# How To Use Linux for Composer to Deploy to the Cloud

Developers are placed into a tough situation when asked to deploy a working Docker container-based application up to a cloud service.  Traditionally, in addition to hours spent wading through cloud services documentation, a deployment was a messy operation involving a combination of *docker exec*, *zip*, *unzip*, *sftp* and *ssh* to get everything up and running.  In this guide we show you how to deploy directly from your local computer to Linux for PHP Cloud Services  in minutes using a fantastic new tool called *Linux for Composer*.

## What the Heck is Linux for Composer?

At this early stage you may be thinking: *well … I've heard of* Linux, *and I've heard of* Composer … *but what the heck is* Linux for Composer (LfC)?  LfC is yet another incredible tool that comes out of the *Linux for PHP* project.  The brainchild of Foreach Code Factory CEO Andrew Caya, LfC is a PHP package, hosted on github.com and packagist.org, made available via Composer.

Obviously any package residing on packagist.org is not Linux, nor the Linux kernel, but what LfC allows you to do is to define a standard *composer.json* file that includes an extra set of directives that essentially mimics some of the things you can do using Docker Compose.  The main difference, however, is that LfC will proceed to not only build the Docker container for you, but actually upload it to a cloud service using credentials you supply.  So, effectively, as long as your Docker container works locally, with a single command, that same container is reconstructed instantly live on the Internet.  Yes … wow!

## What Do I Need to Start?

So if you're anything like me (e.g. total developer nerd), the next question is … *where do I sign up?* Glad you asked!  (Rhetorical question!)  Before you being the happy next phase of your budding career, you're going to need to have some software tools installed.  If you even got to this point in the article, chances are that you already have most of these items.

Just to be on the safe side, here's the list:

- Docker (https://www.docker.com/)
- Composer (https://getcomposer.org/)
- PHP (https://www.php.net/)
- Git (https://git-scm.com/)
- cURL (https://curl.haxx.se/)

In addition, you'll need one of the following, depending on your operating system:

- Unix/Linux/Mac: A *bash* shell
- Windows 10: PowerShell
- Windows 7 – 8 (64 bit only): Docker Toolbox on Windows
- Windows Vista and earlier: we wish you well, friend! May the *Force* be with you.

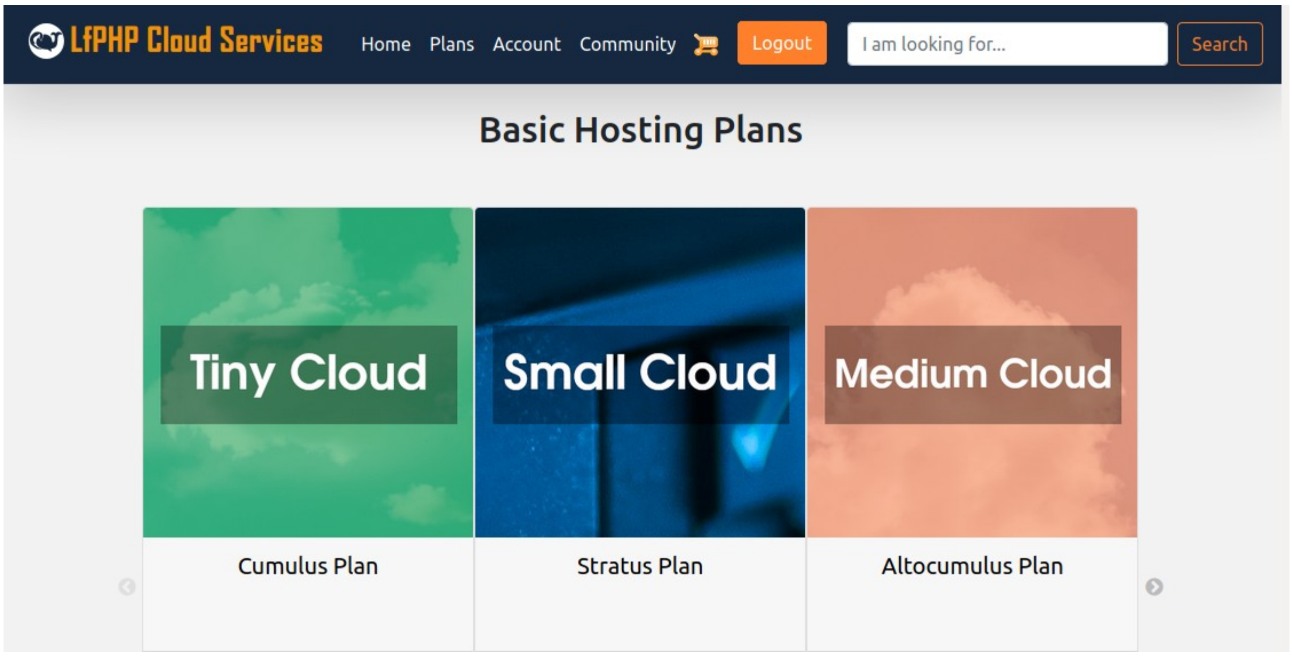
You'll also need to have some sort of software application you're working on: one that's suitable to run in a Docker container. OK … assuming that you've got all the prerequisites in place, and have configured a Dockerfile and any associated init scripts, the next step is to set up the cloud service.

## What About the Cloud Service?

In order to deploy directly to the cloud, you'll need a cloud service with a remote API to which LfC can connect. We're not talking Amazon here folks! There's no way in … heck … they're going to open themselves up to that level of competition! Accordingly it's better to leave the glitzy Amazon department store, and have a look at some of the smaller outfits. Since this article is sponsored by Linux for PHP Cloud Services, by a strange coincidence, they are exactly the outfit we're going to feature in this article! (Full disclosure: this author is a partner in this venture.)

To start off, create an account on Linux for PHP Cloud Services. Go to this page: https://linuxforphp.com/signup, fill in the blanks, and click *Sign Up*. You will need to respond to the email in order to activate your account.

The next step is to login and choose *Plans*. If you hover your mouse over the plan a description pops up. After clicking *Buy Now*, a more detailed description appears with a server location choice. Although from a connectivity perspective it doesn't matter where the server is located, from a legal perspective, if sensitive customer data will be stored, the physical location of the server may be important.

