. I hope you'll benefit from this chapter even if you consider yourself a veteran CI practitioner.

To understand *continuous deployment* we should first define its predecessors, *continuous integration* and *continuous delivery*..

Integration phase of a project development tended to be one of the most painful stages of software development life-cycle. We would spend weeks, months or even years working in separate teams dedicated to separate applications and services. Each of those teams would have their set of requirements and tried their best to meet them. While it wasn't hard to periodically verify each of those applications and services in isolation, we all dreaded the moment when team leads would decide that the time has come to integrate them into a unique delivery. Armed with the experience from previous projects, we knew that integration would be problematic. We knew that we would discover problems, unmet dependencies, interfaces that do not communicate with each other correctly and that managers will get disappointed, frustrated, and nervous. It was not uncommon to spend weeks or even months in this phase. The worse part of all that was that a bug found during the integration phase could mean going back and redoing days or weeks worth of work. If someone asked me how I felt about integration back then, I'd say that it was closest I could get to becoming permanently depressed. Those were different times. We thought that was the "right" way to develop applications.

A lot changed since then. *Extreme Programming (XP)* and other agile methodologies became familiar, automated testing became frequent, and continuous integration started to take ground. Today we know that the way we developed software back then was wrong. The industry moved a long way since those days.

Continuous integration (CI) usually refers to integrating, building, and testing code within the development environment. It requires developers to integrate code into a shared repository often. How often is often can be interpreted in many ways and it depends on the size of the team, the size of the project and the number of hours we dedicate to coding. In most cases, it means that coders either push directly to the shared repository or merge their code with it. No matter whether we're pushing or merging, those actions should, in most cases, be done at least a couple of times a day. Getting code to the shared repository is not enough, and we need to have a pipeline that, as a minimum, checks out the code and runs all the tests related, directly or indirectly, to the code corresponding to the repository. The result of the execution of the pipeline can be either *red* or *green*. Something failed, or everything was run without any problems. In the former case, minimum action would be to notify the person who committed the code.

The continuous integration pipeline should run on every commit or push. Unlike continuous delivery, continuous integration does not have a clearly defined goal of that pipeline. Saying that one application integrates with others does not tell us a lot about its production readiness. We do not know how much more work is required to get to the stage when the code can be delivered to production. All we are striving for is the knowledge that a commit did not break any of the existing tests. Nevertheless, CI is a vast improvement when done right. In many cases, it is a very hard practice to implement, but once everyone is comfortable with it, the results are often very impressive.

Integration tests need to be committed together with the implementation code, if not before. To gain maximum benefits, we should write tests in *test-driven development (TDD)* fashion. That way, not only that tests are ready for commit together with implementation, but we know that they are not faulty and would not pass no matter what we do. There are many other benefits TDD brings to the table and, if you haven't already, I strongly recommend to adopt it. You might want to consult the Test-Driven Development[14] section of the Technology Conversations[15] blog.

Tests are not the only CI prerequisite. One of the most important rules is that when the pipeline fails, fixing the problem has higher priority than any other task. If this action is postponed, next executions of the pipeline will fail as well. People will start ignoring the failure notifications and, slowly, CI process will begin losing its purpose. The sooner we fix the problem discovered during the

[14]http://technologyconversations.com/category/test-driven-development/
[15]http://technologyconversations.com/

execution of the CI pipeline, the better we are. If corrective action is taken immediately, knowledge about the potential cause of the problem is still fresh (after all, it's been only a few minutes between the commit and the failure notification) and fixing it should be trivial.

Defining a Fully Dockerized Manual Continuous Integration Flow

Every continuous integration process starts with a code that is checked out from a repository. We'll use the GitHub repository vfarcic/go-demo[16] throughout the book. It contains the code of the service we'll use throughout the book. The service is written in Go[17]. Fear not! Even though I consider it one of the best currently available languages, you will not be required to learn *Go*. We'll use the *go-demo* service only as a demonstration of the processes explained throughout the book. Even though I strongly recommend learning *Go*, the book does not assume any knowledge of the language. All the examples will be programming language agnostic.

All the commands from this chapter are available in the 01-continuous-integration.sh[18] Gist.

Let's get going and check out the *go-demo* code.

A note to Windows users

Please make sure that your Git client is configured to check out the code *AS-IS*. Otherwise, Windows might change carriage returns to the Windows format.

```
git clone https://github.com/vfarcic/go-demo.git

cd go-demo
```

Some of the files will be shared between the host file system and Docker Machines we'll start creating soon. Docker Machine makes the whole directory that belongs to the current user available inside the VM. Therefore, please make sure that the code is checked out inside one of the user's sub-folders.

Now that we have the code checked out from the repository, we need a server that we'll use to build and run tests. For now, we'll use *Docker Machine*, since it provides an easy way to create a "Docker ready" VMs on our laptops.

[16] https://github.com/vfarcic/go-demo
[17] https://golang.org/
[18] https://gist.github.com/vfarcic/886ae97fe7a98864239e9c61929a3c7c

Docker Machine[19] is a tool that lets you install Docker Engine on virtual hosts, and manage the hosts with docker-machine commands. You can use Machine to create Docker hosts on your local Mac or Windows box, on your company network, in your data center, or on cloud providers like AWS or DigitalOcean.

Using docker-machine commands, you can start, inspect, stop, and restart a managed host, upgrade the Docker client and daemon, and configure a Docker client to talk to your host.

Machine was the only way to run Docker on Mac or Windows previous to Docker v1.12. Starting with the beta program and Docker v1.12, Docker for Mac and Docker for Windows are available as native apps and the better choice for this use case on newer desktops and laptops. I encourage you to try out these new apps. The installers for Docker for Mac and Docker for Windows include Docker Machine, along with Docker Compose.

The examples that follow assume that you have Docker Machine[20] version v0.9+ that includes Docker Engine[21] v1.13+. The installation instructions can be found in the Install Docker Machine[22] page.

A note to Windows users

The recommendation is to run all the examples from *Git Bash* (installed through *Docker Toolbox* as well as *Git*). That way the commands you'll see throughout the book will be same as those that should be executed on *OS X* or any *Linux* distribution.

Let's create our first server called *go-demo*.

```
docker-machine create -d virtualbox go-demo
```

A note to Windows users

If you're using *Docker for Windows* instead of *Docker Toolbox*, you will need to change the driver from `virtualbox` to `hyperv`. The problem is that `hyperv` does not allow mounting host volumes, so it is still highly recommended to use *Docker Toolbox* when working with *Docker Machine*. The reason behind the choice of running *Docker* inside *Docker Machines* instead natively on the host lies in the need to run a cluster (coming in the next chapter). *Docker Machine* is the easiest way to simulate a multi-node cluster.

The command should be self-explanatory. We specified *virtualbox* as the driver (or `hyperv` if you're running *Docker for Windows*) and named the machine *go-demo*.

Now that the machine is running, we should instruct our local Docker Engine to use it.

[19]https://docs.docker.com/machine/overview/
[20]https://www.docker.com/products/docker-machine
[21]https://www.docker.com/products/docker-engine
[22]https://docs.docker.com/machine/install-machine/

```
docker-machine env go-demo
```

The `docker-machine env go-demo` command outputs environment variables required for the local engine to find the server we'd like to use. In this case, the remote engine is inside the VM we created with the `docker-machine create` command.

The output is as follows.

```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/vfarcic/.docker/machine/machines/go-demo"
export DOCKER_MACHINE_NAME="go-demo"
```

We can envelop the `env` command into an `eval` that will evaluate the output and, in this case, create the environment variables.

```
eval $(docker-machine env go-demo)
```

From now on, all the Docker commands we execute locally will be channeled to the engine running inside the `go-demo` machine.

Now we are ready to run the first two steps in the CI flow. We'll execute unit tests and build the service binary.

Running Unit Tests And Building Service Binary

We'll use Docker Compose[23] for the CI flow. As you will see soon, Docker Compose has little, if any, value when operating the cluster. However, for operations that should be performed on a single machine, Docker Compose is still the easiest and the most reliable way to go.

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your application's services. Then, using a single command, you create and start all the services from your configuration. Compose is great for development, testing, and staging environments, as well as CI workflows.

The repository that we cloned earlier, already has all the services we'll need defined inside the docker-compose-test-local.yml[24].

Let's take a look at the content of the docker-compose-test-local.yml[25] file.

[23]https://docs.docker.com/compose/
[24]https://github.com/vfarcic/go-demo/blob/master/docker-compose-test-local.yml
[25]https://github.com/vfarcic/go-demo/blob/master/docker-compose-test-local.yml

```
cat docker-compose-test-local.yml
```

The service we'll use for our unit tests is called `unit`. It is as follows.

```
  unit:
    image: golang:1.6
    volumes:
      - .:/usr/src/myapp
      - /tmp/go:/go
    working_dir: /usr/src/myapp
    command: bash -c "go get -d -v -t && go test --cover -v ./... && go build -v\
 -o go-demo"
```

It is a relatively simple definition. Since the service is written in *Go*, we are using the `golang:1.6` image.

Next, we are exposing a few volumes. Volumes are directories that are, in this case, mounted on the host. They are defined with two arguments. The first argument is the path to the host directory while the second represents a directory inside the container. Any file already inside the host's directory will be available inside the container and vice versa.

The first volume is used for the source files. We are sharing the current host directory (`.`) with the container directory `/usr/src/myapp`. The second volume is used for *Go* libraries. Since we want to avoid downloading all the dependencies every time we run unit tests, they will be stored inside the host's directory `/tmp/go`. That way, dependencies will be downloaded only the first time we run the service.

Volumes are followed with the `working_dir` instruction. When the container is run, it will use the specified value as the starting directory.

Finally, we are specifying the command we want to run inside the container. I won't go into details since they are specific to *Go*. In short, we download all the dependencies (`go get -d -v -t`), run unit tests (`go test --cover -v ./...`), and build the *go-demo* binary (`go build -v -o go-demo`). Since the directory with the source code is mounted as a volume, the binary will be stored on the host and available for later use.

With this single Compose service, we defined two steps of the CI flow. It contains unit tests and build of the binary.

Please note that even though we run the service called `unit`, the real purpose of this CI step is to run any type of tests that do not require deployment. Those are the tests we can execute before we build the binary and, later on, Docker images.

Let's run it.

```
docker-compose \
    -f docker-compose-test-local.yml \
    run --rm unit
```

A note to Windows users

Windows does not yet support running containers in the interactive mode. You will have to add the `-d` argument to all `docker-compose run` commands. The downside is that `-d` will run containers in detached mode. For long running processes, that is the preferable way of running containers. However, when using containers to run tests, we want to see the output and the exit code. Since those features are available only in the interactive mode which is not yet supported on Windows, you will have to check the logs manually. Having all that into account, the command from above should be as follows in Windows.

```
docker-compose -f docker-compose-test-local.yml run -d --rm unit
```

It should be followed by the `docker logs` command.

```
docker logs godemo_unit_run_1
```

The format of the container names created with Docker Compose `run` command is `<PROJECT_NAME>_<SERVICE_NAME>_run_<INDEX>`. Please note that the project name defaults to the directory Compose file resides in. Underscores (_), dashes (-), and spaces (* *) are removed. Index is incremental. The first run will be index `1`, the second index `2`, and so on. When you execute the `docker-compose run` command in detached mode, the name of the container is output on the screen.

Since the container will run in detached mode, we cannot know whether it finished running or not. The `docker ps` command will show you whether the container is still running or not.

A note to Windows users

You might experience a problem with volumes not being mapped correctly. If you see an *Invalid volume specification* error, please export the environment variable `COMPOSE_CONVERT_WINDOWS_PATHS` set to `0`.

```
export COMPOSE_CONVERT_WINDOWS_PATHS=0
```

If that fixed the problem with volumes, please make sure that the variable is exported every time you run `docker-compose`.

We specified that Compose should use `docker-compose-test-local.yml` file (default is *docker-compose.yml*) and run the service called `unit`. The `--rm` argument means that the container should be removed once it stops. The `run` command should be used for services that are not meant to run forever. It is perfect for batch jobs and, as in this case, for running tests.

As you can see from the output, we pulled the `golang` image, downloaded service dependencies, successfully ran the tests, and built the binary.

We can confirm that the binary is indeed built and available on the host by listing the files in the current directory. For brevity, we'll filter the result.

```
ls -l *go-demo*
```

Now that we passed the first round of tests and have the binary, we can proceed and build the Docker images.

