LUKASZ GUMINSKI

Orchestrate Containers for Development with Docker Compose

Share this Codeship Guide

About the Author.

Lukasz Guminski is Senior Engineer at Container Solutions in Amsterdam. He mainly works on research & development in the field of Continuous Delivery and has vast experience in cloud computing and virtualization.

Container Solutions is a premium software consultancy that focuses on programmable infrastructure to help their customers innovate at speed and scale.

They provide support and consulting to technical and management teams who are carrying out complex technical projects and transitions as well as training for a range of technologies, including Docker and Mesos.

Learn more about Container Solutions here.

Orchestrate Containers for Development with Docker Compose.

The powerful concept of microservices is gradually changing the industry. Large monolithic services are slowly giving way to swarms of small and autonomous microservices that work together.

The process is accompanied by another market trend: **Containerization**. Together, they help us build systems of unprecedented resilience.

In this eBook, you will get a detailed breakdown of how to use Docker Compose, with a focus on how to orchestrate containers in development.

WHAT IS DOCKER COMPOSE?

Compose is a tool for defining and running multi-container applications with Docker. With Compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running.

Using Containers

Containerization changes not only the architecture of services, but also the structure of environments used to create them. Now, when software is distributed in containers, developers have full freedom to decide what applications they need.

As a result, even complex environments like Continuous Integration servers with database backends and analytical infrastructure can be instantiated **within seconds**. Software development becomes easier and more effective.

The changes naturally cause new problems. For instance, as a developer, how can I easily recreate a microservice architecture on my development machine? And how can I be sure that it remains unchanged as it propagates through a Continuous Delivery process? And finally, how can I be sure that a complex build & test environment can be reproduced easily?

The answer to these questions is **Docker Compose.**

Introducing Docker Compose

Docker Compose is "a tool for defining and running complex applications with Docker. With Compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running." (https://docs.docker.com/compose/).

All of that can be done by Docker Compose in the scope of a single host. In that sense, its concept is very similar to Kubernetes pods. For multi-host deployment, you should use more advanced solutions, like Apache Mesos or a complete Google Kubernetes architecture.

On the following pages you will learn how you can use Docker Compose in detail. Specifically, how to orchestrate containers in development.

Functionality of Docker Compose

The main function of Docker Compose is the creation of microservice architecture, meaning the containers and the links between them. But the tool is capable of much more. Let's have a look!

- **Building images** (if an appropriate Dockerfile is provided)

COMMAND LINE
```
docker-compose build
```

- **Scaling containers running a given service**

COMMAND LINE
```
docker-compose scale SERVICE=3
```

- **Healing, i.e., re-running containers that have stopped**

COMMAND LINE
```
docker-compose up --no-recreate
```

All of this functionality is available through the `docker-compose.yml` utility, which has a very similar set of commands to what is offered by `docker`.

The set of commands provided by the docker-compose utility

COMMANDS OF DOCKER-COMPOSE

```
build    Build or rebuild services
help     Get help on a command
kill     Kill containers
logs     View output from containers
port     Print the public port for a port binding
ps       List containers
pull     Pulls service images
rm       Remove stopped containers
run      Run a one-off command
scale    Set number of containers for a service
start    Start services
stop     Stop services
restart  Restart services
up       Create and start containers
```

The docker-compose commands are not only similar to docker commands, but they also behave like `docker` counterparts. The only difference is that they affect the entire multi-container architecture defined in the `docker-compose.yml` configuration file and not just a single container.

You'll notice some `docker` commands are not present in `docker-compose`. Those are the ones that don't make sense in the context of a completely multi-container setup. For instance:

- Commands for image manipulation like: `save, search, images, import, export, tag, history`
- User-interactive like: `attach, exec, 'run -i', login, wait`

A command that is worth your attention is the `docker-compose up` command. It is a shorthand form of `docker-compose build && docker-compose run`.

Docker Compose Workflow

There are three steps to using Docker Compose:

- Define each service in a Dockerfile.
- Define the services and their relation to each other in the `docker-compose.yml` file.
- Use `docker-compose up` to start the system.

I'll show you the workflow in action using **two real-life examples**. First, I'll demonstrate the basic syntax of `docker-compose.yml` and how to link containers. The second example will show you how to manage an application's configuration data across development and testing environments.

Example 1: Basic Structure

The syntax of the `docker-compose.yml` file closely reflects the underlying Docker operations. To demonstrate this, I'll build a container from Redis Commander sources and connect it to the Redis database.

Implementation

Let's create a project with the following structure:

STRUCTURE

```
example1
├── commander
│   └── Dockerfile
└── docker-compose.yml
```

Now let's follow the workflow. Define the Dockerfile that builds Redis Commander (see source), and then create `docker-compose.yml`.

