Bit Press

DOCKER
For PHP Developers

BY PAUL REDMOND

A Guide to Using Docker for PHP Development

Chapter 4: Development Tools

Now that you have a foundation of running a LAMP application with Docker, we are going focus this chapter on your development workflow.

Starting development with Docker will be a bit of a transition for you. When I first started using Docker for development, I felt uncomfortable with the workflow and environment. My goal is to ease that transition for you as you start developing applications in Docker. We are going to go over managing environment variables, setting up Xdebug for debugging and profiling, and making sure development tools don't affect production.

If you are following along, we will continue with the code from Chapter 3; just continuing to work on the same project will do.

Environment configuration

Each environment your code runs in has different needs. Environment variables help you create flexible Docker images by allowing you to configure your applications for different environments. For example, in development, you might be connecting to a database within a container, and in production, use something like Amazon Relational Database Service (RDS). Environment variables also help keep sensitive data out of your codebase.

How do we use environment variables with a container?

You've already seen an example: the MariaDB container from Chapter 3 uses environment variables to configure the database name and root password (Listing 4.1).

Listing 4.1: Example Environment configuration in docker-compose.yml

```
services:
  mariadb:
    # ...
    environment:
      - MYSQL_DATABASE=dockerphp
      - MYSQL_ROOT_PASSWORD=password
```

If you are not using Docker Compose, you can pass environment variables using the *-e* flag or with a separate file using `docker run`:

Listing 4.2: Setting Environment Variables with *docker run*

```
# Passing environment variables with the -e|--env flag
$ docker run -e MYSQL_ROOT_PASSWORD=password --name my-db \
  -d mariadb

# Using an external file in the same path
$ docker run --env-file .docker.env --name my-db -d mariadb
```

In Docker Compose, you can also pass environment variables from an external file with the env_file configuration option. Using an external file allows developers to maintain their own unversioned file, and provides a starting point that works. Here's an example:

```
# An env_file configuration example
services:
  mariadb:
    env_file: .docker.env
```

Setting up Environment Variables

We are going to walk through setting up environment variables for development. We will use an external file to house most of our Docker variables for development, and along the way, you'll learn how configure Xdebug with environment variables.

Before we get to Xdebug, we are going to create our Docker environment file and an example file that you can version as a good starting point for your application (Listing 4.3):

Listing 4.3: Create the Docker Environment Files

```
$ touch .docker.env.example .docker.env

# Set an example value for demonstration
$ echo "HELLO=WORLD" >> .docker.env.example
$ cat .docker.env.example > .docker.env

# Ignore the .docker.env file when using VCS
$ echo ".docker.env" >> .gitignore
```

In Listing 4.3, we created an example environment file and a local environment file. Later on, when a new developer first checks out your repository, they will copy the versioned `.docker.env.example` file to the ignored `.docker.env` with `cp .docker.env.example .docker.env`. We set *HELLO=WORLD* to verify that our external file is working with Compose.

Let's add this environment file to *docker-compose.yml* so we can test it out (Listing 4.4):

Listing 4.4: Adding the .docker.env file

```
version: "2"
services:
  app:
    build: .
    depends_on:
      - mariadb
    ports:
      - "8080:80"
    volumes:
      - .:/srv
    links:
      - mariadb:mariadb
    env_file: .docker.env
  mariadb:
    image: mariadb:10.1.21
    ports:
      - "13306:3306"
    environment:
      - MYSQL_DATABASE=dockerphp
      - MYSQL_ROOT_PASSWORD=password
```

You added the *env_file* configuration pointing to the unversioned *.docker.env* file we created. Now you can manage your environment for Docker through this file, and when you run *docker-compose*, the values will take effect in the container. Having your own local file keeps secrets out of the codebase, and allows you to tweak them however you see fit.

Trying out Environment Variables

We need to confirm that our file is working as expected. Let's run the container with the new configuration option and verify that the environment variable has been set (Listing 4.5).