Building Service Images

Docker images are built through a definition stored in a *Dockerfile*. With few exceptions, it takes a similar approach as if we would define a simple script. We will not explore all the options we can use when defining a Dockerfile, but only those used for the *go-demo* service. Please consult the Dockerfile reference[26] page for more info.

The *go-demo* Dockerfile[27] is as follows.

```
FROM alpine:3.4
MAINTAINER Viktor Farcic <viktor@farcic.com>

RUN mkdir /lib64 && ln -s /lib/libc.musl-x86_64.so.1 /lib64/ld-linux-x86-64.so.2

EXPOSE 8080
ENV DB db
CMD ["go-demo"]
HEALTHCHECK --interval=10s CMD wget -qO- localhost:8080/demo/hello

COPY go-demo /usr/local/bin/go-demo
RUN chmod +x /usr/local/bin/go-demo
```

Each of the statements will be built as a separate image. A container is a collection of images stacked one on top of the other.

Every Dockerfile starts with the `FROM` statement. It defines the base image that should be used. In most cases, my preference is to use `alpine` Linux. With its size being around 2MB it is probably the smallest distribution we can use. That is aligned with the idea that containers should have only things that are needed and avoid any extra overhead.

[26]https://docs.docker.com/engine/reference/builder/
[27]https://github.com/vfarcic/go-demo/blob/master/Dockerfile

`MAINTAINER` is for informational purposes only.

The `RUN` statement executes any command set as its argument. I won't explain this one since it is very specific to the service we're building.

The `EXPOSE` statement defines the port the service will be listening to. It is followed by the definition of the environment variable `DB` that tells the service the address of the database. The default value is `db` and, as you'll see soon, it can be changed at runtime. The `CMD` statement represents the command that will be run when containers start.

The `HEALTHCHECK` instruction tells Docker how to test a container to check that it is still working. This can detect cases such as a web server that is stuck in an infinite loop and unable to handle new connections, even though the server process is still running. When a container has a healthcheck specified, it has a health status in addition to its normal status. This status is initially starting. Whenever a health check passes, it becomes healthy (from whatever state it was previously in). After a certain number of consecutive failures, it becomes unhealthy.

In our case, the healthcheck will be executed every ten seconds. The command sends a simple request to one of the API endpoints. If the service responds with status `200`, the `wget` command will return `0` and Docker will consider the service healthy. Any other response will be considered as unhealthy and Docker Engine will perform certain actions to fix the situation.

Finally, we copy the `go-demo` binary from the host to the `/usr/local/bin/` directory inside the image and give it executable permissions with the `chmod` command.

To some, the order of the statements might not look logical. However, there is a good reason behind such declarations and their order. Those that are less likely to change are defined before those that are prone to changes. Since `go-demo` will be a new binary every time we build the images, it is defined last.

The reasons behind such order lie in the way Docker Engine creates images. It starts from the top-most definition and checks whether it changed since the last time the build was run. If it didn't, it moves to the next statement. As soon as it finds a statement that would produce a new image, it, and all the statements below it are built into Docker images. By placing those that are less likely to change closer to the top, we can reduce the build time, disk usage, and bandwidth.

Now that we understand the *Dockerfile* behind the *go-demo* service, we can build the images.

The command is very straightforward.

```
docker build -t go-demo .
```

As an alternative, we can define build arguments inside a Docker Compose file. The service defined in docker-compose-test-local.yml[28] is as follows.

[28]https://github.com/vfarcic/go-demo/blob/master/docker-compose-test-local.yml

```
  app:
    build: .
    image: go-demo
```

In both cases, we specified that the current directory should be used for the build process (`.`) and that the name of the image is `go-demo`.

We can run the build through Docker compose with the command that follows.

```
docker-compose \
    -f docker-compose-test-local.yml \
    build app
```

We'll use the latter method throughout the rest of the book.

We can confirm that the image was indeed built, by executing the `docker images` command.

```
docker images
```

The output is as follows.

```
REPOSITORY TAG    IMAGE ID     CREATED        SIZE
go-demo    latest 5e90126bebf1 49 seconds ago 23.61 MB
golang     1.6    08a89f0a4ee5 11 hours ago   744.2 MB
alpine     latest 4e38e38c8ce0 9 weeks ago    4.799 MB
```

As you can see, `go-demo` is one of the images we have inside the server.

Now that the images are built, we can run staging tests that depend on the service and its dependencies to be deployed on a server.

Running Staging Tests

Please note that the real purpose of this step in the CI flow is to run the tests that require the service and its dependencies to be running. Those are still not integration tests that require production or production-like environment. The idea behind those tests is to run the service together with its direct dependencies, run the tests, and, once they're finished, remove everything and free the resources for some other task. Since these are still not integration tests, some, if not all, dependencies can be mocks.

Due to the nature of these tests, we need to split the task into three actions.

1. Run the service and all the dependencies

2. Run the tests
3. Destroy the service and all the dependencies

The dependencies are defined as the `staging-dep` service inside the docker-compose-test-local.yml[29] file. The definition is as follows.

```
  staging-dep:
    image: go-demo
    ports:
      - 8080:8080
    depends_on:
      - db

  db:
    image: mongo:3.2.10
```

The image is `go-demo`, and it exposes the port `8080` (both on the host and inside the container). It depends on the service `db` which is a `mongo` image. Services defined as `depends_on` will be run before the service that defines the dependency. In other words, if we run the `staging-dep` target, Compose will run the `db` first.

Let's run the dependencies.

```
docker-compose \
    -f docker-compose-test-local.yml \
    up -d staging-dep
```

Once the command is finished, we will have two containers running (*go-demo* and *db*). We can confirm that by listing all the processes.

```
docker-compose \
    -f docker-compose-test-local.yml \
    ps
```

The output is as follows.

[29] https://github.com/vfarcic/go-demo/blob/master/docker-compose-test-local.yml

```
Name                 Command              State Ports
-----------------------------------------------------------------------
godemo_db_1          /entrypoint.sh mongod Up    27017/tcp
godemo_staging-dep_1 go-demo              Up    0.0.0.0:8080->8080/tcp
```

Now that the service and the database it depends on are running, we can execute the tests. They are defined as the service `staging`. The definition is as follows.

```
  staging:
    extends:
      service: unit
    environment:
      - HOST_IP=localhost:8080
    network_mode: host
    command: bash -c "go get -d -v -t && go test --tags integration -v"
```

Since the definition of the staging tests is very similar to those we run as unit tests, the `staging` service extends `unit`. By extending a service, we inherit its full definition. Further on, we defined an environment variable `HOST_IP`. The tests code uses that variable to determine the location of the service under test. In this case, since the `go-demo` service is running on the same server as tests, the IP is server's `localhost`. Since, by default, localhost inside a container is not the same as the one on the host, we had to define `network_mode` as `host`. Finally, we defined the command that should be executed. It will download tests dependencies (`go get -d -v -t`) and run the tests (`go test --tags integration -v`).

Let's run it.

```
docker-compose \
    -f docker-compose-test-local.yml \
    run --rm staging
```

All the tests passed, and we are one step closer to the goal of having full confidence that the service is indeed safe to be deployed to production.

We don't have any use for keeping the service and the database running so let's remove them and free the resources for some other task.

```
docker-compose \
    -f docker-compose-test-local.yml \
    down
```

The `down` command stops and removes all services defined in that Compose file. We can verify that by running the `ps` command.

```
docker-compose \
    -f docker-compose-test-local.yml \
    ps
```

The output is as follows.

```
Name   Command   State   Ports
------------------------------
```

There is only one thing missing for the CI flow to be complete. At this moment we have the `go-demo` image that is usable only inside the `go-demo` server. We should store it in a registry so that it can be accessed from other servers as well.

Pushing Images To The Registry

Before we push our *go-demo* image, we need a place to push to. Docker offers multiple solutions that act as a registry. We can use Docker Hub[30], Docker Registry[31], and Docker Trusted Registry[32]. On top of those, there are many other solutions from third party vendors.

Which registry should we use? Docker Hub requires a username and password, and I do not trust you enough to provide my own. One of the goals I defined before I started working on the book is to use only open source tools so Docker Trusted Registry, while being an excellent choice under different circumstances, is also not suitable. The only option left (excluding third party solutions), is Docker Registry[33].

The registry is defined as one of the services inside the docker-compose-local.yml[34] Compose file. The definition is as follows.

```
  registry:
    container_name: registry
    image: registry:2.5.0
    ports:
      - 5000:5000
    volumes:
      - .:/var/lib/registry
    restart: always
```

[30]https://hub.docker.com/
[31]https://docs.docker.com/registry/
[32]https://docs.docker.com/docker-trusted-registry/
[33]https://docs.docker.com/registry/
[34]https://github.com/vfarcic/go-demo/blob/master/docker-compose-local.yml

We set `registry` as an explicit container name, specified the `image`, and opened the port `5000` (both on the host and inside the container).

Registry stores the images inside the `/var/lib/registry` directory, so we mounted it as a volume on the host. That way, data will not be lost if the container fails. Since this is a production service that could be used by many, we defined that it should always be restarted on failure.

Let's run it.

```
docker-compose \
    -f docker-compose-local.yml \
    up -d registry
```

Now that we have the registry, we can do a dry-run. Let's confirm that we can pull and push images to it.

```
docker pull alpine

docker tag alpine localhost:5000/alpine

docker push localhost:5000/alpine
```

Docker uses a naming convention to decide where to pull and push images from. If the name is prefixed with an address, the engine will use it to determine the location of the registry. Otherwise, it assumes that we want to use Docker Hub. Therefore, the first command pulled the `alpine` image from Docker Hub.

The second command created a tag of the alpine image. The tag is a combination of the address of our registry (`localhost:5000`) and the name of the image. Finally, we pushed the alpine image to the registry running on the same server.

Before we start using the registry in a more serious fashion, let's confirm that the images are indeed persisted on the host.

```
ls -1 docker/registry/v2/repositories/alpine/
```

The output is as follows.

```
_layers
_manifests
_uploads
```

I won't go into details what each of those sub-directories contains. The important thing to note is that registry persists the images on the host so no data will be lost if it fails or, in this case, even if we destroy the VM since that Machine directory is mapped to the same directory on our laptop.

We were a bit hasty when we declared that this registry should be used in production. Even though data is persisted, if the whole VM crashes, there would be a downtime until someone brings it up again or creates a new one. Since one of the goals is to avoid downtime whenever possible, later on, we should look for a more reliable solution. The current setup should do for now.

Now we are ready to push the *go-demo* image to the registry.

```
docker tag go-demo localhost:5000/go-demo:1.0

docker push localhost:5000/go-demo:1.0
```

As with the *alpine* example, we tagged the image with the registry prefix and pushed it to the registry. We also added a version number (`1.0`).

The push was the last step in the CI flow. We run unit tests, built the binary, built the Docker image, ran staging tests, and pushed the image to the registry. Even though we did all those things, we are not yet confident that the service is ready for production. We never tested how it would behave when deployed to a production (or production-like) cluster. We did a lot, but not enough.

If CI were our final objective, this would be the moment when manual validations should occur. While there is a lot of value in manual labor that requires creativity and critical thinking, we cannot say the same for repetitive tasks. Tasks required for converting this continuous integration flow into continuous delivery and, later on, deployment are, indeed repetitive.

> We have the CI process done, and it is time to do the extra mile and convert it into continuous delivery.

Before we move into the steps required for the continuous integration process to become continuous delivery, we need to take a step back and explore cluster management. After all, in most cases, there is no production environment without a cluster.

We'll destroy the VMs at the end of each chapter. That way, you can come back to any of part of the book and do the exercises without the fear that you might need to do some steps from one of the earlier chapters. Also, such a procedure will force us to repeat a few things. Practice makes perfect. To reduce your waiting times, I did my best to keep things as small as possible and keep download times to a minimum.

```
docker-machine rm -f go-demo
```

The next chapter is dedicated to the setup and operation of a Swarm cluster.

Setting Up And Operating a Swarm Cluster

Organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations

– M. Conway

Many will tell you that they have a *scalable system*. After all, scaling is easy. Buy a server, install WebLogic (or whichever other monster application server you're using) and deploy your applications. Then wait for a few weeks until you discover that everything is so "fast" that you can click a button, have some coffee, and, by the time you get back to your desk, the result will be waiting for you. What do you do? You scale. You buy a few more servers, install your monster application servers and deploy your monster applications on top of them. Which part of the system was the bottleneck? Nobody knows. Why did you duplicate everything? Because you must. And then some more time passes, and you continue scaling until you run out of money and, simultaneously, people working for you go crazy. Today we do not approach scaling like that. Today we understand that scaling is about many other things. It's about elasticity. It's about being able to quickly and easily scale and de-scale depending on variations in your traffic and growth of your business, and that you should not go bankrupt during the process. It's about the need of almost every company to scale their business without thinking that IT department is a liability. It's about getting rid of those monsters.