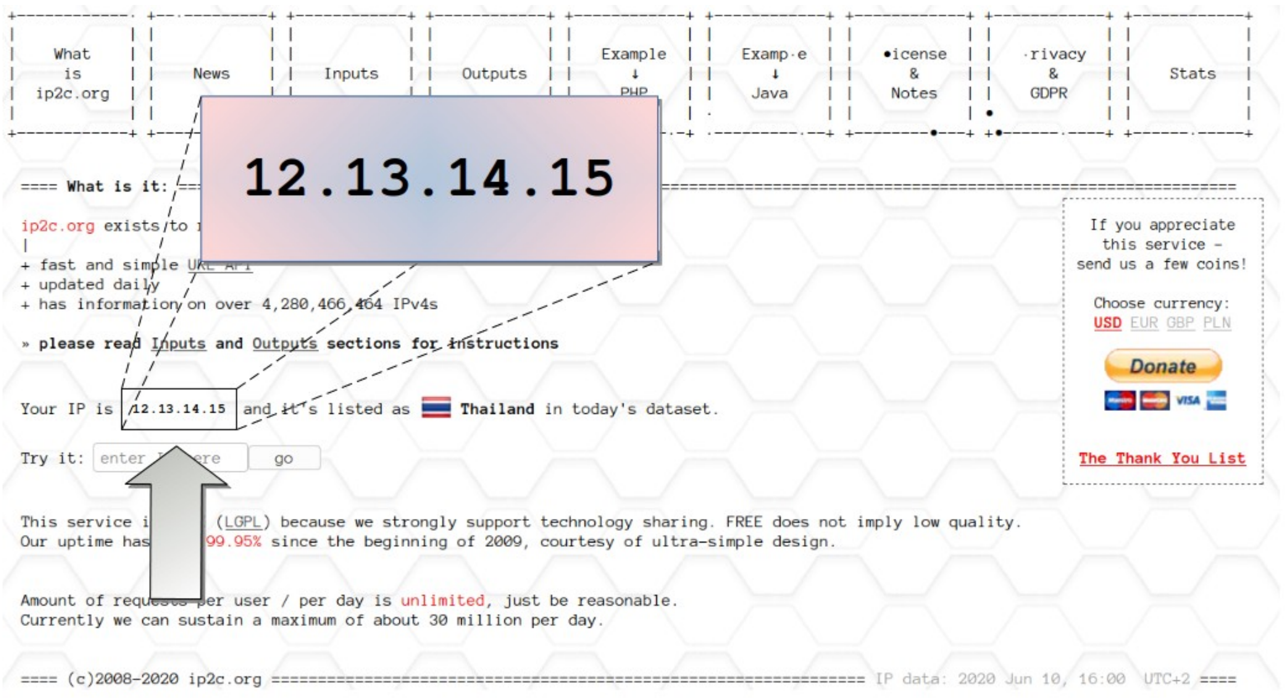


Prices are extremely reasonable and there are free trial offers available.  Promotional codes can be applied when you go to checkout.  Now that you've got a cloud service upon to contain your application, let's have a look at granting deployment access.

## How Do I Make the Deployment Secure?

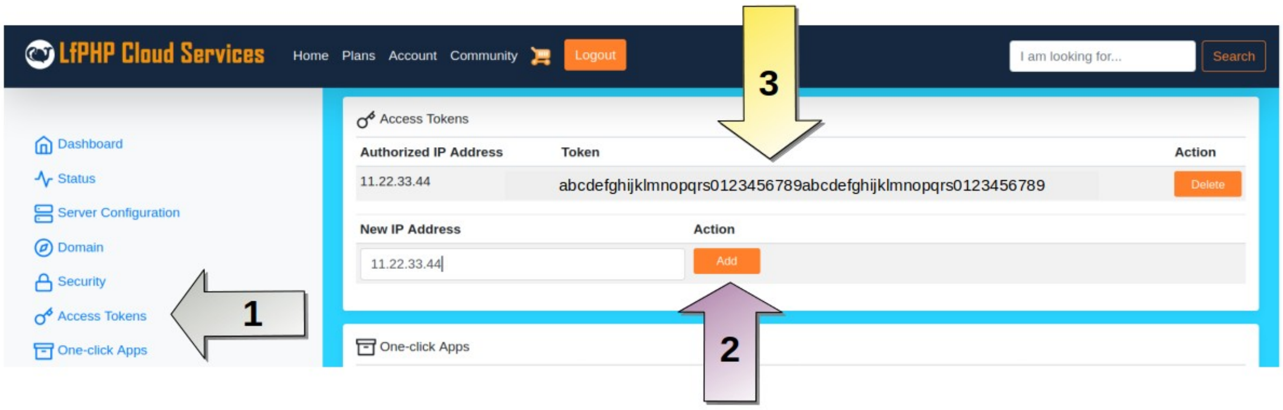
Obviously you don't want any Dick, Jane (or Spot, for that matter!) to be able to deploy code to your website.  Accordingly, you'll need to establish a security token that only allows code deployment from a specific IP address.  Please note that the IP address we refer to here, is *not* the one you see on your own home or office network.  The IP address needed is the one that's visible to the Internet.

To find your Internet-visible IP address, first of all, be sure that you are on the computer from which you plan to deploy.  You can then simply open a browser to this URL: **[http://ip2c.org/](http://ip2c.org/)**.  Look in the middle of the page for a sentence that reads *Your IP is aaa.bbb.ccc.ddd*.



```
==== What is it: ====

ip2c.org exists to
+ fast and simple URL API
+ updated daily
+ has information on over 4,280,466,464 IPv4s

» please read Inputs and Outputs sections for instructions

Your IP is 12.13.14.15 and it's listed as Thailand in today's dataset.

Try it: enter here  go

This service i   (LGPL) because we strongly support technology sharing. FREE does not imply low quality.
Our uptime has   99.95% since the beginning of 2009, courtesy of ultra-simple design.

Amount of requests per user / per day is unlimited, just be reasonable.
Currently we can sustain a maximum of about 30 million per day.

==== (c)2008-2020 ip2c.org ==== IP data: 2020 Jun 10, 16:00  UTC+2 ====
```

You can now return to your *Linux for PHP* cloud services page for the plan you've chosen.  From the left side menu choose *Access Tokens* [1].  Enter your Internet IP address in the input area labeled *New IP Address* [2], and click *Add*.  Your new security access token will appear above [3].



| Authorized IP Address | Token | Action |
|---|---|---|
| 11.22.33.44 | abcdefghijklmnopqrs0123456789abcdefghijklmnopqrs0123456789 | Delete |

| New IP Address | Action |
|---|---|
| 11.22.33.44 | Add |

OK, now that you've got access squared away, the next step is to install Linux for Composer.