Listing 4.5: Verify the Docker Env File is Working

```
$ docker-compose up -d && docker ps # note the app container id
$ docker exec -it 36b079b1aa04 bash
root@36b079b1aa04:/var/www/html# echo $HELLO
WORLD

$ docker-compose stop # Shut down the docker containers
```

You should see "WORLD" printed when you echo the *$HELLO* environment variable inside of the running container.

Environment variables are an excellent way to make your Docker setup more flexible for applications like Laravel. For example, Laravel uses phpdotenv to read environment variables through a *.env* file. However, if environment variables are defined on the system, phpdotenv uses the system values instead of the value from the .env file, meaning we can override values in the .env file with container environment variables.

Xdebug Setup

Now that you have the basic idea of how to manage the environment variables with Docker, we are ready to move on to installing Xdebug. My favorite use of environment variables in Docker is defining PHP INI variables that make PHP configuration files easier to modify and more flexible. This approach is compelling for developers wanting to customize the way Xdebug works during runtime without changing the Docker image or having to rebuild each time changes occur.

I don't like spending a ton of time with debuggers. In my own opinion, if you are doing a lot of debugging, it might mean a weak test suite or overly-complicated code; but I'm just generalizing. I reach for Xdebug at least once a day, but when I moved my development to Docker, it was a bit tricky getting debugging working consistently. Let's work through it together.

After this section, setting up Xdebug with Docker should be a breeze. Along the way, we'll also ensure that the Docker builds don't have a trace of the Xdebug module when you ship your code to non-development environments.

Installing Xdebug

The first thing we are going to tackle is installing the Xdebug module in our container. Xdebug is a PECL extension, and the official PHP Docker image provides the *pecl* command to install PECL modules. Our updated Dockerfile will install Xdebug with PECL and then enable the extension (Listing 4.6).

Listing 4.6: Installing and Enabling Xdebug

```
FROM php:7.1.0-apache

LABEL maintainer Paul Redmond <paul@bitpress.io>
COPY .docker/php/php.ini /usr/local/etc/php/
COPY . /srv
COPY .docker/apache/vhost.conf /etc/apache2/sites-available/000-
default.conf

RUN docker-php-ext-install pdo_mysql opcache \
    && pecl install xdebug-2.5.1 \
    && docker-php-ext-enable xdebug

RUN chown -R www-data:www-data /srv/app
```

Make sure you stop any running containers with *docker-compose stop* and then rebuild the app service with *docker-compose build app*.

You might recall specifying "app" when running "docker-compose build" from Chapter 3. Providing an argument allows you to build specific services defined in your docker-compose.yml file. Running without any arguments will build all images.

With our latest build completed, let's run the container to make sure our installation worked (Listing 4.7):

Listing 4.7: Verify that Xdebug was installed and enabled

```
$ docker-compose build app
$ docker-compose up -d
# Note the container id from `docker ps`
$ docker exec -it 00fded44032c bash
```

```
# Inside the container...
root@00fded44032c:/var/www/html# php -i | grep ^extension_dir
extension_dir => /usr/local/lib/php/extensions/no-debug-non-zts-
20160303
root@00fded44032c:/var/www/html# ls -la
/usr/local/lib/php/extensions/no-debug-non-zts-20160303 | grep
xdebug
-rw-r--r-- 1 root staff 1086808 Feb 28 06:03 xdebug.so

# Lists all the lines mentioning xdebug in the ini settings
root@00fded44032c:/var/www/html# php -i | grep xdebug
root@00fded44032c:/var/www/html# php -m | grep xdebug
xdebug
```

You should see the xdebug module when you run *php -m* in the running container, meaning that Xdebug is now installed. We are now ready to configure Xdebug!