A note to *The DevOps 2.0 Toolkit* readers

The text that follows is identical to the one published in *The DevOps 2.0 Toolkit*. If it is still fresh in your mind, feel free to jump to the Docker Swarm Mode sub-chapter. You'll see that a lot has changed. One of those changes is that the old Swarm running as a separate container is deprecated for *Swarm Mode*. There are many other new things we'll discover along the way.

Scalability

Let us, for a moment take a step back and discuss why we want to scale applications. The main reason is *high availability*. Why do we want high availability? We want it because we want our business to be available under any load. The bigger the load, the better (unless you are under DDoS). It means that our business is booming. With high availability our users are happy. We all want speed, and many of us simply leave the site if it takes too long to load. We want to avoid having outages because every minute our business is not operational can be translated into a money loss. What would you do if an online store is not available? Probably go to another. Maybe not the first time, maybe not the second, but, sooner or later, you would get fed up and switch it for another. We are used to everything being fast and responsive, and there are so many alternatives that we do not think twice before trying something else. And if that something else turns up to be better… One man's loss is another man's gain. Do we solve all our problems with scalability? Not even close. Many other factors decide the availability of our applications. However, scalability is an important part of it, and it happens to be the subject of this chapter.

What is scalability? It is a property of a system that indicates its ability to handle increased load in a graceful manner or its potential to be enlarged as demand increases. It is the capacity to accept increased volume or traffic.

The truth is that the way we design our applications dictates the scaling options available. Applications will not scale well if they are not designed to scale. That is not to say that an application not designed for scaling cannot scale. Everything can scale, but not everything can scale well.

Commonly observed scenario is as follows.

We start with a simple architecture, sometimes with load balancer sometimes without, setup a few application servers and one database. Everything is great, complexity is low, and we can develop new features very fast. The cost of operations is low, income is high (considering that we just started), and everyone is happy and motivated.

Business is growing, and the traffic is increasing. Things are beginning to fail, and performance is dropping. Firewalls are added, additional load balancers are set up, the database is scaled, more application servers are added and so on. Things are still relatively straightforward. We are faced with new challenges, but we can overcome obstacles in time. Even though the complexity is increasing, we can still handle it with relative ease. In other words, what we're doing is still, more or less, the same but bigger. Business is doing well, but it is still relatively small.

And then it happens. The big thing you've been waiting for. Maybe one of the marketing campaigns hit the spot. Maybe there was an adverse change in your competition. Maybe that last feature was indeed a killer one. No matter the reasons, business got a big boost. After a short period of happiness due to this change, your pain increases tenfold. Adding more databases does not seem to be enough. Multiplying application servers does not appear to fulfill the needs. You start adding caching and what not. You start getting the feeling that every time you multiply something, benefits are not equally big. Costs increase, and you are still not able to meet the demand. Database replications are too slow. New application servers do not make such a big difference anymore. Operational costs are increasing faster than you expected. The situation hurts the business and the team. You are starting to realize that the architecture you were so proud of cannot fulfill this increase in load. You cannot split it. You cannot scale things that hurt the most. You cannot start over. All you can do is continue multiplying with ever decreasing benefits of such actions.

The situation described above is quite common. What was good at the beginning, is not necessarily right when the demand increases. We need to balance the need for YAGNI (You Ain't Gonna Need It) principle and the longer term vision. We cannot start with the system optimized for large companies because it is too expensive and does not provide enough benefits when business is small. On the other hand, we cannot lose the focus from one of the primary objectives of any business. We cannot not think about scaling from the very first day. Designing scalable architecture does not mean that we need to start with a cluster of a hundred servers. It does not mean that we have to develop something big and complex from the start. It means that we should start small, but in the way that, when it becomes big, it is easy to scale. While microservices are not the only way to accomplish that goal, they are indeed a good way to approach this problem. The cost is not in development but operations. If operations are automated, that cost can be absorbed quickly and does not need to represent a massive investment. As you already saw (and will continue seeing throughout the rest of the book), there are excellent open source tools at our disposal. The best part of automation is that the investment tends to have lower maintenance cost than when things are done manually.

We already discussed microservices and automation of their deployments on a tiny scale. Now it's time to convert this small scale to something bigger. Before we jump into practical parts, let us explore what are some of the different ways one might approach scaling.

We are often limited by our design and choosing the way applications are constructed limits our choices severely. Although there are many different ways to scale, the most common one is called *Axis Scaling*.

Axis Scaling

Axis scaling can be best represented through three dimensions of a cube; *x-axis*, *y-axis*, and *z-axis*. Each of those dimensions describes a type of scaling.

- X-Axis: Horizontal duplication
- Y-Axis: Functional decomposition
- Z-Axis: Data partitioning

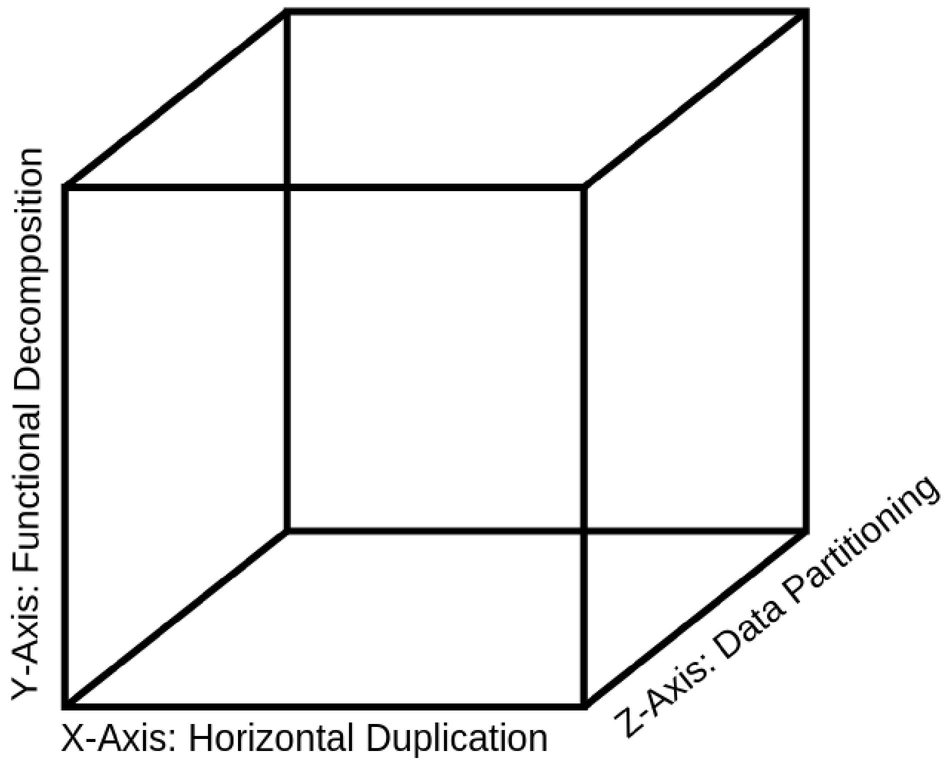

Y-Axis: Functional Decomposition
X-Axis: Horizontal Duplication
Z-Axis: Data Partitioning

Figure 2-1: Scale cube

Let's go through axes, one at a time.

X-Axis Scaling

In a nutshell, *x-axis scaling* is accomplished by running multiple instances of an application or a service. In most cases, there is a load balancer on top that makes sure that the traffic is shared among all those instances. The biggest advantage of x-axis scaling is simplicity. All we have to do is deploy the same application on multiple servers. For that reason, this is the most commonly used type of scaling. However, it comes with its set of disadvantages when applied to monolithic applications.

Having a large application usually requires a big cache that demands heavy usage of memory. When such an application is multiplied, everything is multiplied with it, including the cache. Another, often more important, problem is an inappropriate usage of resources. Performance issues are almost never related to the whole application. Not all modules are equally affected, and, yet, we multiply everything. That means that even though we could be better off by scaling only part of the application that requires such an action, we scale everything. Nevertheless, x-scaling is important no matter the architecture. The major difference is the effect that such a scaling has.

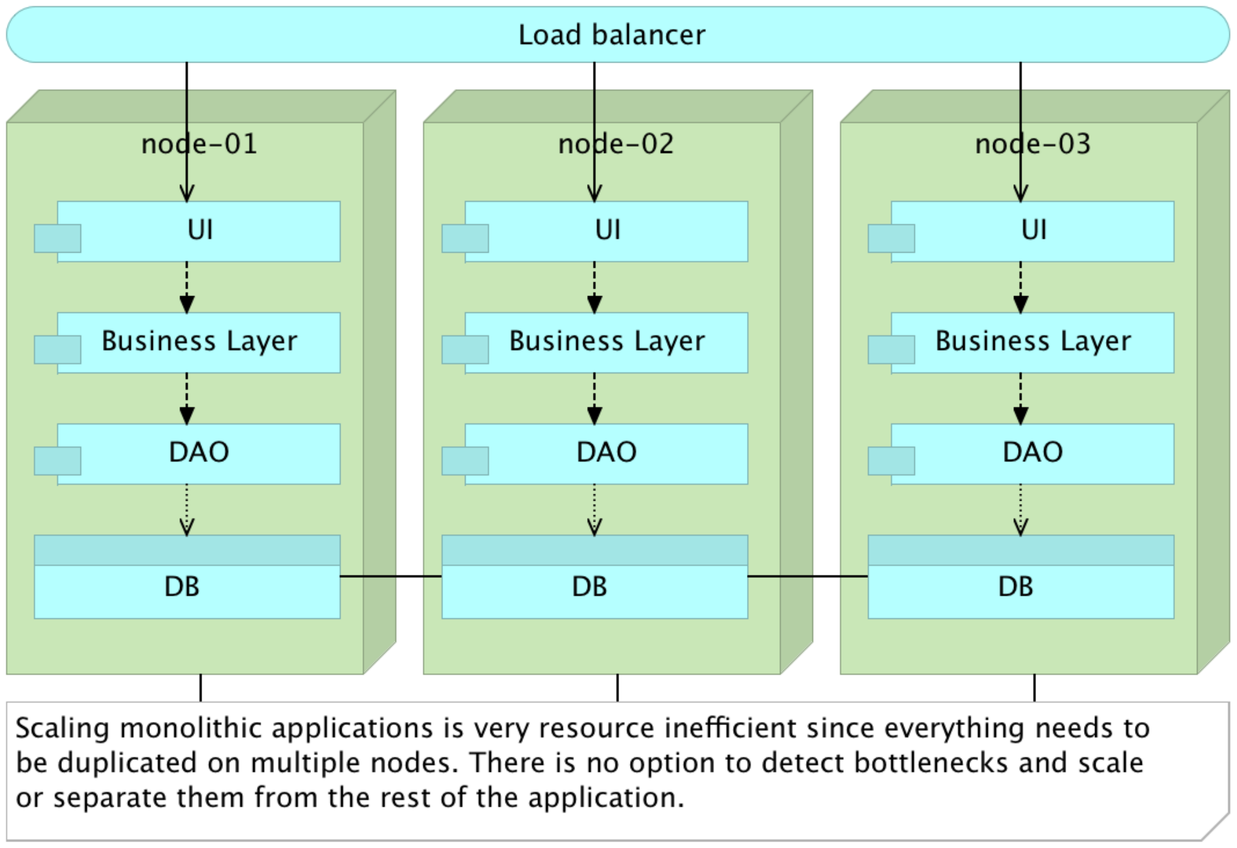

Load balancer
node-01
node-02
node-03
UI
Business Layer
DAO
DB
Scaling monolithic applications is very resource inefficient since everything needs to be duplicated on multiple nodes. There is no option to detect bottlenecks and scale or separate them from the rest of the application.

Figure 2-2: Monolithic application scaled inside a cluster

By using microservices, we are not removing the need for x-axis scaling but making sure that due to their architecture such scaling has more effect than with alternative and more traditional approaches to architecture. With microservices, we have the option to fine-tune scaling. We can have many instances of services that suffer a lot under heavy load and only a few instances of those that are used less often or require fewer resources. On top of that, since they are small, we might never reach a limit of a service. A small service in a big server would need to receive a truly massive amount of traffic before the need for scaling arises. Scaling microservices is more often related to fault tolerance than performance problems. We want to have multiple copies running so that, if one of them dies, the others can take over until recovery is performed.