## How Do I Install Linux for Composer?

Linux for Composer is installed using *Composer*. The installation itself is quite simple. If you installed *Composer* "globally" on your computer, just create a directory, or from the root of your application project, issue this command:

```
composer require --dev linuxforphp/linuxforcomposer
```

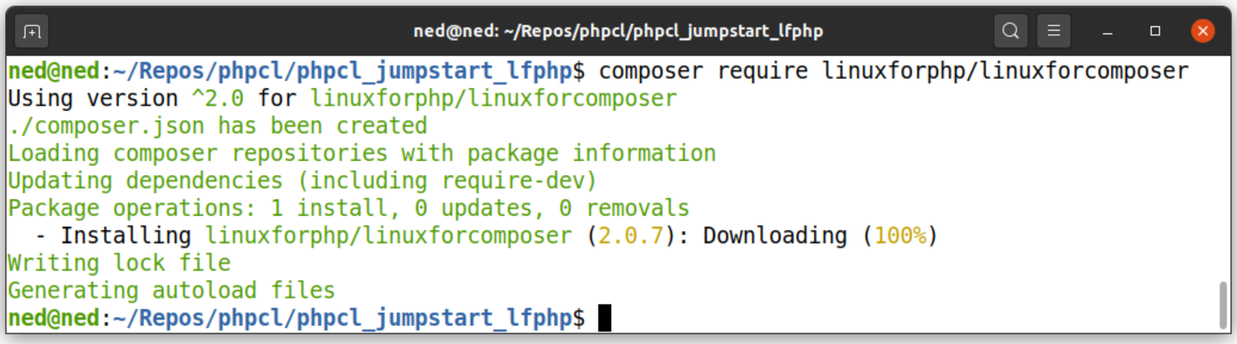
Alternatively, if *Composer* available as a *phar* file, you would use this alternative:

```
php composer.phar require --dev linuxforphp/linuxforcomposer
```

To access Linux for Composer, you have a couple of different options, depending on where you placed the LfC installation:

| If You Placed It ... | Execute Commands Using ... |
|---|---|
| … within your application project | `php vendor/bin/linuxforcomposer.phar <command>` |
| … in a directory ***/some/path*** | `php /some/path/vendor/bin/linuxforcomposer.phar <command>` |

For further information, have a look at the LfC installation documentation. Here is a screenshot of the installation process:



```
ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$ composer require linuxforphp/linuxforcomposer
Using version ^2.0 for linuxforphp/linuxforcomposer
./composer.json has been created
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing linuxforphp/linuxforcomposer (2.0.7): Downloading (100%)
Writing lock file
Generating autoload files
ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$
```

In order to deploy your app, you next need to configure Linux for PHP.

## How Do I Configure Linux for Composer?

*Linux for Composer* looks for a file named **linuxforcomposer.json**. This is the primary configuration file, and serves the same purpose as the *composer.json* file serves for Composer. The complete documentation on all settings in the *linuxforcomposer.json* file is available here:

https://linux-for-composer.readthedocs.io/en/latest/configuration.html

A quick way to get started is to issue the *linuxforcomposer* command with an *init* flag, which generates a sample configuration file. Be sure you are in the main directory of the project you wish to deploy. Issue the command as follows:

```
cd /path/to/project
php vendor/bin/linuxforcomposer.phar --init
```

This creates a default *linuxforcomposer.json* file which you can then modify to suit your application needs.

## Linux for Composer JSON File Keys

The first few lines of the default *linuxforcomposer.json* file appear as follows:

```
{
    "name": "linuxforphp/linuxforcomposer",
    "description": "A Composer interface to run 'Linux for PHP'  … ",
    // other top-level keys not shown
```

The top-level definition keys are summarized here:

| LfC Key | Description |
|---|---|
| `name` | The name of this project. |
| `description` | A short description of this project. |
| `single` | Use this key to deploy a single container. Defines the Docker image and characteristics of the container to be defined, including operational mode, port mappings, volumes and data persistence. |
| `docker-compose` | Use this key if you plan to deploy multiple containers simultaneously. Sub-keys define the location and access information on the online resource containing the *docker-compose.yml* file. |
| `lfphp-cloud` | This key is used in conjunction with either of the above keys. Use it to define the account, username and access token for your Linux for PHP Cloud Services account. |

Within the *single* top-level *linuxforcomposer.json* file definition key, there are two primary sub-keys:

- `single : image {}`
- `single : containers {}`

Let's first have a look at defining a single container using LfPHP.

## Defining a Container Based Upon the LfPHP Docker Image

Within the definition for *single:image{}* you can define settings that pertain to the Linux for Composer docker image (LfPHP) by defining the following:

```
{
    "name": "linuxforphp/linuxforcomposer",
    "description": "A Composer interface to run 'Linux for PHP'  … ",
    "single" : {
        "linuxforcomposer" : {
            // other settings not shown
        }
    }
}
```

The following table summarizes the *single : image : linuxforcomposer* settings:

| **single:image: linuxforcomposer** | **Description** |
|---|---|
| `php-versions : [ ]`<br><br>Example:<br>`"php-versions" : [`<br>`    "7.1",`<br>`    "7.2"`<br>`]` | This is an array of PHP versions you wish to create. If more than one version is included, a detached container is created for each one listed. If you specify "8.0", the latest alpha version of PHP 8 is compiled. For a list of versions have a look at the docker hub listings for linuxforphp-8.2-ultimate. Use the version, but leave off "-nts" or "-zts". |
| `scripts : [ ]`<br><br>Examples:<br>`"scripts" : [`<br>` "lfphp-get cms drupal app"`<br>`]`<br><br>`"scripts" : [`<br>`  "lfphp-get \`<br>`    php-frameworks \`<br>`    symfony demo"`<br>`]`<br><br>`"scripts" : [`<br>`  "lfphp"`<br>`]`<br><br>Note: the "\" is used to indicate that the option should be all on a single line. | Defines a command to execute upon container startup. Common examples include "/bin/bash" to start a command shell, or "lfphp" to launch components of a LAMP stack.<br><br>Use this script to install and launch a Content Management System (CMS) such as Drupal or WordPress (choices listed below):<br>`"lfphp-get cms` *`CMS PATH`*`"`<br><br>Use this script to install and launch a PHP framework such as Laravel or Symfony (choices listed below):<br>`"lfphp-get php-frameworks` *`FRAMEWORK PATH`*`"`<br><br>*PATH* becomes a subdirectory off ***/srv***.<br><br>Alternatively, just specify "lfphp" as the script, and you will not only start all built-in services (e.g. Apache, MySQL, PHP-FPM, etc.), but this also signals the cloud to provide a file manager and *phpMyAdmin* database access. |
| `thread-safe : <bool>`<br><br>Example:<br>`"thread-safe" : true` | Set to *true* if you wish to install an image based on one of the<br>*-zts (Zend Thread Safe) docker images. The default is *false*, which causes one of the *-nts (Non-Thread Safe) images to be used. |

The following table summarizes options for *lfphp-get* scripts

| lfphp-get cms path | lfphp-get php-frameworks path |
|---|---|
| concrete5 | symfony |
| drupal | laravel |
| joomla | cakephp |
| wordpress | slim |
| magento | lightmvc |
| prestashop | lightmvc-swoole |
| | laminas |
| | mezzio |

**NOTE**: you can also specify **zf** if you wish a legacy Zend Framework installation, or **ze** for a legacy Zend Expressive installation (otherwise use *laminas* or *mezzio*).

As you can see, the automation possibilities inherent using the LfPHP image are staggering. However, in many cases you might be modeling a customer environment not based upon LfPHP, in which case you would need to use the *dockerfile* key, discussed next.

