KARL HUGHES
SOFTWARE ENGINEER

7 Ways to Improve Your Test Suite with Docker

About the Author.

Karl Hughes is a startup fanatic, engineering team lead, and CTO at The Graide Network. He is also very involved in the Chicago tech community and enjoys writing, speaking at conferences, and contributing to open source software.

Codeship is a fully customizable hosted Continuous Integration and Delivery platform that helps you build, test, and deploy web applications fast and with confidence.

Learn more about Codeship here.

7 Ways to Improve Your Test Suite with Docker

I try to be disciplined in writing tests, but it's usually not long before something gets in the way.

Integration-testing interdependent APIs, verifying unusual server configurations, and seeding complex data make testing large applications tough. While plenty of solutions exist, I have found myself leaning on Docker more and more in the past year.

ABOUT THIS EBOOK

In this eBook we will talk about 7 ways of using Docker to improve your test suite.

Why Docker for Testing?

If you aren't familiar with Docker, Barry Jones has written a great primer for Codeship's blog. At a high level, Docker makes it faster and easier to spin up services using a variety of configurations. This translates into a number of advantages when running your tests in Docker:

- **You can keep server configuration in code.** Any developer or CI environment will run the exact same setup.
- **It takes just a few seconds to set up or tear down an environment in Docker.** You can quickly test your code with multiple versions of the language or run integration tests with different combinations of supporting services.
- **Docker works great with continuous integration tools.** This allows you to run your CI more often and catch catch issues faster.

Manuel Weiss covers some other great reasons that Docker makes testing more efficient here, but in the remainder of this article, I'll focus on specific examples of how you can use Docker to improve your test suite. These examples are available in more detail on GitHub, so feel free to clone the repository if you want to follow along.

Share this Codeship Guide

1: Working with Older Versions of the Language

Last year, I spent some time working with Maurits van der Schee's "php-crud-api", an open-source PHP project that creates a RESTful API from an existing relational database. The project officially supports PHP 5.3 through 7.1 in combination with Postgres 9.1+, MySQL 5.5+, MS SQL Server 2012+, and SQLite 3, which means there are dozens of combinations of PHP and databases to potentially support.

If you've ever maintained an open-source package, you know how tough it can be to prevent breaking changes in older versions of the language. Docker makes this easier as you no longer have to set up and run multiple virtual machines to verify your application works. For example, let's say you have a PHP function that will sort an array of objects:

CODE

```
< ?php

function sortArrayOfObjects(string $field, array $objects): array
{
    usort($objects, function ($a, $b) use ($field) {
        return $a->{$field} < => $b->{$field};
    });

    return $objects;
}
```

This function uses some of the newest features from PHP 7, including scalar type hinting, return type declarations, and the "spaceship" operator.

In order to find out if this code will work with an older version of PHP, simply run the test (see the test code here) from within a Docker container:

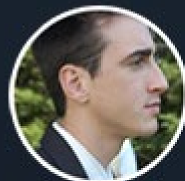
CODE

```
# Running in PHP 5.6 leads to a syntax error
$ docker run --rm -v $(pwd):/app -w /app php:5.6 vendor/bin/phpunit index.php
> Parse error: syntax error, unexpected ':', expecting '{' in /app/index.php on line 12

# Running in PHP 7.0 works
$ docker run --rm -v $(pwd):/app -w /app php:7.0 vendor/bin/phpunit index.php

> PHPUnit 5.7.27 by Sebastian Bergmann and contributors.
> ...
> OK (1 test, 2 assertions)
```

If you're not familiar with the Docker CLI or `run` command, check out the official documentation or this short tutorial on running a PHP script in Docker.



CHESLEY BROWN, INVISION APP

From a vision perspective, Codeship's Continuous Integration service has hit the nail on the head.

LEARN MORE

2: Testing Required Extensions

Another common problem that developers run into is knowing how their code will run on a new server. Even if you know which version of the programming language is running, you may not have all of the same extensions available. For example, PHP does not necessarily include the MySQLi extension, so if your app needs to connect to a MySQL database, this will be a big problem.

Let's say you wrote a function that connects to your database and returns the version of MySQL:

CODE

```
function getMysqlVersion(): string
{
    $link = mysqli_connect(DB_HOST, DB_USER, DB_PASS);

    if (mysqli_connect_errno()) {
        throw new Exception("Connection failed: %s\n" . mysqli_connect_error());
    }

    $version = mysqli_get_server_version($link);

    mysqli_close($link);

    return $version;
}
```

If you test this (see test here) in the default PHP 7.2 Docker image, it will fail:

CODE

```
$ docker run --rm -v $(pwd):/app -w /app --link database php:7.2 vendor/bin/phpunit index.php
> Error: Call to undefined function mysqli_connect()
```

Share this
Codeship Guide

This tells you that the code requires an extension that PHP doesn't have by default, but you can include it if you build your own Docker Image. To build a custom image, add a `Dockerfile` to the project that adds the MySQLi extension:

CODE

```
1  FROM php:7.2
2  RUN docker-php-ext-install mysqli
```

Then build a new image and run your test in the new image:

CODE

```
1  # Build a custom Docker image from our Dockerfile
2  $ docker build . -t php-72-mysqli
3
4  # Run our test using the new image
5  $ docker run --rm -v $(pwd):/app -w /app --link database php-72-mysqli vendor/
   bin/phpunit index.php
```

Now your tests will pass, and you can commit this Dockerfile to ensure that other developers know this extension is required. If all the developers on your team commit to local development in Docker, it greatly reduces the all-to-common "works on my machine" excuse.