Y-Axis Scaling

Y-axis scaling is all about decomposition of an application into smaller services. Even though there are different ways to accomplish this decomposition, microservices are probably the best approach we can take. When they are combined with immutability and self-sufficiency, there is indeed no better alternative (at least from the prism of y-axis scaling). Unlike x-axis scaling, the y-axis is not accomplished by running multiple instances of the same application but by having multiple different services distributed across the cluster.

Z-Axis Scaling

Z-axis scaling is rarely applied to applications or services. Its primary and most common usage is among databases. The idea behind this type of scaling is to distribute data among multiple servers thus reducing the amount of work that each of them needs to perform. Data is partitioned and distributed so that each server needs to deal only with a subset of the data. This type of separation is often called sharding, and there are many databases specially designed for this purpose. Benefits of z-axis scaling are most noticeable in I/O and cache and memory utilization.

Clustering

A server cluster consists of a set of connected servers that work together and can be seen as a single system. They are usually connected to a fast local area network (LAN). The significant difference between a cluster and a group of servers is that the cluster acts as a single system trying to provide high availability, load balancing, and parallel processing.

If we deploy applications, or services, to individually managed servers and treat them as separate units, the utilization of resources is sub-optimum. We cannot know in advance which group of services should be deployed to a server and utilize resources to their maximum. Moreover, resource usage tends to fluctuate. While, in the morning, some service might require a lot of memory, during the afternoon that usage might be lower. Having predefined servers prevents us from having elasticity that would balance that usage in the best possible way. Even if such a high level of dynamism is not required, predefined servers tend to create problems when something goes wrong, resulting in manual actions to redeploy the affected services to a healthy node.

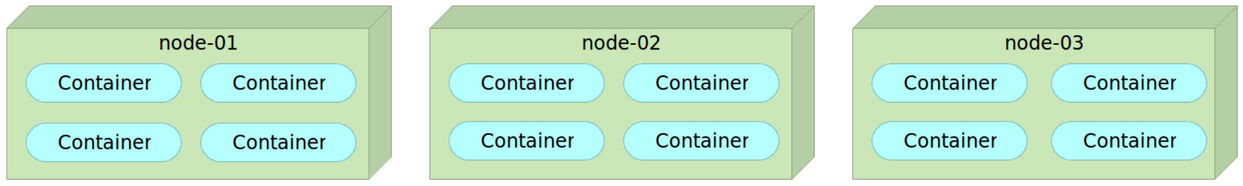


Figure 2-3: Cluster with containers deployed to predefined servers

Real clustering is accomplished when we stop thinking in terms of individual servers and start thinking of a cluster; of all servers as one big entity. That can be better explained if we drop to a bit lower level. When we deploy an application, we tend to specify how much memory or CPU it might need. However, we do not decide which memory slots our application will use nor which CPUs it should utilize. For example, we don't specify that some application should use CPUs 4, 5 and 7. That would be inefficient and potentially dangerous. We only decide that three CPUs are required. The same approach should be taken on a higher level. We should not care where an application or a service will be deployed but what it needs. We should be able to define that the service has certain requirements and tell some tool to deploy it to whichever server in our cluster, as long as it fulfills the needs we have. The best (if not the only) way to accomplish that is to consider the whole cluster as one entity.

We can increase or decrease the capacity of a cluster by adding or removing servers but, no matter what we do, it should still be a single entity. We define a strategy and let our services be deployed somewhere inside the cluster. Those using cloud providers like Amazon Web Services (AWS), Microsoft's Azure and Google Compute Engine (GCE) are already accustomed to this approach, even though they might not be aware of it.

Throughout the rest of this chapter, we'll explore ways to create our cluster and explore tools that can help us with that objective. The fact that we'll be simulating the cluster locally does not mean that the same strategies cannot be applied to public or private clouds and data centers. Quite the opposite.

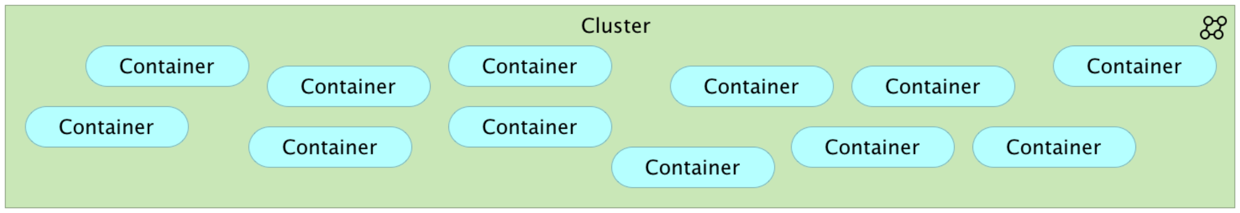


Figure 2-4: Cluster with containers deployed to servers based on a predefined strategy

Docker Swarm Mode

Docker Engine v1.12 was released in July 2016. It is the most significant version since v1.9. Back then, we got Docker networking that, finally, made containers ready for use in clusters. With v1.12, Docker is reinventing itself with a whole new approach to cluster orchestration. Say goodbye to Swarm as a separate container that depends on an external data registry and welcome the **new Docker Swarm** or *Swarm Mode*. Everything you'll need to manage your cluster is now incorporated into Docker Engine. Swarm is there. Service discovery is there. Improved networking is there. That does not mean that we do not need additional tools. We do. The major difference is that Docker Engine now incorporates all the "essential" (not to say minimal) tools we need.

The old Swarm (before Docker v1.12) used **fire-and-forget principle**. We would send a command to Swarm master, and it would execute that command. For example, if we would send it something like `docker-compose scale go-demo=5`, the old Swarm would evaluate the current state of the cluster, discover that, for example, only one instance is currently running, and decide that it should run four more. Once such a decision is made, the old Swarm would send commands to Docker Engines. As a result, we would have five containers running inside the cluster. For all that to work, we were required to set up Swarm agents (as separate containers) on all the nodes that form the cluster and hook them into one of the supported data registries (Consul, etcd, and Zookeeper).

The problem was that Swarm was executing commands we send it. It was not maintaining the desired state. We were, effectively, telling it what we want to happen (e.g. scale up), not the state we wanted (make sure that five instances are running). Later on, the old Swarm got the feature that would reschedule containers from failed nodes. However, that feature had a few problems that prevented it from being a reliable solution (e.g. failed containers were not removed from the overlay network).

Now we got a brand new Swarm. It is part of Docker Engine (no need to run it as separate containers), it has incorporated service discovery (no need to set up Consul or whatever is your data registry of choice), it is designed from the ground up to accept and maintain the desired state, and so on. It is a truly major change in how we deal with cluster orchestration.

In the past, I was inclined towards the old Swarm more than Kubernetes. However, that inclination was only slight. There were pros and cons for using either solution. Kubernetes had a few features Swarm was missing (e.g. the concept of the desired state), the old Swarm shined with its simplicity and low usage of resources. With the new Swarm (the one that comes with v1.12), I have no more doubts which one to use. **The new Swarm is often a better choice than Kubernetes**. It is part of Docker Engine, so the whole setup is a single command that tells an engine to join the cluster. The new networking works like a charm. The bundle that can be used to define services can be created from Docker Compose files, so there is no need to maintain two sets of configurations (Docker Compose for development and a different one for orchestration). Most importantly, the new Docker Swarm continues being simple to use. From the very beginning, Docker community pledged that they are committed to simplicity and, with this release, they, once again, proved that to be true.

And that's not all. The new release comes with a lot of other features that are not directly related with Swarm. However, this book is dedicated to cluster management. Therefore, I'll focus on Swarm and leave the rest for one of the next books or a blog article.

Since I believe that code explains things better than words, we'll start with a demo of some of the new features introduced in version 1.12.

Setting Up a Swarm Cluster

We'll continue using Docker Machine since it provides a very convenient way to simulate a cluster on a laptop. Three servers should be enough to demonstrate some of the key features of a Swarm cluster.

All the commands from this chapter are available in the 02-docker-swarm.sh[35] Gist.

```
for i in 1 2 3; do
    docker-machine create -d virtualbox node-$i
done
```

At this moment, we have three nodes. Please note that those servers are not running anything but Docker Engine.

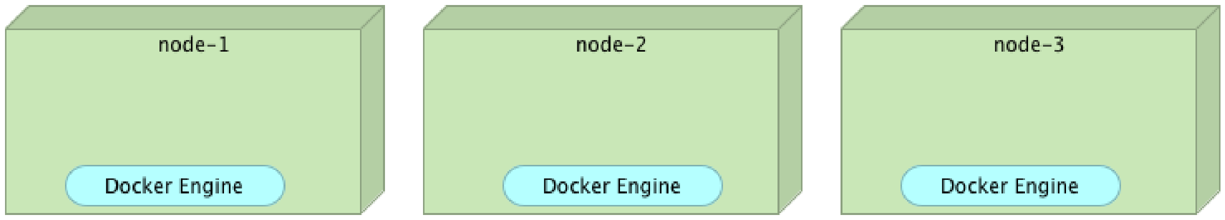
We can see the status of the nodes by executing the `ls` command.

[35]https://gist.github.com/vfarcic/750fc4117bad9d8619004081af171896

```
docker-machine ls
```

The output is as follows.

```
NAME    ACTIVE DRIVER     STATE   URL                        SWARM DOCKER   ERRORS
node-1 -      virtualbox Running tcp://192.168.99.100:2376       v1.12.1
node-2 -      virtualbox Running tcp://192.168.99.101:2376       v1.12.1
node-3 -      virtualbox Running tcp://192.168.99.102:2376       v1.12.1
```



Machines running Docker Engines

With the machines up and running we can proceed and set up the Swarm cluster.

The cluster setup consists of two types of commands. We need to start by initializing the first node which will be our manager.

```
eval $(docker-machine env node-1)

docker swarm init \
    --advertise-addr $(docker-machine ip node-1)
```

The first command set environment variables so that the local Docker Engine is pointing to the *node-1*. The second initialized Swarm on that machine.

We specified only one argument with the `swarm init` command. The `--advertise-addr` is the address that this node will expose to other nodes for internal communication.

The output of the `swarm init` command is as follows.

```
Swarm initialized: current node (1o5k7hvcply6g2excjiqqf4ed) is now a manager.

To add a worker to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-3czblm3rypyvrz6wyijsuwtmk1ozd7giqip0m6k0b3hllycgmv-3851i2ga\
ys638e7unmp2ng3az \
    192.168.99.100:2377

To add a manager to this swarm, run the following command:
    docker swarm join \
    --token SWMTKN-1-3czblm3rypyvrz6wyijsuwtmk1ozd7giqip0m6k0b3hllycgmv-6oukeshm\
w7a295vudzmo9mv6i \
    192.168.99.100:2377
```

We can see that the node is now a manager and we've got the commands we can use to join other nodes to the cluster. As a way to increase security, a new node can be added to the cluster only if it contains the token generated when Swarm was initialized. The token was printed as a result of the `docker swarm init` command. You can copy and paste the code from the output or use the `join-token` command. We'll use the latter.

Right now, our Swarm cluster consists of only one VM. We'll add the other two nodes to the cluster. But, before we do that, let us discuss the difference between a *manager* and a *worker*.

A swarm manager continuously monitors the cluster state and reconciles any differences between the actual state and your expressed desired state. For example, if you set up a service to run ten replicas of a container, and a worker machine hosting two of those replicas crashes, the manager will create two new replicas to replace the ones that failed. Swarm manager assigns new replicas to workers that are running and available. A manager has all the capabilities of a worker.

We can get a token required for adding additional nodes to the cluster by executing the `swarm join-token` command.

The command to obtain a token for adding a manager is as follows.

```
docker swarm join-token -q manager
```

Similarly, to get a token for adding a worker, we would run the command that follows.

```
docker swarm join-token -q worker
```

In both cases, we'd get a long hashed string.

The output of the worker token is as follows.

```
SWMTKN-1-3czblm3rypyvrz6wyijsuwtmk1ozd7giqip0m6k0b3hllycgmv-3851i2gays638e7unmp2\
ng3az
```

Please note that this token was generated on my machine and, in your case, it will be different.