Here is an example *linuxforcomposer.json* file that deploys WordPress:

```
{
    "name": "linuxforphp/linuxforcomposer",
    "description": "'Linux for PHP' example.",
    "single": {
        "image": {
            "linuxforcomposer": {
                "php-versions": ["7.4"],
                "script": ["lfphp-get cms wordpress demo"],
                "thread-safe": "false"
            }
        },
        "containers": {
            "modes": { "mode1": "detached" },
            "ports": {
                "port1": [ "8181:80" ]
            },
            "persist-data": {
                "mount": "false",
                "root-name": "",
                "directories": { }
            }
        }
    },
    "lfphp-cloud": {
        "account": "account_name",
        "username": "user_name",
        "token": "a1b2c3e4e5f6"
    }
}
```

## Defining a Container Based Upon a Custom Dockerfile

If you do not wish to use LfPHP you can also define settings that pertain to a custom *Dockerfile*. that draws from any image available on dockerhub:

```
{
    "name": "linuxforphp/linuxforcomposer",
    "description": "A Composer interface to run 'Linux for PHP'  … ",
    "single" : {
        "dockerfile" : {
            // other settings not shown
        }
    }
}
```

**NOTE**: the two sub-keys *linuxforcomposer* and *dockerfile* are not mutually exclusive. However, if there is a conflict between the two settings, those defined in *dockerfile* take precedence. Also, of course, the base image defined in the *Dockerfile* indicated by the `single:image:dockerfile:url` sub-key can be any image, but is not restricted to LfPHP.

The following table summarizes the *single : image : dockerfile* settings:

| single:image:dockerfile | Description |
|---|---|
| `url : <string>`<br><br>Examples:<br>`"url" : "config/Dockerfile"`<br><br>`"url" : "http://dns.com/Dockerfile"` | This identifies the location of the *Dockerfile* used to build the image. |
| `container-name : <string>`<br><br>Examples:<br>`"container-name" : "my_project"` | Defines a tag used to name the image. LfC appends a random key to the end of this string when it runs the container. |
| `username : <string>`<br><br>Examples:<br>`"username" : "info@etista.com"` | This optional key represents the username in a situation where the location of the *Dockerfile* (see *url* above) is remote and password protected. |
| `token : <string>`<br><br>Examples:<br>`"token" : "a1b2c3d4e5f6"` | This optional key represents either the password or *access* token in a situation where the location of the *Dockerfile* (see *url* above) is remote and password protected. |

Here is an example *linuxforcomposer.json* file that deploys a container using a custom *Dockerfile*:

```
{
    "name": "linuxforphp/linuxforcomposer",
    "description": "'Linux for PHP' example.",
    "single": {
        "image": {
            "dockerfile": {
                "url": "Dockerfile",
                "container-name": "custom_app",
                "username": "",
                "token": ""
            }
        },
        "containers": {
            "modes": { "mode1": "detached" },
            "ports": {
                "port1": [ "8181:80" ],
                "port2": [ "8443:443" ]
            },
            "persist-data": {
                "mount": "false",
                "root-name": "",
                "directories": { }
            }
        }
    },
    "lfphp-cloud": {
        "account": "account_name",
        "username": "user_name",
        "token": "a1b2c3e4e5f6"
    }
}
```

Now let's have a closer look at defining Docker container runtime settings.

## Defining Container Runtime Settings

The *single : image : containers* sub-key can be used to define container runtime characteristics in much the same manner as *docker run*. The following table summarizes the *single : image : containers* sub-keys:

| single:image:containers | Description |
|---|---|
| `modes : [key:<string>, etc.]` Examples: `"modes" : [ "mode1":"detached", "mode2":"tty" ]` | This sub-key represents one or more operating modes for the container. There are three possibilities: *detached*, *interactive* and *tty*. *docker run* options, in order: **-d**, **-t** and **-i** |
| `ports:{ key : [<LOCAL:DOCKER>,etc.] }` Examples: `"ports" : { "port1": ["8181:80"], "port2": ["8443:443"] }` | Defines sets of port mappings between the local computer (*LOCAL*) and the Docker container (*DOCKER*). You can define as many sets as are appropriate. If you are deploying multiple containers, each additional entry for each set is then assigned to each successive container. (see discussion below). *docker run* option: **-p** |
| `volumes : { key : <LOCAL:DOCKER>, etc. }` Examples: `"volumes" : { "volume1": "/home/ned/repo":"/repo" }` | This optional key represents a mapping between a local directory path and a path inside the container. Volumes mapped in this manner are shared between containers. *docker run* option: **-v** |
| `persist-data : { mount : <bool>, root-name : <string>, directories : { key : <PATH>, etc. } }` Examples: `"persist-data" : { "mount" : true, "root-name" : "my_project_vol", "directories" : { "directory1" : "/srv/data" } }` | Use this option to have Docker create a volume that is persistent. The following sub-keys must be defined: `mount` : set *true* to have Docker mount the volume (and therefore have these settings take effect). `root-name` : unique Docker volume name. It's recommended to use the project name as a prefix. `directories` : assign one or more key/value pairs, where the key is "*directory1*", "*directory2*", etc., and the value is a string representing a path internal to the Docker container once running. *docker volume create* |

## Port Mapping Using Linux for Composer

The *single : image : containers : ports* sub-key is an array that maps ports between the local computer and the Docker container. Each additional array element represents a different port. In this example, there a single image defined by a *Dockerfile*. Local port 8181 is mapped to container port 80, and local port 8443 is mapped to container port 443:

```
{
    "single": {
        "image": {
            "dockerfile": {
                "url": "Dockerfile",
                "container-name": "phpcl_jumpstart_lfphp"
            }
        },
        "containers": {
            "modes": {
                "mode1": "detached"
            },
            "ports": {
                "port1": ["8181:80"],
                "port2": ["8443:443"]
            },
            "persist-data": {
                "mount": "false"
            }
        }
    }
}
```

When you plan to deploy multiple containers, it starts to get confusing. Each sub-sub-key is itself an array. Each sub-sub-sub array maps to each of the multiple containers. Multiple containers are launched, for example, when you specify more than one version of PHP.

In this example, two containers are launched: one based upon PHP 5.6, the other upon PHP 7.4:

```
{
    "single": {
        "image": {
            "dockerfile": {
                "url": "Dockerfile",
                "container-name": "phpcl_jumpstart_lfphp"
            },
            "linuxforcomposer": {
                "php-versions": ["5.6","7.4"],
                "script": [ "lfphp" ],
                "thread-safe": "false"
            }
        },
        "containers": {
            "modes": {
                "mode1": "detached"
            },
```

When we get to port mappings, the following example has local ports 8181 and 8443 mapped to the PHP 5.6 container ports 80 and 443 respectively.  For the PHP 7.4 container, on the other hand, the local host ports 8282 map to the 7.4 container's port 80, and port 8543 maps to the 7.4 container's port 443:

```
            "ports": {
                "port1": ["8181:80", "8282:80"],
                "port2": ["8443:443", "8543:443"]
            },
            "persist-data": {
                "mount": "false"
            }
        }
    }
}
```

Next we have a quick look at configuring LfC to use docker-compose.