Configuring Xdebug

Installing Xdebug in the container provides an xdebug.ini file containing configuration for Xdebug. To find out where this INI files is located, run *php --ini* inside the container (Listing 4.8):

Listing 4.8: View the xdebug ini file

```
# Still inside the running container
root@00fded44032c:/var/www/html# php --ini
Configuration File (php.ini) Path: /usr/local/etc/php
Loaded Configuration File:         /usr/local/etc/php/php.ini
Scan for additional .ini files in: /usr/local/etc/php/conf.d
Additional .ini files parsed:
/usr/local/etc/php/conf.d/docker-php-ext-opcache.ini,
/usr/local/etc/php/conf.d/docker-php-ext-pdo_mysql.ini,
/usr/local/etc/php/conf.d/docker-php-ext-xdebug.ini
```

```
# Output the contents of the xdebug file
root@00fded44032c:/var/www/html# cat
/usr/local/etc/php/conf.d/docker-php-ext-xdebug.ini
zend_extension=/usr/local/lib/php/extensions/no-debug-non-zts-
20160303/xdebug.so
```

The Xdebug file just enables the extension, and the rest of the Xdebug values are defaults. We need to configure a few settings, so let's create a new `xdebug-dev.ini` file for our customizations. We will start out with a static configuration, verify that it works, and then move over to an environment-driven configuration.

Create the Xdebug file in your codebase at *.docker/php/xdebug-dev.ini* with the following contents (Listing 4.9).

Listing 4.9: Adding an xdebug ini file

```
[xdebug]
xdebug.default_enable=1
xdebug.remote_autostart=1
; remote_connect_back is not safe in production!
xdebug.remote_connect_back=1
xdebug.remote_port=9001
xdebug.remote_enable=1
xdebug.idekey=DOCKER_XDEBUG
```

Go ahead and familiarize yourself with these Xdebug settings if you need a refresher. There are a few settings I need to explain and how they relate to running Xdebug in a container.

I like enabling *remote_autostart* so that Xdebug automatically tries to connect without a GET, POST, or COOKIE variable. The *remote_connect_back* setting, while not safe in production, is convenient because you don't have to worry about the *remote_host* setting. The remote connect back setting will try to connect to the client that made the

HTTP request. I select a non-default remote port (9001) because I have PHP-FPM running locally, so I need a different port mapping.

For the Xdebug settings to take effect, we need to copy the new INI file into the container (Listing 4.10).

Listing 4:10: Copy the Xdebug INI file into the container

```
FROM php:7.1.0-apache

LABEL maintainer Paul Redmond <paul@bitpress.io>
COPY .docker/php/php.ini /usr/local/etc/php/
COPY . /srv/app
COPY .docker/apache/vhost.conf /etc/apache2/sites-available/000-
default.conf
RUN docker-php-ext-install pdo_mysql opcache \
    && pecl install xdebug-2.5.1 \
    && docker-php-ext-enable xdebug

COPY .docker/php/xdebug-dev.ini /usr/local/etc/php/conf.d/xdebug-
dev.ini

RUN chown -R www-data:www-data /srv/app
```

With our current configuration file saved, rebuild the container with *docker-compose build app*. Once you finish building the container, you should see your Xdebug settings by adding *phpinfo()* to the top of your *public/index.php* file. You should see the specific settings we changed in our *xdebug-dev.ini* file reflected in the output.

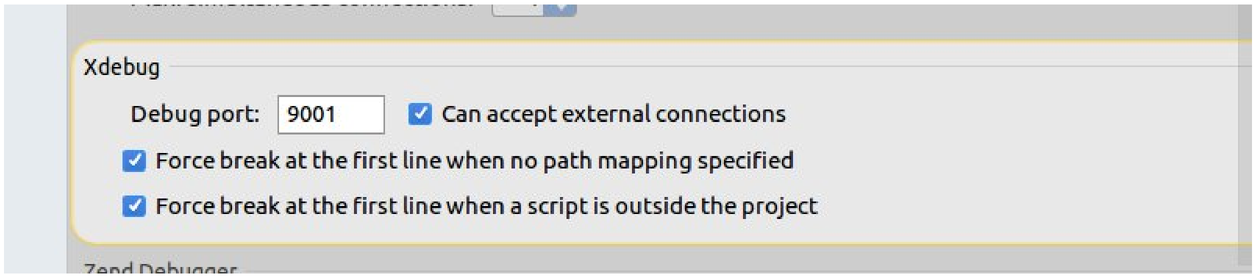
With the Xdebug configuration updated, it's time to verify that we are able to connect to Xdebug with our editor.