Let’s put the token into an environment variable and add the other two nodes as workers.

```
TOKEN=$(docker swarm join-token -q worker)
```

Now that have the token inside a variable, we can issue the command that follows.

```
for i in 2 3; do
  eval $(docker-machine env node-$i)

  docker swarm join \
    --token $TOKEN \
    --advertise-addr $(docker-machine ip node-$i) \
    $(docker-machine ip node-1):2377
done
```

The command we just ran iterates over nodes two and three and executes the `swarm join` command. We set the token, the advertise address, and the address of our manager. As a result, the two machines joined the cluster as workers. We can confirm that by sending the `node ls` command to the *manager* node (*node-1*).

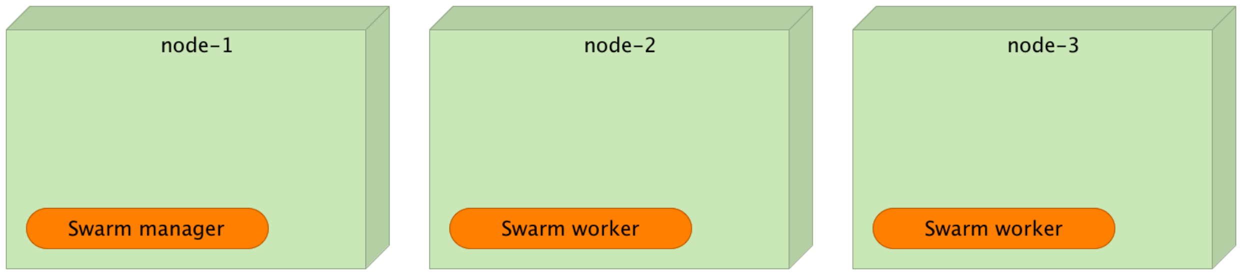
```
eval $(docker-machine env node-1)

docker node ls
```

The output of the `node ls` command is as follows.

```
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
3vlq7dsa8g2sqkp6vl911nha8    node-3    Ready   Active
6cbtgzk19rne5mzwkwugiolox    node-2    Ready   Active
b644vkvs6007rpjre2bfb8cro *  node-1    Ready   Active        Leader
```

The star tells us which node we are currently using. The *manager status* indicates that the *node-1* is the *leader*.



Docker Swarm cluster with three nodes

In a production environment, we would probably set more than one node to be a manager and, thus, avoid deployment downtime if one of them fails. For the purpose of this demo, having one manager should suffice.

Deploying Services To The Swarm Cluster

Before we deploy a demo service, we should create a new network so that all containers that constitute the service can communicate with each other no matter on which nodes they are deployed.

```
docker network create --driver overlay go-demo
```

> The next chapter will explore networking in more details. Right now, we'll discuss and do only the absolute minimum required for an efficient deployment of services inside a Swarm cluster.

We can check the status of all networks with the command that follows.

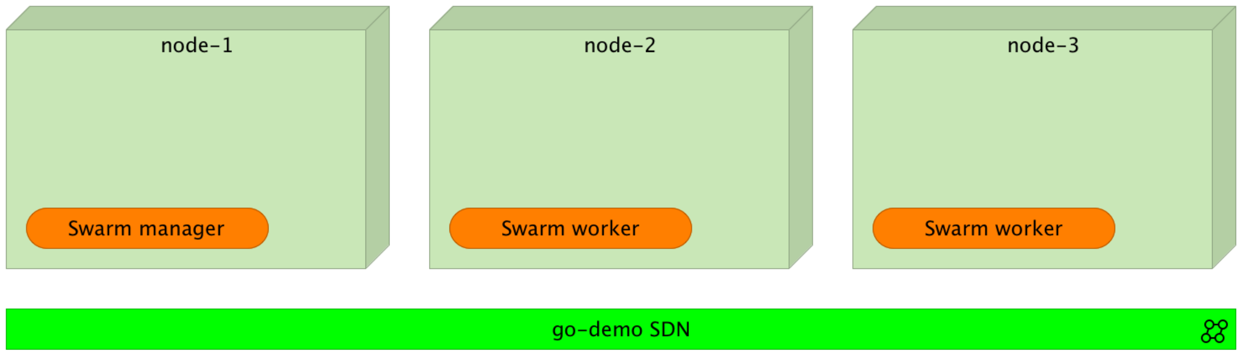
```
docker network ls
```

The output of the `network ls` command is as follows.

```
NETWORK ID   NAME            DRIVER  SCOPE
e263fb34287a bridge          bridge  local
c5b60cff0f83 docker_gwbridge bridge  local
8d3gs95h5c5q go-demo         overlay swarm
4d0719f20d24 host            host    local
eafx9zd0czuu ingress         overlay swarm
81d392ce8717 none            null    local
```

As you can see, we have two networks that have the *swarm* scope. The one named *ingress* was created by default when we set up the cluster. The second (*go-demo*) was created with the `network create` command. We'll assign all containers that constitute the *go-demo* service to that network.

The next chapter will go deep into the Swarm networking. For now, it is important to understand that all services that belong to the same network can speak with each other freely.



Docker Swarm cluster with Docker network (SDN)

The *go-demo* application requires two containers. Data will be stored in MongoDB. The back-end that uses that DB is defined as *vfarcic/go-demo* container.

Let's start by deploying the *mongo* container somewhere within the cluster.

Usually, we'd use constraints to specify the requirements for the container (e.g. HD type, the amount of memory and CPU, and so on). We'll skip that, for now, and tell Swarm to deploy it anywhere within the cluster.

```
docker service create --name go-demo-db \
  --network go-demo \
  mongo:3.2.10
```

Please note that we haven't specified the port Mongo listens to (*27017*). That means that the database will not be accesible to anyone but other services that belong to the same network (*go-demo*).

As you can see, the way we use `service create` is similar to the Docker `run` command you are, probably, already used to.

We can list all the running services.

```
docker service ls
```

Depending on how much time passed between `service create` and `service ls` commands, you'll see the value of the *Replicas* column being zero or one. Immediately after creating the service, the value should be *0/1*, meaning that zero replicas are running, and the objective is to have one. Once the *mongo* image is pulled, and the container is running, the value should change to *1/1*.

The final output of the `service ls` command should be as follows (IDs are removed for brevity).

```
NAME        MODE        REPLICAS  IMAGE
go-demo-db  replicated  1/1       mongo:3.2.10
```

If we need more information about the *go-demo-db* service, we can run the `service inspect` command.

```
docker service inspect go-demo-db
```

Now that the database is running, we can deploy the *go-demo* container.

```
docker service create --name go-demo \
  -e DB=go-demo-db \
  --network go-demo \
  vfarcic/go-demo:1.0
```

There's nothing new about that command. The service will be attached to the *go-demo* network. The environment variable *DB* is an internal requirement of the *go-demo* service that tells the code the address of the database.

At this point, we have two containers (*mongo* and *go-demo*) running inside the cluster and communicating with each other through the *go-demo* network. Please note that none of them is (yet) accessible from outside the network. At this point, your users do not have access to the service API. We'll discuss this in more details soon. Until then, I'll give you only a hint: *you need a reverse proxy* capable of utilizing the new Swarm networking.

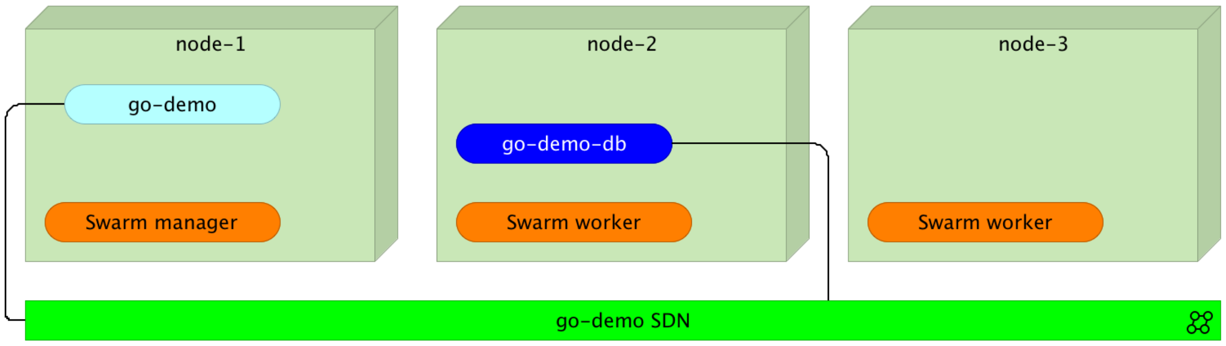
Let's run the `service ls` command one more time.

```
docker service ls
```

The result, after the *go-demo* service is pulled to the destination node, should be as follows (IDs are removed for brevity).

```
NAME       MODE       REPLICAS IMAGE
go-demo    replicated 1/1      vfarcic/go-demo:1.0
go-demo-db replicated 1/1      mongo:3.2.10
```

As you can see, both services are running as a single replica.



Docker Swarm cluster containers communicating through the go-demo SDN

What happens if we want to scale one of the containers? How do we scale our services?

Scaling Services

We should always run at least two instances of any given service. That way they can share the load and, if one of them fails, there will be no downtime. We'll explore Swarm's failover capability soon and leave load balancing for the next chapter.

We can, for example, tell Swarm that we want to run five replicas of the *go-demo* service.

```
docker service scale go-demo=5
```

With the `service scale` command, we scheduled five replicas. Swarm will make sure that five instances of *go-demo* are running somewhere inside the cluster.

We can confirm that, indeed, five replicas are running through the, already familiar, `service ls` command.

```
docker service ls
```

The output is as follows (IDs are removed for brevity).

```
NAME       MODE       REPLICAS IMAGE
go-demo    replicated 5/5      vfarcic/go-demo:1.0
go-demo-db replicated 1/1      mongo:3.2.10
```

As we can see, five out of five replicas of the *go-demo* service are running.

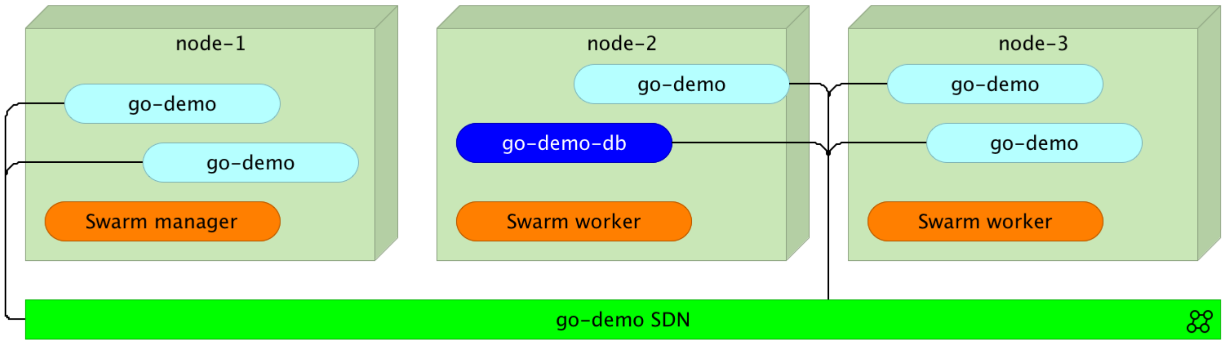
The `service ps` command provides more detailed information about a single service.

```
docker service ps go-demo
```

The output is as follows (IDs are removed for brevity).

```
NAME      IMAGE                NODE   DESIRED STATE CURRENT STATE           ER\
ROR PORTS
go-demo.1 vfarcic/go-demo:1.0 node-3 Running       Running about a minute ago
go-demo.2 vfarcic/go-demo:1.0 node-2 Running       Running 51 seconds ago
go-demo.3 vfarcic/go-demo:1.0 node-2 Running       Running 51 seconds ago
go-demo.4 vfarcic/go-demo:1.0 node-1 Running       Running 53 seconds ago
go-demo.5 vfarcic/go-demo:1.0 node-3 Running       Running about a minute ago
```

We can see that the *go-demo* service is running five instances distributed across the three nodes. Since they all belong to the same *go-demo* network, they can communicate with each other no matter where they run inside the cluster. At the same time, none of them is accessible from "outside".



Docker Swarm cluster with go-demo service scaled to five replicas

What happens if one of the containers is stopped or if the entire node fails? After all, processes and nodes do fail sooner or later. Nothing is perfect, and we need to be prepared for such situations.

Failover

Fortunately, failover strategies are part of Docker Swarm. Remember, when we execute a `service` command, we are not telling Swarm what to do but the state we desire. In turn, Swarm will do its best to maintain the specified state no matter what happens.