## Defining a Container Using Docker-Compose

This option is useful if you are using Docker Compose, and plan to deploy multiple containers. An example would be something like the following trio of "servers":

- Main Application Server
- Database Server
- Mail Server

The main work is not done by Linux for Composer, but rather in the docker-compose.yml file. The coverage of the latter is beyond the scope of this guide. You can find more details here:

https://docs.docker.com/compose/compose-file/

The following table gives a summary of the settings for the top-level directive *docker-compose*:

| docker-compose | Description |
|---|---|
| `url : <string>`<br><br>Examples:<br>`"url" : "/path/to/project"`<br><br>`"url" : "http://dns.com/project"` | This identifies the location of the *docker-compose.yml* file used to build the image. The string can represent either a local directory path to a repository containing the file, or a remote location accessible via HTTP or HTTPS. |
| `username : <string>`<br><br>Examples:<br>`"username" : "info@etista.com"` | This optional key represents the username in a situation where the location of the *Dockerfile* (see *url* above) is remote and password protected. |
| `token : <string>`<br><br>Examples:<br>`"token" : "a1b2c3d4e5f6"` | This optional key represents either the password or *access* token in a situation where the location of the *docker-compose.yml* (see *url* above) is remote and password protected. |

## Defining a Container Using LfPHP Cloud

The last top-level key in the *linuxforcomposer.json* file is *lfphp-cloud*, which allows you to define parameters in order to deploy your application to LfPHP Cloud Services, or any other cloud service allowing report deployment access. The following table gives a summary of the settings for the top-level directive *lfphp-cloud*:

| lfphp-cloud | Description |
|---|---|
| `account : <string>`<br><br>Examples:<br>`"account" : "infoetistacom22"` | This identifies the cloud service account associated with the access token (see below). |
| `username : <string>`<br><br>Examples:<br>`"username" : "info@etista.com"` | This optional key represents the login name for the remote cloud service. |
| `token : <string>`<br><br>Examples:<br>`"token" : "a1b2c3d4e5f6"` | This key represents either the password or *access* token for the remote cloud service. |

Now that you have an idea what goes into the *linuxforcomposer.json* file, it's time for an example.

# LfPHP Cloud Services Deployment Example

In this example, we have a demonstration app based on *Mezzio*, a micro-framework from the *Laminas* project. The name of the project is *IpWhat*. It can get information on an IP address by making a call to an external web service *Ip2C*. It also has an internal database, and can present a list of names sorted alphabetically.

The first thing to do is to define the *linuxforcomposer.json* file.

## Sample Linux for Composer JSON File

The file starts with a name and description:

```
{
    "name": "phpcl_jumpstart_lfphp",
    "description": "Demonstrates deployment using a Dockerfile",
```

Next we define the *single:image* block:

```
    "single": {
        "image": {
            "dockerfile": {
                "url": "Dockerfile",
                "container-name": "phpcl_jumpstart_lfphp",
                "username": "",
                "token": ""
            },
```

We also add a *linuxforcomposer* block mainly for added documentation:

```
            "linuxforcomposer": {
                  "php-versions": ["7.4"],
                "script": ["lfphp --mysql --phpfpm --apache"],
                "thread-safe": "false"
            }
        },
```

Since we'll be deploying to the cloud service, there's no need for *tty* nor *interactive* modes:

```
        "containers": {
            "modes": {
                "mode1": "detached",
            },
```

We define port mappings for *80* and *443*:

```
            "ports": {
                "port1": ["8181:80"],
                "port2": ["8443:443"]
            },
```

And, in this example, no permanent volumes will be mounted:

```
            "persist-data": {
                "mount": "false",
                "root-name": "",
                "directories": {
                    "directory1": "",
                    "directory2": "",
                    "directory3": ""
                }
            }
        }
    },
```

Finally, we define the cloud settings:

```
    "lfphp-cloud": {
        "account": "infoetistacom3",
        "username": "info@etista.com",
        "token": "a1b2c3d4e5f6a1b2c3d4e5f6 "
    }
}
```

Here is the complete *linuxforcomposer.json* file used in this example:

```
{
    "name": "phpcl_jumpstart_lfphp",
    "description": "Demonstrates deployment using a Dockerfile",
    "single": {
        "image": {
            "dockerfile": {
                "url": "Dockerfile",
                "container-name": "phpcl_jumpstart_lfphp",
                "username": "",
                "token": ""
            },
            "linuxforcomposer": {
                  "php-versions": ["7.4"],
                "script": ["lfphp --mysql --phpfpm --apache"],
                "thread-safe": "false"
            }
        },
        "containers": {
            "modes": {
                "mode1": "detached"
            },
            "ports": {
                "port1": ["8181:80"],
                "port2": ["8443:443"]
            },
            "persist-data": {
                "mount": "false",
                "root-name": "",
                "directories": {
                    "directory1": "",
                    "directory2": "",
                    "directory3": ""
                }
            }
        }
    },
    "lfphp-cloud": {
        "account": "infoetistacom3",
        "username": "info@etista.com",
        "token": "a1b2c3d4e5f6a1b2c3d4e5f6 "
    }
}
```

Now we need to define the *Dockerfile*.

## Example Dockerfile

Although advanced Linux for Composer usage allows for a *docker-compose* configuration, for the purposes of this guide we follow the single-service-single-container paradigm. Accordingly, all you need to do is to create a normal *Dockerfile* and confirm that your application runs in its container.

The tricky part is that due to the way the container gets created on the cloud service, you will be unable to *copy* any files into the container as it’s being built. Accordingly, make sure that any external assets you need can be obtained via an Internet download.

If you just need a single file that is accessible via HTTP, in your *Dockerfile*, simply add a directive along these lines:

```
RUN wget --output-file=FILE http://dns.name.com/path/to/remote/file
```

In any event, be sure to place all your initialization scripts in a *git* repository accessible over the Internet, along with your application code. As an example, in the *Dockerfile*, assuming your code is on github.com, add the following:

```
RUN git clone https://github.com/your/repository
```

To preserve any data, one technique is to store a skeleton structure in your git repo, and have your init script create the database, grant permissions, and restore the table structure. You can then use phpMyAdmin, provided by the cloud service, to restore any data you have backed up. (Er … you *did* back up your data, didn’t you?)