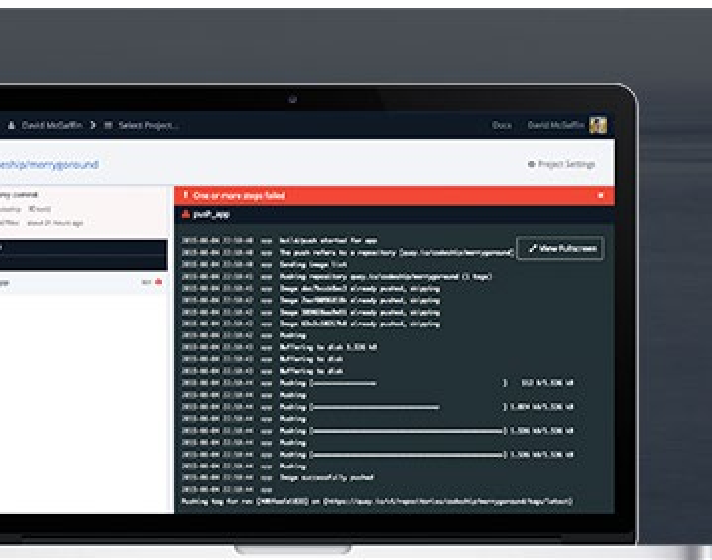
3: Integration Testing Different Services

Another challenge I ran into when testing PHP Crud API was that it had to support four different databases and many versions of each. It would be nearly impossible to test every combination in virtual machines, but Docker makes spinning up and connecting to different databases very easy.

For example, if you want to run your tests in MySQL 5.6 and 5.7, you can start two containers:

CODE

```
1  # Starts a MySQL 5.6 container
2  docker run --name mysql-56 --rm -d -e MYSQL_ALLOW_EMPTY_PASSWORD=true -e MYSQL_
   DATABASE=test mysql:5.6
3
4  # Starts a MySQL 5.7 container
5  docker run --name mysql-57 --rm -d -e MYSQL_ALLOW_EMPTY_PASSWORD=true -e MYSQL_
   DATABASE=test mysql:5.7
```



START WITH THE $0 PLAN

Sign up for Codeship's free plan.

Get 100 builds per month and unlimited private projects for free.

CLICK HERE TO GET STARTED

The following PHP code creates a JSON column on your table (see the whole file on GitHub):

CODE

```
/**
 * Creates a table with JSON fields, a feature only available in MySQL 5.7+.
 */
function createTableWithJsonFields(): bool
{
    $link = getLink();
    mysqli_select_db($link, DB_DATABASE);
    $createTableQuery = 'CREATE TABLE IF NOT EXISTS ' .DB_TABLE. ' (id INT, json_
field JSON);';
    if (!$result = mysqli_query($link, $createTableQuery)) {
        throw new Exception("Table could not be created.");
    }
    mysqli_close($link);
    return $result;
}
```

The test for this code makes sure the table was successfully created. Now you can run your tests against the MySQL 5.6 and MySQL 5.7 containers:

CODE

```
# Connect to the MySQL 5.6 container and run tests
docker run --rm -v $(pwd):/app -w /app --link mysql-56 php-72-mysqli vendor/bin/
phpunit index.php
> Exception: Table could not be created.

# Connect to the MySQL 5.7 container and run tests
docker run --rm -v $(pwd):/app -w /app --link mysql-57 php-72-mysqli vendor/bin/
phpunit index.php
> OK (1 test, 3 assertions)
```

This tells you that the application will run with a MySQL 5.7 database, but not 5.6, and that can save you a lot of time debugging. You can use this same strategy to verify

that updating your web server or caching service won't break your core codebase.

4: Seeding Data with Volumes

While running multiple versions of your language and services may be helpful for open-source projects, you might not have to worry about those situations in your day-to-day work. That said, there are still plenty of cases where Docker can help your test suite. For example, what if you need to seed the same data in your database every time you run your tests?

At The Graide Network, we run a suite of microservices that can seed themselves independently, but when we want to run end-to-end tests, it's a little trickier. We use a SQL file mounted into our database container via a Docker volume to make sure that the database is set up the same every time we run our tests.

For example, if you check some dummy data into your version control, you can make sure all developers use the same data to start their MySQL container:

CODE

```
1  docker run --name database --rm -d -e MYSQL_ALLOW_EMPTY_PASSWORD=true -v $(pwd)/
   data:/var/lib/mysql mysql:5.7
```

Share this Codeship Guide

The command above mounts the `./data` directory into `/var/lib/mysql`, the folder that hosts MySQL's data files, so each time you start up this database, the same data will be present in the container.