To test a failure scenario, we'll destroy one of the nodes.

```
docker-machine rm -f node-3
```

Swarm needs a bit of time until it detects that the node is down. Once it does, it will reschedule containers. We can monitor the situation through the `service ps` command.

```
docker service ps go-demo
```

The output (after rescheduling) is as follows (ID are removed for brevity).

```
NAME           IMAGE               NODE   DESIRED STATE CURRENT STATE            \
  ERROR PORTS
go-demo.1      vfarcic/go-demo:1.0 node-2 Running       Running 13 seconds ago
 \_ go-demo.1 vfarcic/go-demo:1.0 node-3 Shutdown      Running about a minute ago
go-demo.2      vfarcic/go-demo:1.0 node-2 Running       Running about a minute ago
go-demo.3      vfarcic/go-demo:1.0 node-2 Running       Running about a minute ago
go-demo.4      vfarcic/go-demo:1.0 node-1 Running       Running about a minute ago
go-demo.5      vfarcic/go-demo:1.0 node-1 Running       Running 13 seconds ago
 \_ go-demo.5 vfarcic/go-demo:1.0 node-3 Shutdown      Running about a minute ago
```

As you can see, after a short period, Swarm rescheduled containers among healthy nodes (*node-1* and *node-2*) and changed the state of those that were running on the failed node to *Shutdown*. If your output still shows that some instances are running on the *node-3*, please wait for a few moments and repeat the `service ps` command.

What Now?

That concludes the exploration of basic concepts of the new Swarm features we got with Docker v1.12+.

Is this everything there is to know to run a Swarm cluster successfully? Not even close! What we explored by now is only the beginning. There are quite a few questions waiting to be answered. How do we expose our services to the public? How do we deploy new releases without downtime? I'll try to give answers to those and quite a few other questions in the chapters that follow. The next one will be dedicated to the exploration of the ways we can expose our services to the public. We'll try to integrate a proxy with a Swarm cluster. To do that, we need to dive deeper into Swarm networking.

Now is the time to take a break before diving into the next chapter. As before, we'll destroy the machines we created and start fresh.

```
docker-machine rm -f node-1 node-2
```

Docker Swarm Networking And Reverse Proxy

The most compelling reason for most people to buy a computer for the home will be to link it to a nationwide communications network. We're just in the beginning stages of what will be a truly remarkable breakthrough for most people - as remarkable as the telephone.

– Steve Jobs

Software-defined network (SDN) is a cornerstone of efficient cluster management. Without it, services distributed across the cluster would not be able to find each other.

Having proxies based on static configuration does not fit the world of highly dynamic scheduling. Services are created, updated, moved around the cluster, scaled and de-scaled, and so on. In such a setting, information changes all the time.

One approach we can take is to use a proxy as a central communication point and make all the services speak with each other through it. Such a setting would require us to monitor changes in the cluster continuously and update the proxy accordingly. To make our lives easier, a monitoring process would probably use one of the service registries to store the information and a templating solution that would update proxy configuration whenever a change in the registry is detected. As you can imagine, building such a system is anything but trivial.

Fortunately, Swarm comes with a brand new networking capability. In a nutshell, we can create networks and attach them to services. All services that belong to the same network can speak with each other using only the name of the service. It goes even further. If we scale a service, Swarm networking will perform round-robin load balancing and distribute the requests across all the instances. When even that is not enough, we have a new network called *ingress* with *routing mesh* that has all those and a few additional features.

Efficient usage of Swarm networking is not sufficient by itself. We still need a reverse proxy that will be a bridge between the external world and our services. Unless there are special requirements, the proxy does not need to perform load balancing (Swarm networking does that for us). However, it does need to evaluate request paths and forward requests to a destination service. Even in that case, Swarm networking helps a lot. Configuring reverse proxy becames a relatively easy thing to do as long as we understand how networking works and can harness its full potential.

Let's see the networking in practice.

Setting Up a Cluster

We'll create a similar environment as we did in the previous chapter. We'll have three nodes which will form a Swarm cluster.

All the commands from this chapter are available in the 03-networking.sh [36] Gist.

By this time, you already know how to set up a cluster so we'll skip the explanation and just do it.

```
for i in 1 2 3; do
  docker-machine create -d virtualbox node-$i
done

eval $(docker-machine env node-1)

docker swarm init \
  --advertise-addr $(docker-machine ip node-1)

TOKEN=$(docker swarm join-token -q worker)

for i in 2 3; do
  eval $(docker-machine env node-$i)

  docker swarm join \
    --token $TOKEN \
    --advertise-addr $(docker-machine ip node-$i) \
    $(docker-machine ip node-1):2377
done

eval $(docker-machine env node-1)

docker node ls
```

The output of the last command (`node ls`) is as follows (IDs were removed for brevity).

[36] https://gist.github.com/vfarcic/fd7d7e04e1133fc3c90084c4c1a919fe

```
HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
node-2    Ready   Active
node-1    Ready   Active        Leader
node-3    Ready   Active
```

As you can see, we have a cluster of three nodes with `node-1` being the only manager (and hence the leader).

Now that we have a fully operating cluster, we can explore the benefits Docker networking provides in conjunction with Swarm. We already worked with Swarm networking in the previous chapter. Now its time to go deeper, gain a better understanding of what we already saw, and unlock some new features and use cases.

Requirements Of Secured And Fault Tolerant Services Running With High Availability

Let us quickly go over the internals of the *go-demo* application. It consists of two services. Data is stored in a Mongo DB. The database is consumed by a backend service called *go-demo*. No other service should access the database directly. If another service needs the data, it should send a request to the *go-demo* service. That way we have clear boundaries. Data is owned and managed by the *go-demo* service. It exposes an API that is the only access point to the data.

The system should be able to host multiple applications. Each will have a unique base URL. For example, the *go-demo* path starts with */demo*. The other applications will have different paths (e.g. */users*, */products*, and so on). The system will be accessible only through ports *80* (*HTTP*) and *443* (*HTTPS*). Please note that there can be no two processes that can listen to the same port. In other words, only a single service can be configured to listen to port *80*.

To meet load fluctuations and use the resources effectively, we must be able to scale (or de-scale) each service individually and independently from the others. Any request to any of the services should pass through a load balancer that will distribute the load across all instances. As a minimum, at least two instances of any service should be running at any given moment. That way, we can accomplish high availability even in case one of the instances stops working. We should aim even higher than that and make sure that even a failure of a whole node does not interrupt the system as a whole.

To meet performance and failover needs services should be distributed across the cluster.

> We'll make a temporary exception to the rule that each service should run multiple instances. Mongo volumes do not work with Docker Machine on OS X and Windows. Later on, when we reach the chapters that provide guidance towards production setup inside major hosting providers (e.g. AWS), we'll remove this exception and make sure that the database is also configured to run with multiple instances.

Taking all this into account, we can make the following requirements.

1. A load balancer will distribute requests evenly (*round-robin*) across all instances of any given service (proxy included). It should be fault tolerant and not depend on any single node.
2. A reverse proxy will be in charge of routing requests based on their base URLs.
3. The *go-demo* service will be able to communicate freely with the *go-demo-db* service and will be accessible only through the reverse proxy.
4. The database will be isolated from any but the service it belongs to (*go-demo*).

A logical architecture of what we're trying to accomplish can be presented with the diagram that follows.

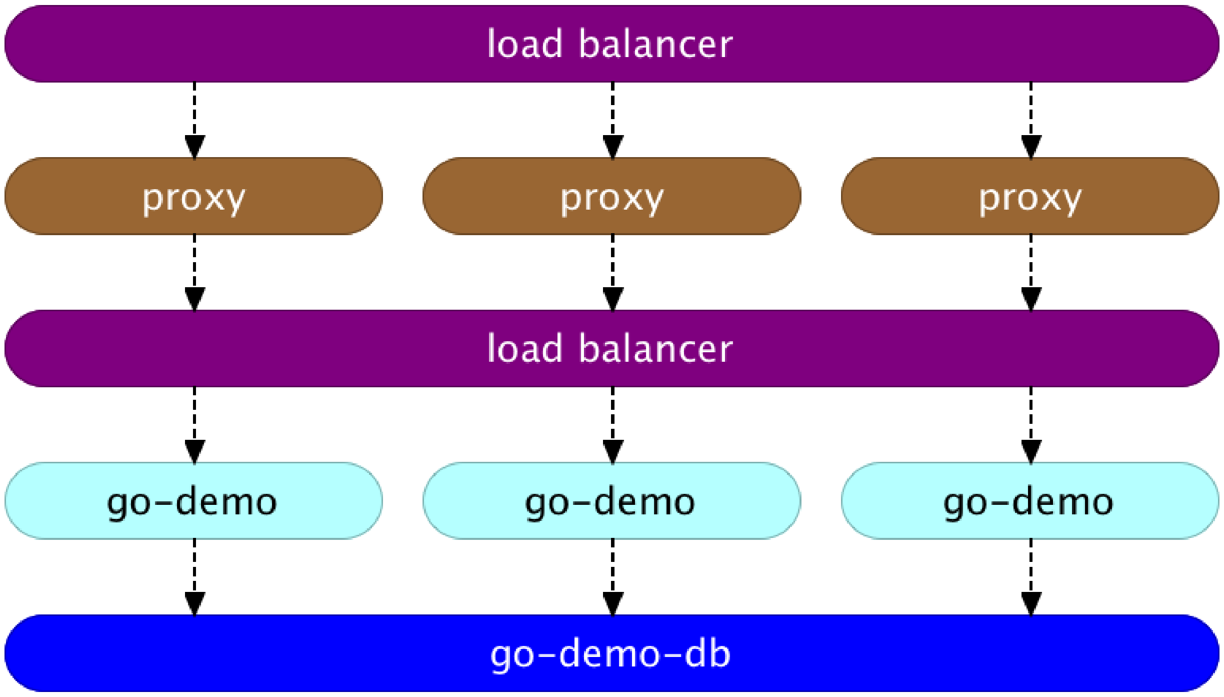


Figure 3-1: A logical architecture of the go-demo service

How can we accomplish those requirements?

Let us solve each of the four requirements one by one. We'll start from the bottom and move towards the top.

The first problem to tackle is how to run a database isolated from any but the service it belongs to.

Running a Database In Isolation

We can isolate a database service by not exposing its ports. That can be accomplished easily with the `service create` command.

```
docker service create --name go-demo-db \
  mongo:3.2.10
```

We can confirm that the ports are indeed not exposed by inspecting the service.

```
docker service inspect --pretty go-demo-db
```

The output is as follows.

```
ID:             rcedo70r2f1njpm0eyb3nwf8w
Name:           go-demo-db
Service Mode:   Replicated
 Replicas:      1
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Max failure ratio: 0
ContainerSpec:
 Image:         mongo:3.2.10@sha256:532a19da83ee0e4e2a2ec6bc4212fc4af26357c040675\
d5c2629a4e4c4563cef
Resources:
Endpoint Mode: vip
```

As you can see, there is no mention of any port. Our *go-demo-db* service is fully isolated and inaccessible to anyone. However, that is too much isolation. We want the service to be isolated from anything but the service it belongs to (*go-demo*). We can accomplish that through the usage of Docker Swarm networking.

Let us remove the service we created and start over.

```
docker service rm go-demo-db
```

This time, we should create a network and make sure that the *go-demo-db* service is attached to it.

```
docker network create --driver overlay go-demo

docker service create --name go-demo-db \
  --network go-demo \
  mongo:3.2.10
```

We created an overlay network called `go-demo` followed with the `go-demo-db` service. This time, we used the `--network` argument to attach the service to the network. From this moment on, all services attached to the `go-demo` network will be accessible to each other.

Let's inspect the service and confirm that it is indeed attached to the network.

```
docker service inspect --pretty go-demo-db
```

The output of the `service inspect` command is as follows.

```
ID:             ktrxcgp3gtszsjvi7xg0hmd73
Name:                    go-demo-db
Service Mode:   Replicated
 Replicas:      1
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Max failure ratio: 0
ContainerSpec:
 Image:         mongo:3.2.10@sha256:532a19da83ee0e4e2a2ec6bc4212fc4af26357c040675\
d5c2629a4e4c4563cef
Resources:
Networks:       go-demo
Endpoint Mode: vip
```

As you can see, this time, there is a `Networks` entry with the value set to the ID of the `go-demo` network we created earlier.

Let us confirm that networking truly works. To prove it, we'll create a global service called `util`.

```
docker service create --name util \
    --network go-demo --mode global \
    alpine sleep 1000000000
```

Just as `go-demo-db`, the `util` service also has the `go-demo` network attached.