In our sample *Dockerfile* we first indicate the source of the original image:

```
FROM asclinux/linuxforphp-8.2-ultimate:7.4-nts
```

Next, we clone from the repository:

```
RUN git clone https://github.com/path/to/repo /target/dir/
```

Note that if your repository is private, generate a security access token, and use the following syntax:

```
RUN git clone https://user:token@github.com/path/to/repo /target/dir
```

If you've updated your source code, don't forget to inform the Docker daemon there's been a change … otherwise it will continue to pull from the original branch!

```
RUN git clone https://github.com/path/to/repo --branch NEW /target/dir/
```

You might then wish to run an initialization script (contained in your newly restored repo):

```
RUN chmod +x /srv/jumpstart/init.sh
RUN /srv/jumpstart/init.sh
```

After that you will probably need to assign the web server document root to the correct directory path.  In this example, the sample application is built using the *Mezzio* framework (formerly Zend Expressive), so the document  root needs to be the *public* folder off the project root of *ip_demo*:

```
RUN \
  cd /srv && \
  mv -f -v /srv/www /srv/www.OLD && \
  ln -s -f -v /srv/jumpstart/ip_demo/public /srv/www
RUN chown apache:apache /srv/www
RUN chown -R apache:apache /srv/jumpstart
RUN chmod -R 775 /srv/jumpstart
```

Finally, depending on the original source image chosen, you can designate an entry point and followup command.  As our source image is Linux for PHP (LfPHP), the entry point is the *lfphp* command, and our commands are flags that activate MySQL, PHP FastCGI processing module, and Apache:

```
ENTRYPOINT ["/bin/lfphp"]
CMD ["--mysql", "--phpfpm", "--apache"]
```

Here is the complete *Dockerfile*:

```
FROM asclinux/linuxforphp-8.2-ultimate:7.4-nts
MAINTAINER doug.bierer@etista.com
RUN git clone https://github.com/phpcl/phpcl_jumpstart_lfphp /srv/jumpstart
RUN chmod +x /srv/jumpstart/init.sh
RUN /srv/jumpstart/init.sh
RUN \
  cd /srv && \
  mv -f -v /srv/www /srv/www.OLD && \
  ln -s -f -v /srv/jumpstart/ip_demo/public /srv/www
RUN chown apache:apache /srv/www
RUN chown -R apache:apache /srv/jumpstart
RUN chmod -R 775 /srv/jumpstart
ENTRYPOINT ["/bin/lfphp"]
CMD ["--mysql", "--phpfpm", "--apache"]
```

Next we have a look at the init script.

## You Mentioned an "Init" Script?

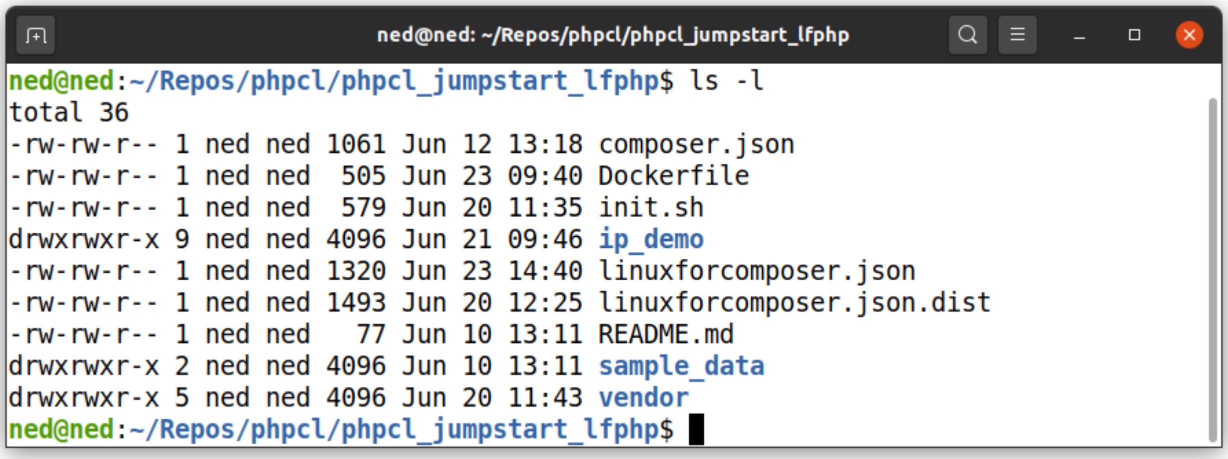
The init script is a simple *BASH* (or other shell) script that can perform any desired initialization. In our example, there are two main tasks to be performed: use *Composer* to install third party software into our demo app, and to initialize the database. We also add the *ServerName* to the Apache *httpd.conf* file. Here is the complete script:

```
#!/bin/bash
echo "ServerName jumpstart" >> /etc/httpd/httpd.conf
cd /srv/jumpstart/ip_demo
php composer.phar self-update
rm -f /srv/jumpstart/ip_demo/composer.lock
php composer.phar install
echo "Restoring database ..."
/etc/init.d/mysql start
sleep 5
mysql -uroot -v -e "CREATE DATABASE jumpstart;"
mysql -uroot -v -e "CREATE USER 'jumpstart'@'localhost' \
    IDENTIFIED BY 'password';"
mysql -uroot -v -e "GRANT ALL PRIVILEGES ON *.* TO 'jumpstart'@'localhost';"
mysql -uroot -v -e "FLUSH PRIVILEGES;"
mysql -uroot -v -e "SOURCE /srv/jumpstart/sample_data/jumpstart.sql;" jumpstart
```

The next step is to test everything locally.

## Testing the Deployment Locally

In order to test the deployment locally, we must have *Docker* installed. In addition, of course, both Composer and Linux for Composer must be installed. Here is the base project structure listing:

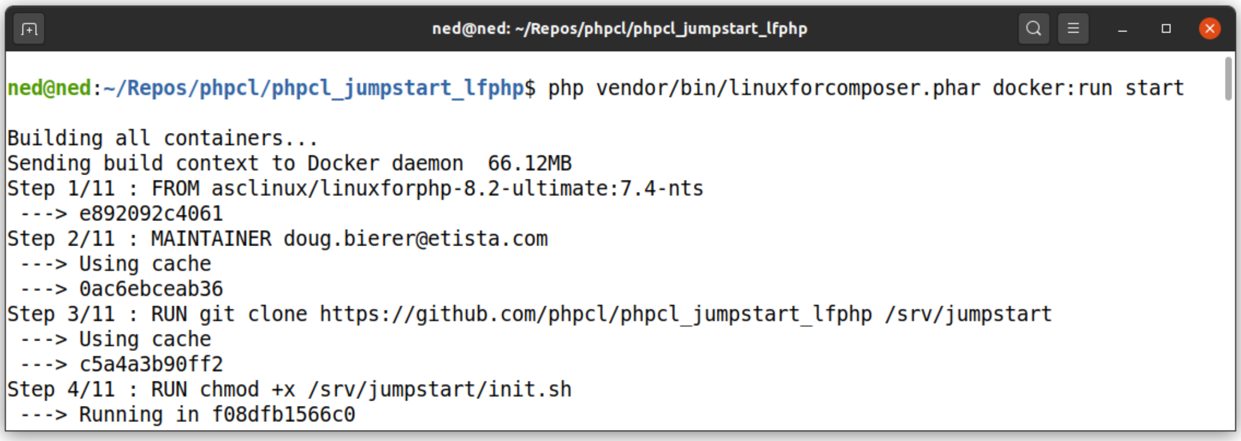


```
ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$ ls -l
total 36
-rw-rw-r-- 1 ned ned 1061 Jun 12 13:18 composer.json
-rw-rw-r-- 1 ned ned  505 Jun 23 09:40 Dockerfile
-rw-rw-r-- 1 ned ned  579 Jun 20 11:35 init.sh
drwxrwxr-x 9 ned ned 4096 Jun 21 09:46 ip_demo
-rw-rw-r-- 1 ned ned 1320 Jun 23 14:40 linuxforcomposer.json
-rw-rw-r-- 1 ned ned 1493 Jun 20 12:25 linuxforcomposer.json.dist
-rw-rw-r-- 1 ned ned   77 Jun 10 13:11 README.md
drwxrwxr-x 2 ned ned 4096 Jun 10 13:11 sample_data
drwxrwxr-x 5 ned ned 4096 Jun 20 11:43 vendor
ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$
```