Setting up PhpStorm

We are going to use PhpStorm to connect to Xdebug in this chapter, an excellent commercial IDE for PHP. I prefer PhpStorm's debugging UI and capabilities, but adapting these instructions to any Xdebug client should be fairly straightforward.

If you are not familiar with using Xdebug with PhpStorm, I recommend following the zero-configuration web application debugging article. Another good article is debugging PHP Web applications with run debug configurations. You should be able to get it working by following along here, but I recommend going through the article if you need a more thorough walkthrough.

The first thing you'll do is open PhpStorm's preferences and navigate to *Languages & Frameworks > PHP > Debug*. Because we specified port *9001* in our Xdebug configuration, we need to change PhpStorm to use the right port as well (Figure 4.1):


Xdebug

Debug port: 9001 ☑ Can accept external connections

☑ Force break at the first line when no path mapping specified

☑ Force break at the first line when a script is outside the project

Figure 4.1: Tweaking PhpStorm Xdebug Port

The next step is setting up a run configuration and server. In PhpStorm, click the Xdebug bar (usually in the top right corner) drop-down and click "edit configurations..." (Figure 4.2).

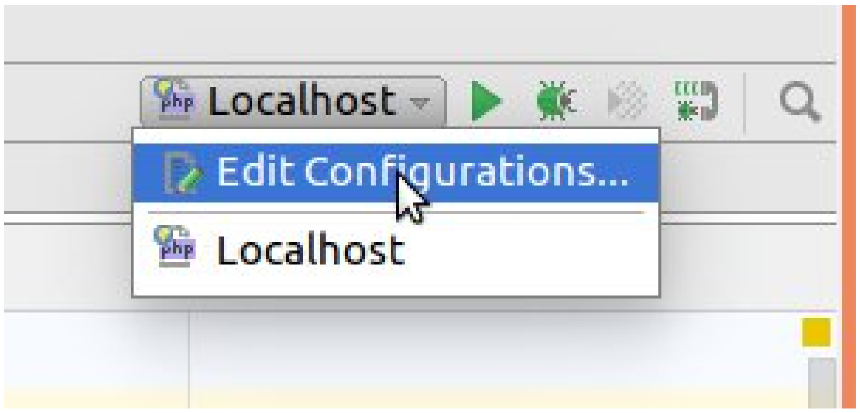
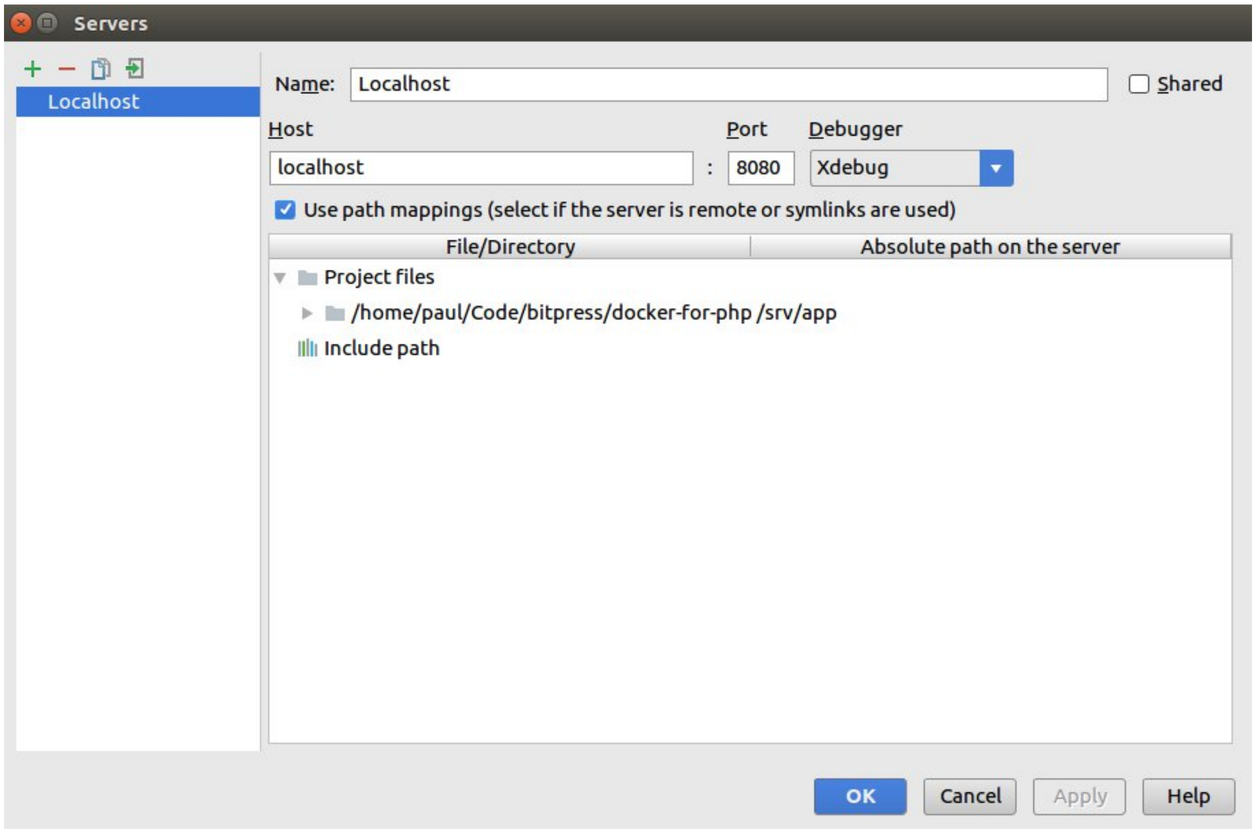