A new argument is `--mode`. When set to global, the service will run on every node of the cluster. That is a very useful feature when we want to set up infrastructure services that should span the whole cluster.

We can confirm that it is running everywhere by executing the `service ps` command.

```
docker service ps util
```

The output is as follows (IDs are removed for brevity).

```
NAME    IMAGE         NODE   DESIRED STATE CURRENT STATE         ERROR PORTS
util... alpine:latest node-1 Running       Running 6 minutes ago
util... alpine:latest node-3 Running       Running 6 minutes ago
util... alpine:latest node-2 Running       Running 6 minutes ago
```

As you can see, the `util` service is running on all three nodes.

We are running the `alpine` image (a minuscule Linux distribution). We put it to sleep for a very long time. Otherwise, since no processes are running, the service would stop, Swarm would restart it, it would stop again, and so on.

The purpose of the `util` service will be to demonstrate some of the concepts we're exploring. We'll `exec` into it and confirm that the networking truly works.

To enter the `util` container, we need to find out the ID of the instance running on the *node-1* (the node our local Docker is pointing to).

A note to Windows users

Windows does not yet support running containers in the interactive mode. You will have to enter the machine for the commands in the rest of this sub-chapter to work. To enter the Docker Machine, please execute the command that follows.

```
docker-machine ssh node-1
```

When you reach the end of this sub-chapter, please exit the machine.

```
exit
```

```
ID=$(docker ps -q --filter label=com.docker.swarm.service.name=util)
```

We listed all the processes (`ps`) in quiet mode so that only IDs are returned (`-q`), and limited the result to the service name `util` (`--filter label=com.docker.swarm.service.name=util`). The result is stored as the environment variable `ID`.

We'll install a tool called *drill*. It is a tool designed to get all sorts of information out of a DNS and it will come in handy very soon.

```
docker exec -it $ID apk add --update drill
```

Alpine Linux uses the package management called `apk`, so we told it to add `drill`.

Now we can see whether networking truly works. Since both `go-demo-db` and `util` services belong to the same network, they should be able to communicate with each other using DNS names. Whenever we attach a service to the network, a new virtual IP is created together with a DNS that matches the name of the services.

Let's try it out.

```
docker exec -it $ID drill go-demo-db
```

We entered into one of the instances of the `util` service and “drilled” the DNS `go-demo-db`. The output is as follows.

```
;; ->>HEADER<<- opcode: QUERY, rcode: NOERROR, id: 5751
;; flags: qr rd ra ; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; QUESTION SECTION:
;; go-demo-db.        IN            A

;; ANSWER SECTION:
go-demo-db.           600           IN            A             10.0.0.2

;; AUTHORITY SECTION:

;; ADDITIONAL SECTION:

;; Query time: 0 msec
;; SERVER: 127.0.0.11
;; WHEN: Thu Sep  1 12:53:42 2016
;; MSG SIZE  rcvd: 54
```

The response code is `NOERROR` and the `ANSWER` is `1` meaning that the DNS `go-demo-db` responded correctly. It is reachable.

We can also observe that the `go-demo-db` DNS is associated with the IP `10.0.0.2`. Every service attached to a network gets its IP. Please note that I said service, not an instance. That’s a huge difference that we’ll explore later. For now, it is important to understand that all services that belong to the same network are accessible through service names.

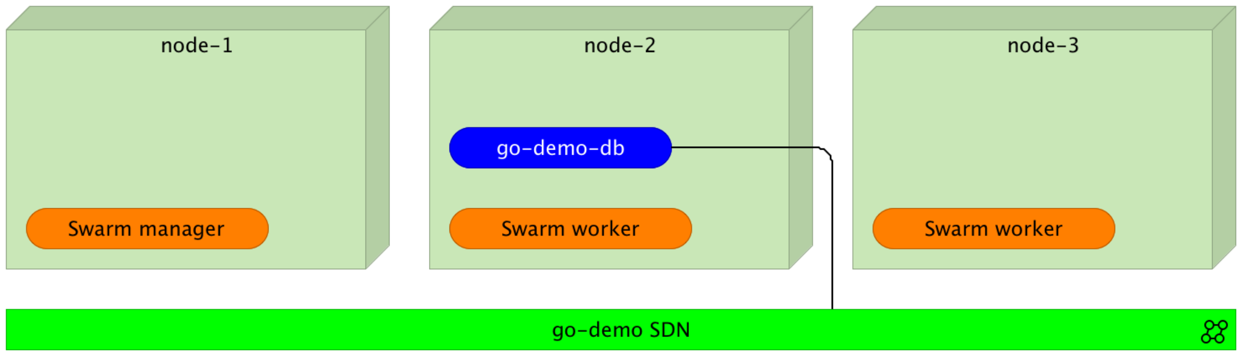


Figure 3-2: go-demo-db service attached to the go-demo network

Let’s move up through the requirements.

Running a Service Through a Reverse Proxy

We want the *go-demo* service to be able to communicate freely with the *go-demo-db* service and to be accessible only through the reverse proxy. We already know how to accomplish the first part. All we have to do is make sure that both services belong to the same network (`go-demo`).

How can we accomplish the integration with a reverse proxy?

We can start by creating a new network and attach it to all services that should be accessible through a reverse proxy.

```
docker network create --driver overlay proxy
```

Let's list the currently running overlay networks.

```
docker network ls -f "driver=overlay"
```

The output is as follows.

```
NETWORK ID   NAME    DRIVER  SCOPE
b17kzasd3gzu go-demo overlay swarm
0d7ssryojcyg ingress overlay swarm
9e4o7abyts0v proxy   overlay swarm
```

We have the `go-demo` and `proxy` networks we created earlier. The third one is called `ingress`. It is set up by default and has a special purpose that we'll explore later.

Now we are ready to run the `go-demo` service. We want it to be able to communicate with the `go-demo-db` service so it must be attached to the `go-demo` network. We also want it to be accessible to a proxy (we'll create it soon) so we'll attach it to the `proxy` network as well.

The command that creates the `go-demo` service is as follows.

```
docker service create --name go-demo \
  -e DB=go-demo-db \
  --network go-demo \
  --network proxy \
  vfarcic/go-demo:1.0
```

It is very similar to the command we executed in the previous chapter with the addition of the `--network proxy` argument.

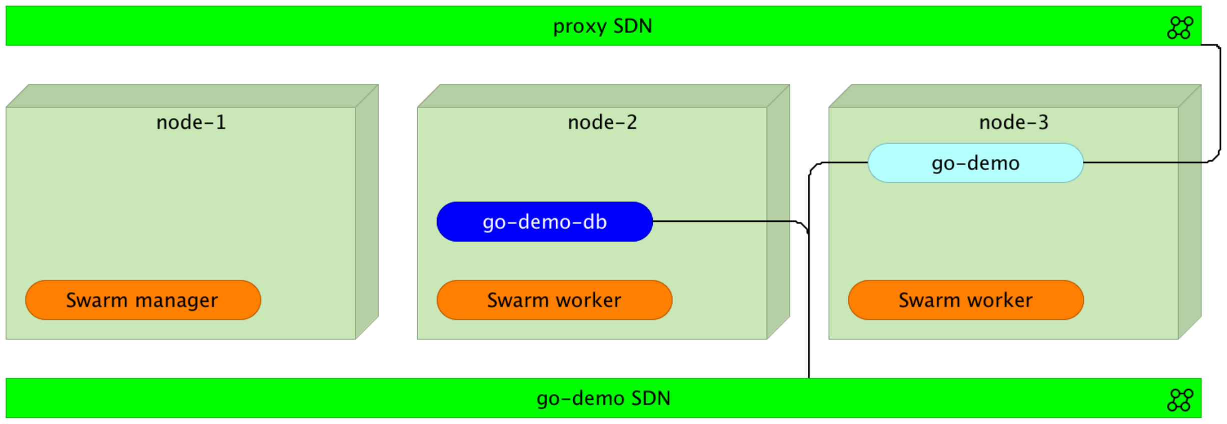


Figure 3-3: Docker Swarm cluster with three nodes, two networks and a few containers

Now both services are running somewhere inside the cluster and can communicate with each other through the *go-demo* network. Let's bring the proxy into the mix. We'll use the Docker Flow Proxy[37] project that is a combination of HAProxy[38] and a few additional features that make it more dynamic. The principles we'll explore are the same no matter which one will be your choice.

Please note that, at this moment, none of the services are accessible to anyone except those attached to the same network.

Creating a Reverse Proxy Service In Charge Of Routing Requests Depending On Their Base URLs

We can implement a reverse proxy in a couple of ways. One would be to create a new image based on *HAProxy*[39] and include configuration files inside it. That approach would be a good one if the number of different services is relatively static. Otherwise, we'd need to create a new image with a new configuration every time there is a new service (not a new release).

The second approach would be to expose a volume. That way, when needed, we could modify the configuration file instead building a whole new image. However, that has downsides as well. When deploying to a cluster, we should avoid using volumes whenever they're not necessary. As you'll see soon, a proxy is one of those that do not require a volume. As a side note, `--volume` has been replaced with the `docker service` argument `--mount`.

The third option is to use one of the proxies designed to work with Docker Swarm. In this case, we'll use the container *vfarcic/docker-flow-proxy*[40]. It is based on HAProxy with additional features that allow us to reconfigure it by sending HTTP requests.

Let's give it a spin.

[37]https://github.com/vfarcic/docker-flow-proxy
[38]http://www.haproxy.org/
[39]https://hub.docker.com/_/haproxy/
[40]https://hub.docker.com/r/vfarcic/docker-flow-proxy/

The command that creates the proxy service is as follows.

```
docker service create --name proxy \
    -p 80:80 \
    -p 443:443 \
    -p 8080:8080 \
    --network proxy \
    -e MODE=swarm \
    vfarcic/docker-flow-proxy
```

We opened ports *80* and *443* that will serve Internet traffic (*HTTP* and *HTTPS*). The third port is *8080*. We'll use it to send configuration requests to the proxy. Further on, we specified that it should belong to the *proxy* network. That way, since *go-demo* is also attached to the same network, the proxy can access it through the SDN.

Through the proxy we just ran, we can observe one of the cool features of the network routing mesh. It does not matter which server the proxy is running in. We can send a request to any of the nodes and Docker networking will make sure that it is redirected to one of the proxies. We'll see that in action very soon.

The last argument is the environment variable *MODE* that tells the proxy that containers will be deployed to a Swarm cluster. Please consult the project README[41] for other combinations.

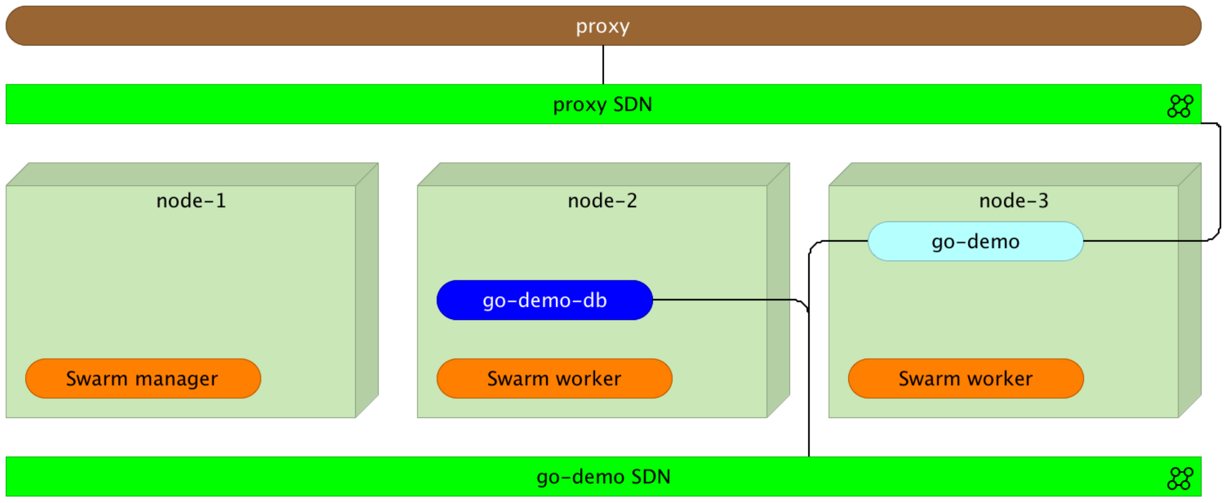


Figure 3-4: Docker Swarm cluster with the proxy service

> Please note that the proxy, even though it is running inside one of the nodes, is placed outside to illustrate the logical separation better.