5: Docker Compose for End-to-End Tests

Unit and integration tests might make up the bulk of your test suite, but end-to-end tests should also play an important role. At The Graide Network, we run a few dozen microservices, which makes end-to-end testing especially challenging. Fortunately, Docker Compose has made testing our whole application much easier.

While many options exist for running end-to-end tests, I like Nightwatch.js, a Node-powered browser testing tool. In order to run your end-to-end tests with Docker Compose and Nightwatch, you'll need to create a `docker-compose.yml` file in the root directory of your project:

CODE

```
version: '3.5'
services:
  nginx:
    image: nginx:alpine
    volumes:
      - ./app:/usr/share/nginx/html
  chromedriver:
    image: blueimp/chromedriver
```

CODE

```
 9      environment:
10        - VNC_ENABLED=true
11        - EXPOSE_X11=true
12      ports:
13        - 5900:5900
14    nightwatch:
15      image: blueimp/nightwatch:0.9
16      depends_on:
17        - chromedriver
18        - nginx
19      environment:
20        - WAIT_FOR_HOSTS=nginx:80 chromedriver:4444 chromedriver:6060
21      volumes:
22        - ./:/home/node
```

This Compose file starts three Docker containers: an NGINX server, a headless Chrome instance, and a Nightwatch test runner. Assuming your folders and `nightwatch.json` file are set up properly (check the example to see how you should do it), you can run your Nightwatch suite with one single Docker Compose command:

CODE

```
1  # Run the nightwatch tests
2  docker-compose run nightwatch
```

6: Spin Up Multiple Services

What if your test suite requires a more complicated set of apps? For example, a Node backend app with a single page Javascript and HTML frontend?

Share this
Codeship Guide

Docker Compose can handle this setup as well. Containers in a Docker Compose file are automatically networked together, you can use this method to start a Node API and NGINX webserver for the frontend:

CODE

```yaml
version: '3.5'
services:
  api:
    build: .
    volumes:
      - ./api:/app
  nginx:
    image: nginx:alpine
    volumes:
      - ./app:/usr/share/nginx/html
  chromedriver:
    image: blueimp/chromedriver
    environment:
      - VNC_ENABLED=true
      - EXPOSE_X11=true
    ports:
      - 5900:5900
  nightwatch:
    image: blueimp/nightwatch:0.9
    depends_on:
      - chromedriver
      - nginx
      - api
    environment:
      - WAIT_FOR_HOSTS=nginx:80 chromedriver:4444 chromedriver:6060 api:3000
    volumes:
      - ./:/home/node
```

When you run this set of containers using the same Docker Compose command we used before, they will automatically be linked and your frontend will be able to make API calls to the backend Node app using a

DNS record that Docker sets. In the example above, this means your frontend will call the backend like this:

CODE

```
fetch('http://api:3000/')
  .then(resp => resp.json())
  .then(function(data) {
    // Do something with the data
  });
```

Docker Compose has some other great features for complex applications like advanced networking options and extensible Compose files, so be sure to read the docs for more.

7: Automate Your Continuous Integration

Finally, automation is one of the biggest advantages to running your test suite in Docker. One way to do this is with Codeship Pro, which uses Docker to run your application and test suite. Once you build a Docker Compose file that works locally, you can copy and tweak that file to create a Codeship Services configuration file. Add a Codeship Steps file to your repository and your app is ready to go.

To test out your Codeship CI configuration locally (and prevent wasting builds debugging configuration issues), you can download and run their handy CLI tool Jet.

Because the Codeship Services file can run many containers at once, this method of automating tests makes it possible to automatically run your app in different version of the language, using different databases, or with anciliarry services connected.

Docker has helped improve our test suite at The Graide Network as well as many open-source projects I've worked on. If you have your own tips for using Docker to improve your test suite, let me hear about them on Twitter as I'm always looking to learn more.

CODESHIP by CloudBees

Share this Codeship Guide

More Codeship Resources.

EBOOKS

Why Containerization is the Future.

In this eBook we will learn why the Container Stack "is the future", as so many people have proclaimed.

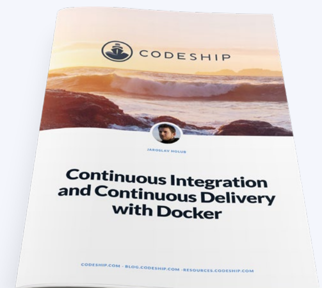
Download this eBook

EBOOKS

CI and CD with Docker.

In this eBook we take a look at how to set up a Continuous Delivery Pipeline with Docker and containers.

Download this eBook



EBOOKS

An Introduction to Deploying Docker Apps with Codeship Pro.

In this eBook we will walk you through building and deploying applications with Codeship Pro.

Download this eBook

CODESHIP by CloudBees

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.

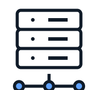
Codeship Basic

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

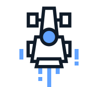
Starting at $0/month.

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