DOCKER-COMPOSE.YML

```
backend:
  image: redis:3
  restart: always

frontend:
  build: commander
  links:
    - backend:redis
  ports:
    - 8081:8081
  environment:
    - VAR1=value
  restart: always
```

Now execute:

COMMAND LINE

```
docker-compose up -d
```

After that, point your browser to http://localhost:8081/. You should see the user interface of Redis Commander that's connected to database.

Note:

- The command `docker-compose up -d` has the same effect as the following sequence of commands:

COMMAND LINE

```
docker build -t commander commander
docker run -d --name frontend -e VAR1=value -p 8081:8081
   --link backend:redis commander
```

- Each service needs to point to an image or build directory; all other keywords (`links, ports, environment, restart`) correspond to `docker` options.
- `docker-compose up -d` builds images if needed.
- `docker-compose ps` shows running containers.

COMMAND LINE

```
$ docker-compose ps
Name                        State                 Ports
------------------------------------------------------------------
example1_backend_1          Up                    6379/tcp
example1_frontend_1 ...     Up                    0.0.0.0:8081->8081/tcp
```

- `docker-compose stop && docker-compose rm -v` stops and removes all containers.

Example 2: Configuration Pack

Now let's deploy an application to two different environments — development and testing — in such a way that it would use different configuration depending on the target environment.

Implementation

One of our possible options is to wrap all environment-specific files into separate containers and inject them into the environment if needed. You can download the source files for this example here.

Our project will have the following structure:

STRUCTURE

```
example2
├── common
│   └── docker-compose.yml
├── development
│   ├── content
│   │   ├── Dockerfile
│   │   └── index.html
│   └── docker-compose.yml
└── testing
├── content
│   ├── Dockerfile
│   └── index.html
└── docker-compose.yml
```

And again let's follow the workflow.

First let's create Dockerfiles (`{development,testing}/content/Dockerfile`) that wrap environment-specific content in containers. In this case, just to illustrate the mechanism, containers will contain only an `index.html` file with an environment-specific message (e.g., **"You are in development!"**).

DOCKERFILE

```
FROM tianon/true

VOLUME ["/usr/share/nginx/html/"]
ADD index.html /usr/share/nginx/html/
```

Note that because we don't need an operating system, the images are built on top of the smallest fully functional image: `tianon/true`. The image is originally built FROM scratch and contains only `/true` binary.

Now let's create `docker-compose.yml` files. `common/docker-compose.yml` contains shared services of application. In this case, it's just one service: `nginx` server with the definition of port mappings:

COMMON/DOCKER-COMPOSE.YML

```
web:
  image: nginx
  ports:
    - 8082:80
```

The definition will be used through inheritance.

Each environment needs its own `docker-compose.yml` file (`development/docker-compose.yml` and `testing/docker-compose.yml`) that injects the correct configuration pack.

DEVELOPMENT/DOCKER-COMPOSE.YML

```
web:
  extends:
    file: ../common/docker-compose.yml
    service: web
  volumes_from:
    - content

content:
  build: content
```

Executing the following commands:

COMMAND LINE

```
cd development
docker-compose up -d
```

Point a browser to http://localhost:8082/. You should be able to see the **"You are in development!"** message. You can activate testing environment setup by executing:

COMMAND LINE

```
docker-compose stop
cd ../testing
docker-compose up -d
```

When you reload the browser, you should see the **"You are in testing!"** message.

Note:

- `web` service inherits from `common/docker-compose.yml`. Its original definition is extended by the `volumes_from` directive that maps volumes from `content` container. This is how the application gets access to environment-specific configuration.
- After start, `content` container executes the `true` command and immediately exits, but its data remains exposed to the `web` container, which stays active.

Summary

The simplicity of Docker Compose, and the fact that an entire configuration can be stored and versioned along with your project, makes it a very helpful utility for development. For more information, refer to official documentation at https://docs.docker.com/compose/.

Share this Codeship Guide

Further Reading

DOCKER

- The Future is Containerized
- Container Operating Systems Comparison
- Building a Minimal Docker Container for Ruby Apps

CONTINUOUS DELIVERY

- Running a MEAN web application in Docker containers on AWS
- Running a Rails Development Environment in Docker
- Testing your Rails Application with Docker

More Codeship Resources.

CODESHIP

Share this Codeship Guide

About Codeship.

Codeship is a hosted Continuous Integration service that fits all your needs. Codeship Basic provides pre-installed dependencies and a simple setup UI that let you incorporate CI and CD in only minutes. Codeship Pro has native Docker support and gives you full control of your CI and CD setup while providing the convenience of a hosted solution.

Codeship Basic

A simple out-of-the-box Continuous Integration service that just works.

Starting at $0/month.

- Works out of the box
- Preinstalled CI dependencies
- Optimized hosted infrastructure
- Quick & simple setup

LEARN MORE

Codeship Pro

A fully customizable hosted Continuous Integration service.

Starting at $0/month.

- Customizability & Full Autonomy
- Local CLI tool
- Dedicated single-tenant instances
- Deploy anywhere

LEARN MORE