Before we move on, let's confirm that the proxy is running.

[41]https://github.com/vfarcic/docker-flow-proxy

```
docker service ps proxy
```

We can proceed if the *CURRENT STATE* is *Running*. Otherwise, please wait until the service is up and running.

Now that the proxy is deployed, we should let it know about the existence of the *go-demo* service.

```
curl "$(docker-machine ip node-1):8080/v1/docker-flow-proxy/reconfigure?serviceN\
ame=go-demo&servicePath=/demo&port=8080"
```

The request was sent to *reconfigure* the proxy specifying the service name (*go-demo*), base URL path of the API (*/demo*), and the internal port of the service (*8080*). From now on, all the requests to the proxy with the path that starts with */demo* will be redirected to the *go-demo* service. This request is one of the additional features *Docker Flow Proxy* provides on top of *HAProxy*.

Please note that we sent the request to `node-1`. The proxy could be running inside any of the nodes and, yet, the request was successful. That is where Docker's Routing Mesh plays a critical role. We'll explore it in more detail later. For now, the important thing to note is that we can send a request to any of the nodes, and it will be redirected to the service that listens to the same port (in this case *8080*).

The output of the request is as follows (formatted for readability).

```
{
  "Mode": "swarm",
  "Status": "OK",
  "Message": "",
  "ServiceName": "go-demo",
  "AclName": "",
  "ConsulTemplateFePath": "",
  "ConsulTemplateBePath": "",
  "Distribute": false,
  "HttpsOnly": false,
  "HttpsPort": 0,
  "OutboundHostname": "",
  "PathType": "",
  "ReqMode": "http",
  "ReqRepReplace": "",
  "ReqRepSearch": "",
  "ReqPathReplace": "",
  "ReqPathSearch": "",
  "ServiceCert": "",
  "ServiceDomain": null,
  "SkipCheck": false,
  "TemplateBePath": "",
  "TemplateFePath": "",
  "TimeoutServer": "",
  "TimeoutTunnel": "",
  "Users": null,
  "ServiceColor": "",
  "ServicePort": "",
  "AclCondition": "",
  "FullServiceName": "",
  "Host": "",
  "LookupRetry": 0,
  "LookupRetryInterval": 0,
  "ServiceDest": [
    {
      "Port": "8080",
      "ServicePath": [
        "/demo"
      ],
      "SrcPort": 0,
      "SrcPortAcl": "",
      "SrcPortAclName": ""
    }
  ]
}
```

I won't go into details but note that the status is `OK` indicating that the proxy was reconfigured correctly.

We can test that the proxy indeed works as expected by sending an HTTP request.

```
curl -i "$(docker-machine ip node-1)/demo/hello"
```

The output of the `curl` command is as follows.

```
HTTP/1.1 200 OK
Date: Thu, 01 Sep 2016 14:23:33 GMT
Content-Length: 14
Content-Type: text/plain; charset=utf-8

hello, world!
```

The proxy works! It responded with the HTTP status *200* and returned the API response *hello, world!*. As before, the request was not, necessarily, sent to the node that hosts the service but to the routing mesh that forwarded it to the proxy.

As an example, let's send the same request but this time, to *node-3*.

```
curl -i "$(docker-machine ip node-3)/demo/hello"
```

The result is still the same.

Let's explore the configuration generated by the proxy. It will give us more insights into the Docker Swarm Networking inner workings. As another benefit, if you choose to roll your own proxy solution, it might be useful to understand how to configure the proxy and leverage new Docker networking features.

We'll start by examining the configuration *Docker Flow Proxy*[42] created for us. We can do that by entering the running container to take a sneak peek at the file */cfg/haproxy.cfg*. The problem is that finding a container run by Docker Swarm is a bit tricky. If we deployed it with Docker Compose, the container name would be predictable. It would use the format *<PROJECT>_<SERVICE>_<INDEX>*. The `docker service` command runs containers with hashed names. The *docker-flow-proxy* created on my laptop has the name *proxy.1.e07jvhdb9e6s76mr9ol41u4sn*. Therefore, to get inside a running container deployed with Docker Swarm, we need to use a filter with, for example, an image name.

First, we need to find out on which node the proxy is running.

```
NODE=$(docker service ps proxy | tail -n +2 | awk '{print $4}')
```

We listed the `proxy` service processes (`docker service ps proxy`), removed the header (`tail -n +2`), and output the node that resides inside the fourth column (`awk '{print $4}'`). The output is stored as the environment variable `NODE`.

Now we can point our local Docker Engine to the node where the proxy resides.

```
eval $(docker-machine env $NODE)
```

A note to Windows users

Windows does not yet support running containers in the interactive mode. Instead running the `eval` command, please enter the machine and run the rest of the commands from inside it.

```
docker-machine ssh $NODE
```

When you reach the end of this sub-chapter, please exit the machine.

```
exit
```

[42]https://github.com/vfarcic/docker-flow-proxy

Finally, the only thing left is to find the ID of the proxy container. We can do that with the following command.

```
ID=$(docker ps -q \
    --filter label=com.docker.swarm.service.name=proxy)
```

Now that we have the container ID stored inside the variable, we can execute the command that will retrieve the HAProxy configuration.

```
docker exec -it \
    $ID cat /cfg/haproxy.cfg
```

The important part of the configuration is as follows.

```
frontend services
    bind *:80
    bind *:443
    mode http

    acl url_go-demo8080 path_beg /demo
    use_backend go-demo-be8080 if url_go-demo8080

backend go-demo-be8080
    mode http
    server go-demo go-demo:8080
```

The first part (`frontend`) should be familiar to those who have used HAProxy. It accepts requests on ports `80` (*HTTP*) and `443` (*HTTPS*). If the path starts with `/demo`, it will be redirected to the `backend go-demo-be`. Inside it, requests are sent to the address `go-demo` on the port `8080`. The address is the same as the name of the service we deployed. Since `go-demo` belongs to the same network as the proxy, Docker will make sure that the request is redirected to the destination container. Neat, isn't it? There is no need, anymore, to specify IPs and external ports.

The next question is how to do load balancing. How should we specify that the proxy should, for example, perform round-robin across all instances? Should we use a proxy for such a task?

Load Balancing Requests Across All Instances Of A Service

Before we explore load balancing, we need to have something to balance. We need multiple instances of a service. Since we already explored scaling in the previous chapter, the command should not come as a surprise.

```
eval $(docker-machine env node-1)

docker service scale go-demo=5
```

Within a few moments, five instances of the *go-demo* service will be running.

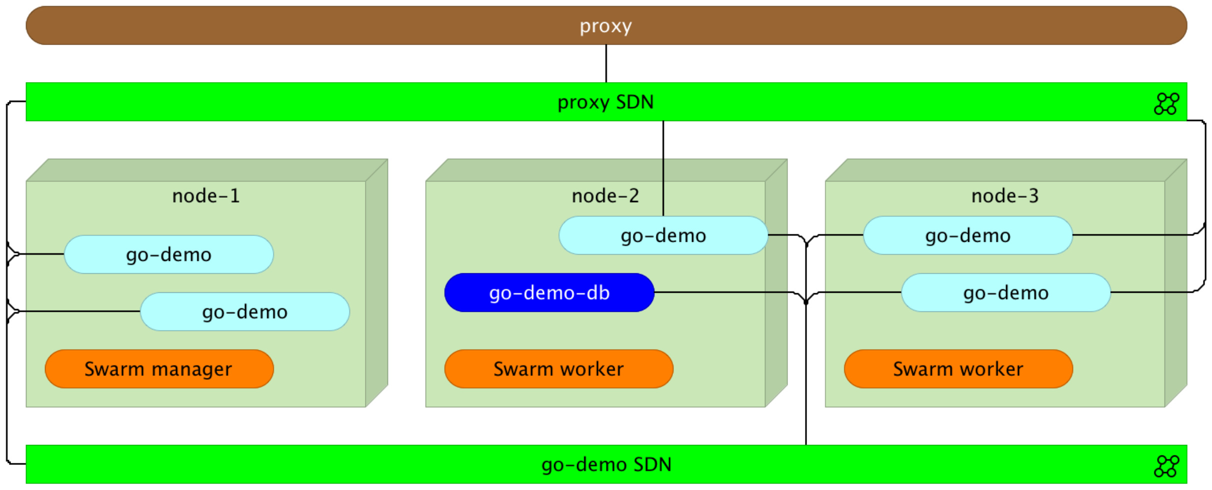


Figure 3-5: Docker Swarm cluster with the go-demo service scaled

What should we do to make the proxy load balance requests across all instances? The answer is nothing. No action is necessary on our part. Actually, the question is wrong. The proxy will not load balance requests at all. Docker Swarm networking will. So, let us reformulate the question. What should we do to make the *Docker Swarm network* load balance requests across all instances? Again, the answer is nothing. No action is necessary on our part.

To understand load balancing, we might want to go back in time and discuss load balancing before Docker networking came into being.

Normally, if we didn't leverage Docker Swarm features, we would have something similar to the following proxy configuration mock-up.

```
backend go-demo-be
  server instance_1 <INSTANCE_1_IP>:<INSTANCE_1_PORT>
  server instance_2 <INSTANCE_2_IP>:<INSTANCE_2_PORT>
  server instance_3 <INSTANCE_3_IP>:<INSTANCE_3_PORT>
  server instance_4 <INSTANCE_4_IP>:<INSTANCE_4_PORT>
  server instance_5 <INSTANCE_5_IP>:<INSTANCE_5_PORT>
```

Every time a new instance is added, we would need to add it to the configuration as well. If an instance is removed, we would need to remove it from the configuration. If an instance failed… Well, you get the point. We would need to monitor the state of the cluster and update the proxy configuration whenever a change occurs.

If you read *The DevOps 2.0 Toolkit*, you probably remember that I advised a combination of Registrator[43], Consul[44], and Consul Template[45]. Registrator would monitor Docker events and update Consul whenever a container is created or destroyed. With the information stored in Consul, we would use Consul Template to update *nginx* or *HAProxy* configuration. There is no need for such a combination anymore. While those tools still provide value, for this particular purpose, there is no need for them.

We are not going to update the proxy every time there is a change inside the cluster, for example, a scaling event. Instead, we are going to update the proxy every time a new service is created. Please note that service updates (deployment of new releases) do not count as service creation. We create a service once and update it with each new release (among other reasons). So, only a new service requires a change in the proxy configuration.

The reason behind that reasoning is in the fact that load balancing is now part of Docker Swarm networking. Let's do another round of *drilling* from the `util` service.

A note to Windows users

Windows does not yet support running containers in the interactive mode. Please enter the `node-1` machine and run the rest of the commands from inside it.

```
docker-machine ssh node-1
```

When you reach the end of this sub-chapter, please exit the machine.

```
exit
```

```
ID=$(docker ps -q --filter label=com.docker.swarm.service.name=util)

docker exec -it $ID apk add --update drill

docker exec -it $ID drill go-demo
```

The output of the last command is as follows.

[43] https://github.com/gliderlabs/registrator
[44] https://www.consul.io/
[45] https://github.com/hashicorp/consul-template

```
;; ->>HEADER<<- opcode: QUERY, rcode: NOERROR, id: 50359
;; flags: qr rd ra ; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; QUESTION SECTION:
;; go-demo.             IN          A

;; ANSWER SECTION:
go-demo.              600          IN          A          10.0.0.8

;; AUTHORITY SECTION:

;; ADDITIONAL SECTION:

;; Query time: 0 msec
;; SERVER: 127.0.0.11
;; WHEN: Thu Sep  1 17:46:09 2016
;; MSG SIZE  rcvd: 48
```

The IP `10.0.0.8` represents the `go-demo` service, not an individual instance. When we sent a drill request, Swarm networking performed load balancing (LB) across all of the instances of the service. To be more precise, it performed *round-robin* LB.

Besides creating a virtual IP for each service, each instance gets its own IP as well. In most cases, there is no need discovering those IPs (or any Docker network endpoint IP) since all we need is a service name, which gets translated to an IP and load balanced in the background.

What Now?

That concludes the exploration of basic concepts of the Docker Swarm networking.

Is this everything there is to know to run a Swarm cluster successfully? In this chapter, we went deeper into Swarm features, but we are not yet done. There are quite a few questions waiting to be answered. In the next chapter, we'll explore *service discovery* and the role it has in the Swarm Mode.

Now is the time to take a break before diving into the next chapter. As before, we'll destroy the machines we created and start fresh.

```
docker-machine rm -f node-1 node-2 node-3
```