Issue this command, from the root directory of your project where the *linuxforcomposer.json* file resides, to have Linux for Composer start the container running locally:

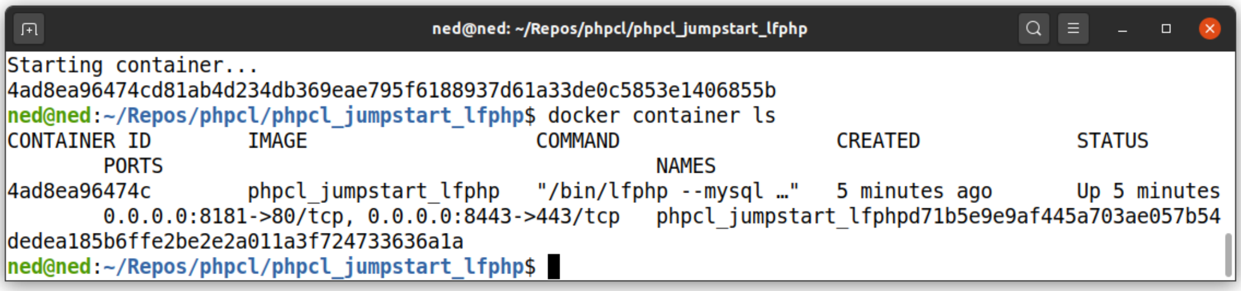
```
php vendor/bin/linuxforcomposer.phar docker:run start
```

This launches *docker build* using the *Dockerfile* specified under the *dockerfile:url* key as shown here:



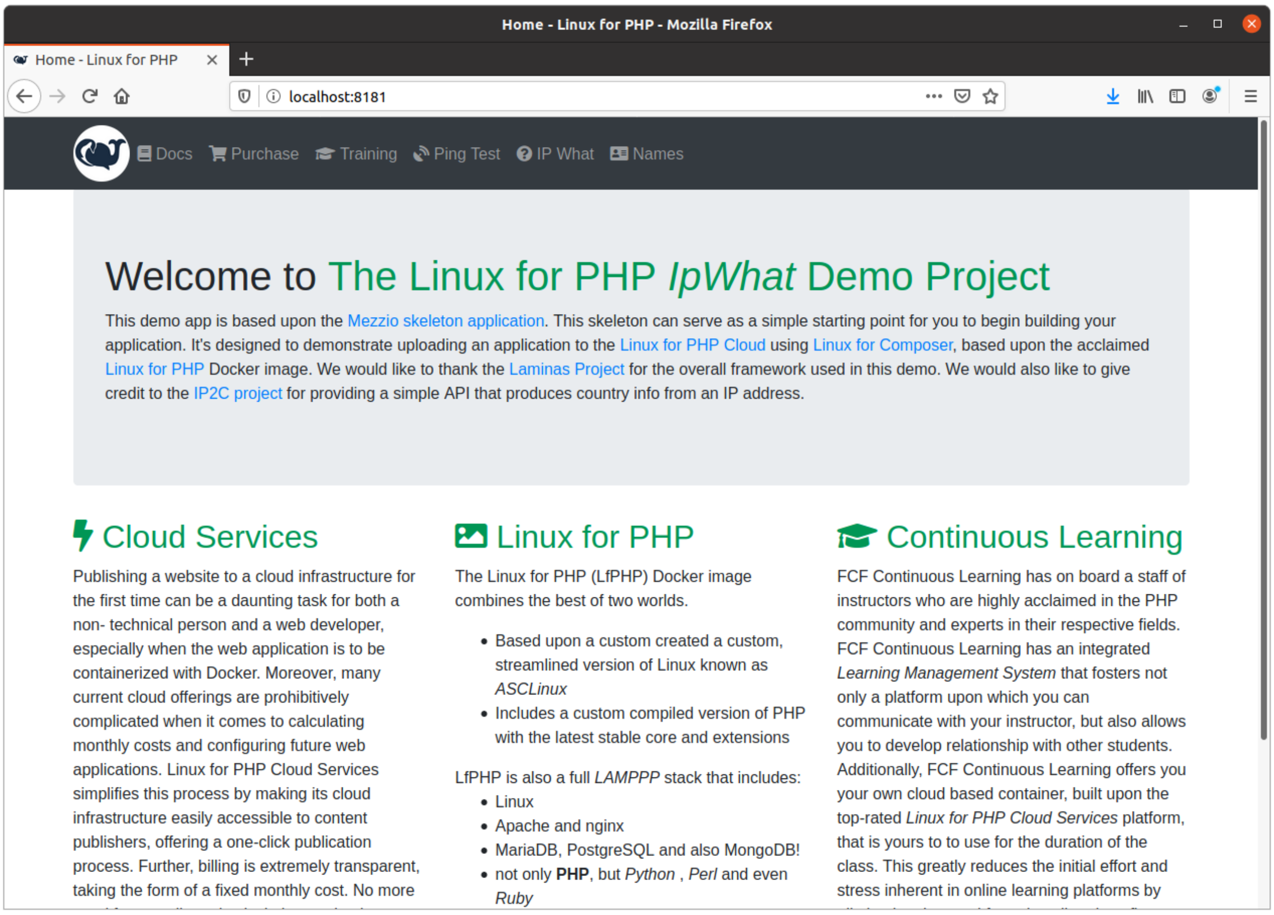
```
ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$ php vendor/bin/linuxforcomposer.phar docker:run start

Building all containers...
Sending build context to Docker daemon  66.12MB
Step 1/11 : FROM asclinux/linuxforphp-8.2-ultimate:7.4-nts
 ---> e892092c4061
Step 2/11 : MAINTAINER doug.bierer@etista.com
 ---> Using cache
 ---> 0ac6ebceab36
Step 3/11 : RUN git clone https://github.com/phpcl/phpcl_jumpstart_lfphp /srv/jumpstart
 ---> Using cache
 ---> c5a4a3b90ff2
Step 4/11 : RUN chmod +x /srv/jumpstart/init.sh
 ---> Running in f08dfb1566c0
```

Once finished, as we specified *detached* mode, the container, if built successfully, should now be running:



```
Starting container...
4ad8ea96474cd81ab4d234db369eae795f6188937d61a33de0c5853e1406855b
ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$ docker container ls
CONTAINER ID        IMAGE                    COMMAND                  CREATED             STATUS
        PORTS                                          NAMES
4ad8ea96474c        phpcl_jumpstart_lfphp    "/bin/lfphp --mysql …"   5 minutes ago       Up 5 minutes
        0.0.0.0:8181->80/tcp, 0.0.0.0:8443->443/tcp   phpcl_jumpstart_lfphpd71b5e9e9af445a703ae057b54
dedea185b6ffe2be2e2a011a3f724733636a1a
ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$ 
```

If we now open a browser to [http://localhost:8181/](http://localhost:8181/) we should see the demo application running:



Welcome to The Linux for PHP *IpWhat* Demo Project

This demo app is based upon the Mezzio skeleton application. This skeleton can serve as a simple starting point for you to begin building your application. It's designed to demonstrate uploading an application to the Linux for PHP Cloud using Linux for Composer, based upon the acclaimed Linux for PHP Docker image. We would like to thank the Laminas Project for the overall framework used in this demo. We would also like to give credit to the IP2C project for providing a simple API that produces country info from an IP address.