Figure 4.2: Edit run configurations

On the following screen, click the plus (+) button in the top left and select "PHP Web Application" to create a run configuration. Give the application a name (i.e., Localhost), and then click the "..." button to add a server. On the server screen, make sure and enter port *8080*, select Xdebug as the debugger, and map your local project path to the path on the server (*/srv/app*) so Xdebug knows how to map files correctly (Figure 4.3).



Name: Localhost

Host: localhost : Port: 8080 Debugger: Xdebug

Use path mappings (select if the server is remote or symlinks are used) ✓

File/Directory	Absolute path on the server
Project files	
/home/paul/Code/bitpress/docker-for-php	/srv/app
Include path	

Figure 4.3: Add a server to the run configuration

Once you have configured the web server and run configuration, toggle the "start listening for PHP Debug Connections" icon (Figure 4.4):

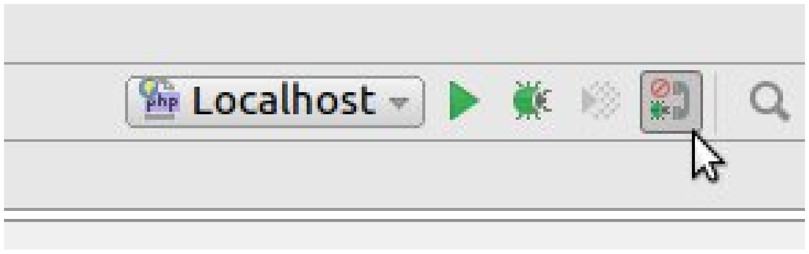


Figure 4.4: Toggle start listening for connections

You should be able to connect to Xdebug by setting a breakpoint or breaking on the first line. If Xdebug doesn't work, try to retrace each step.

Build The Latest Image

Remember to rebuild the latest container with *docker-compose build*, so your INI settings are copied into the Docker image.

Using Environment to Configure Xdebug

The last part of this section is one of my favorite Docker PHP tricks. We are going to use environment variables to set up xdebug INI settings. Using environment variables allows you to make PHP configuration changes without rebuilding an image each time you want to change something.

We will add some default Xdebug settings to the example environment file and to your local *.docker.env* file. When new developers start working with your project, the example file will give them a customizable Xdebug setup they can tweak according to their own preferences.

Open the *.docker.env.example* file and replace the contents of the file with the following variables (Listing 4.11).

Listing 4.11: Add Xdebug Environment Variables to *.docker.env.example*

```
# Xdebug
PHP_XDEBUG_DEFAULT_ENABLE=1
PHP_XDEBUG_REMOTE_AUTOSTART=1
PHP_XDEBUG_REMOTE_CONNECT_BACK=1
PHP_XDEBUG_REMOTE_PORT=9001
PHP_XDEBUG_REMOTE_ENABLE=1
PHP_XDEBUG_IDEKEY=DOCKER_XDEBUG
```

Be sure to update your .docker.env file with the same values so docker-compose will pick up the new variables from your .docker.env file.

Variable Naming Convention

You can name the environment variables anything you'd like. I like to prefix my PHP environment variables with *PHP_* and match the INI configuration name by replacing dots (.) with underscores (_). This convention gives me an idea of configuration values at a glance. For example, *PHP_XDEBUG_REMOTE_AUTOSTART* matches *xdebug.remote_autostart*.

With the variables in place, let's update the *.docker/php/xdebug-dev.ini* file to use them (Listing 4.12):

Listing 4.12: Use Environment Variables in the Xdebug INI File

```
[xdebug]
xdebug.default_enable = ${PHP_XDEBUG_DEFAULT_ENABLE}
xdebug.remote_autostart = ${PHP_XDEBUG_REMOTE_AUTOSTART}
; remote_connect_back is not safe in production!
xdebug.remote_connect_back = ${PHP_XDEBUG_REMOTE_CONNECT_BACK}
xdebug.remote_port = ${PHP_XDEBUG_REMOTE_PORT}
xdebug.remote_enable = ${PHP_XDEBUG_REMOTE_ENABLE}
xdebug.idekey = ${PHP_XDEBUG_IDEKEY}
```

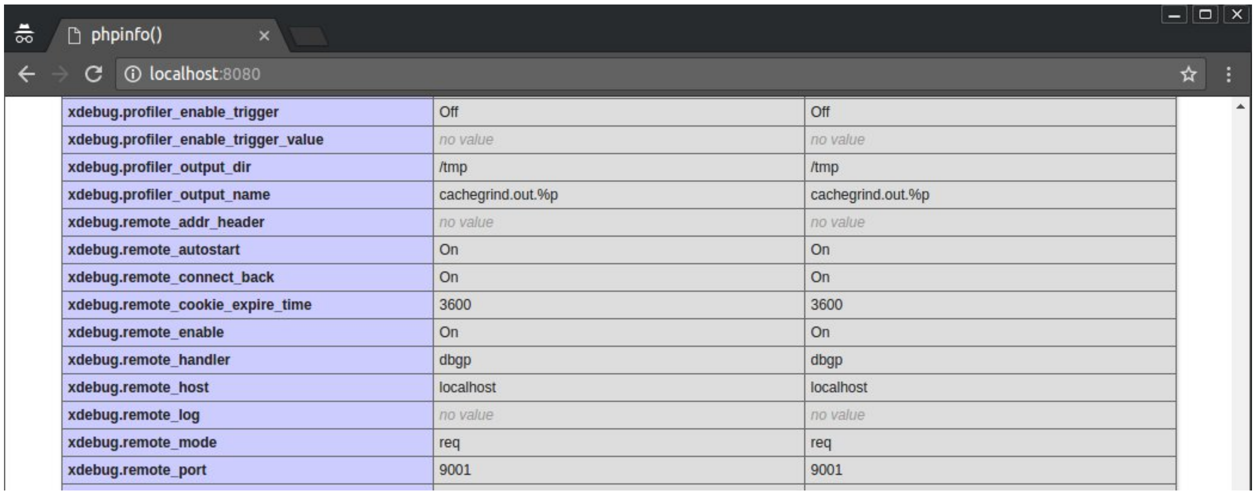
PHP INI configuration files can read from the environment by wrapping environment variables in curly brackets (*${}*). When the container runs, PHP will use the environment values to configure these Xdebug settings.

Before we can test out our changes, we need to rebuild the Docker image so that our latest *xdebug-dev.ini* file gets built into the image (Listing 4.13):

Listing 4.13: Rebuild the Application Image

```
$ docker-compose build app
$ docker-compose up -d
```

With the container running, add "*phpinfo(); exit;*" to the top of your project's *public/index.php* file so you can verify your settings in the browser; you should see something like the following (Figure 4.5):



xdebug.profiler_enable_trigger	Off	Off
xdebug.profiler_enable_trigger_value	no value	no value
xdebug.profiler_output_dir	/tmp	/tmp
xdebug.profiler_output_name	cachegrind.out.%p	cachegrind.out.%p
xdebug.remote_addr_header	no value	no value
xdebug.remote_autostart	On	On
xdebug.remote_connect_back	On	On
xdebug.remote_cookie_expire_time	3600	3600
xdebug.remote_enable	On	On
xdebug.remote_handler	dbgp	dbgp
xdebug.remote_host	localhost	localhost
xdebug.remote_log	no value	no value
xdebug.remote_mode	req	req
xdebug.remote_port	9001	9001

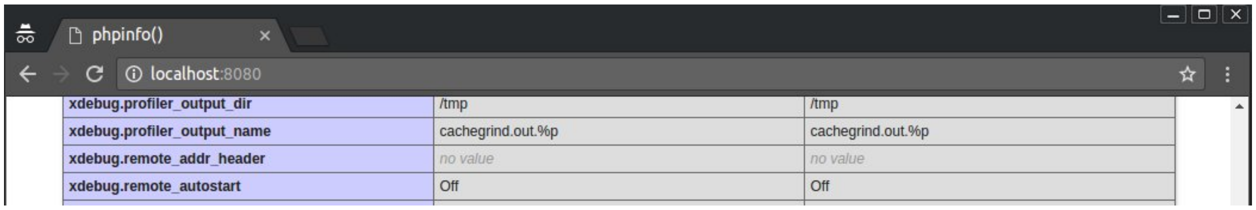
Figure 4.5: Verify Xdebug Settings via Environment

Note the non-standard port of *9001* and *xdebug.remote_autostart On*. At this point, you should have a working environment configuration, but let's verify that it's working as expected by changing a value and confirming that the value is updated in our *phpinfo()* output. Modify the *xdebug.remote_autostart* setting turning it off by default (Listing 4.15).

Listing 4.15: Change Remote Autostart to off in .docker.env

```
PHP_XDEBUG_REMOTE_AUTOSTART=0
```

For environment changes to take effect in a container, you need to restart it with *docker-compose restart*. Once the container is finished rebooting (which should be very quick) you should see that remote autostart is disabled (Figure 4.6).



phpinfo()

localhost:8080

xdebug.profiler_output_dir	/tmp	/tmp
xdebug.profiler_output_name	cachegrind.out.%p	cachegrind.out.%p
xdebug.remote_addr_header	no value	no value
xdebug.remote_autostart	Off	Off

Figure 4.6: Verify Xdebug Remote Autostart Changes

You now have a flexible Xdebug configuration that you can customize without much work. Anyone new to the project will appreciate the easy, flexible setup. You don't have to rebuild the image, and each developer can have his or her own desired settings. Don't go overboard on environment configuration within your containers, but I hope you see how powerful this technique can be!

Xdebug Profiling

Thanks for reading! This is a partial sample of Chapter 4.

Visit https://bitpress.io/docker-for-php-developers/ for more information.