Cloud Services

Publishing a website to a cloud infrastructure for the first time can be a daunting task for both a non- technical person and a web developer, especially when the web application is to be containerized with Docker. Moreover, many current cloud offerings are prohibitively complicated when it comes to calculating monthly costs and configuring future web applications. Linux for PHP Cloud Services simplifies this process by making its cloud infrastructure easily accessible to content publishers, offering a one-click publication process. Further, billing is extremely transparent, taking the form of a fixed monthly cost. No more

Linux for PHP

The Linux for PHP (LfPHP) Docker image combines the best of two worlds.

- Based upon a custom created a custom, streamlined version of Linux known as *ASCLinux*
- Includes a custom compiled version of PHP with the latest stable core and extensions

LfPHP is also a full *LAMPPP* stack that includes:

- Linux
- Apache and nginx
- MariaDB, PostgreSQL and also MongoDB!
- not only **PHP**, but *Python* , *Perl* and even *Ruby*

Continuous Learning

FCF Continuous Learning has on board a staff of instructors who are highly acclaimed in the PHP community and experts in their respective fields. FCF Continuous Learning has an integrated *Learning Management System* that fosters not only a platform upon which you can communicate with your instructor, but also allows you to develop relationship with other students. Additionally, FCF Continuous Learning offers you your own cloud based container, built upon the top-rated *Linux for PHP Cloud Services* platform, that is yours to to use for the duration of the class. This greatly reduces the initial effort and stress inherent in online learning platforms by

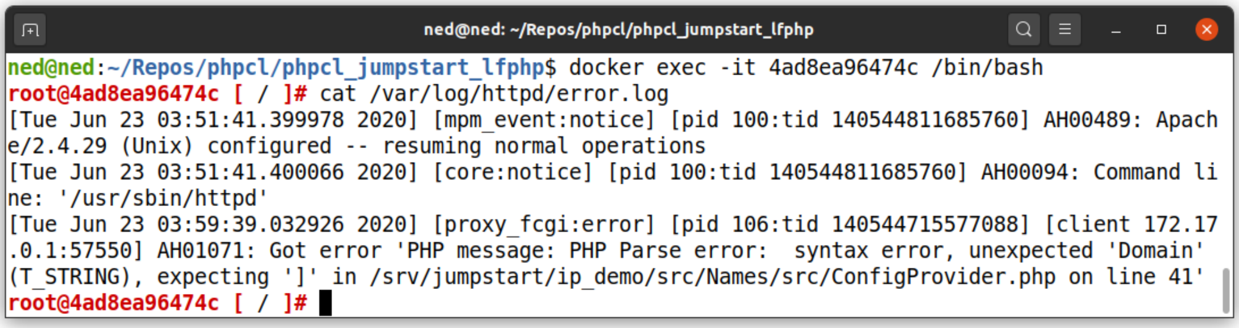
If you get a blank screen, or otherwise you note that all is not well, it's time to perform some troubleshooting.

## Troubleshooting Your Deployment

If you get a blank screen, you can shell into the running Docker container by making a note of its ID or tag, and issuing the following command:

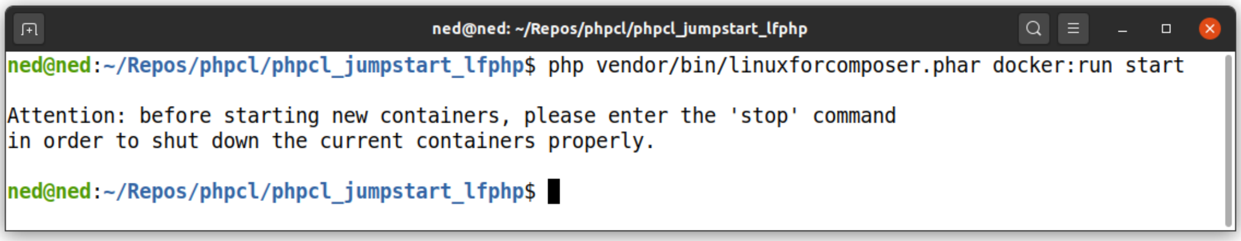
```
docker exec -it <CONTAINER ID OR TAG> /bin/bash
```

You can then consult the appropriate error log to determine the nature of the error:



```
ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$ docker exec -it 4ad8ea96474c /bin/bash
root@4ad8ea96474c [ / ]# cat /var/log/httpd/error.log
[Tue Jun 23 03:51:41.399978 2020] [mpm_event:notice] [pid 100:tid 140544811685760] AH00489: Apach
e/2.4.29 (Unix) configured -- resuming normal operations
[Tue Jun 23 03:51:41.400066 2020] [core:notice] [pid 100:tid 140544811685760] AH00094: Command li
ne: '/usr/sbin/httpd'
[Tue Jun 23 03:59:39.032926 2020] [proxy_fcgi:error] [pid 106:tid 140544715577088] [client 172.17
.0.1:57550] AH01071: Got error 'PHP message: PHP Parse error:  syntax error, unexpected 'Domain'
(T_STRING), expecting ']' in /srv/jumpstart/ip_demo/src/Names/src/ConfigProvider.php on line 41'
root@4ad8ea96474c [ / ]#
```

If you encounter this error: *Attention: before starting new containers* … it's most likely that a



```
ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$ php vendor/bin/linuxforcomposer.phar docker:run start

Attention: before starting new containers, please enter the 'stop' command
in order to shut down the current containers properly.

ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$
```

*linuxforcomposer.pid* file exists in the vendor/composer directory.  To get rid of this error, stop the running container by moving the project directory and issuing this command:

```
php vendor/bin/linuxforcomposer.phar docker:run stop
```

Or, if the container has already been stopped (e.g. using *docker container stop*), simply delete the */path/to/project/vendor/composer/linuxforcomposer.pid* file.

**NOTE**:
Once you have finished troubleshooting, don't forget to push your changes to the repository!

One final point is that you might find yourself in the situation where no matter how many times you fix a bug in the application code, you *still keep getting the same error* or blank screen.  This is extremely frustrating and can waste hours and hours of your time.

In this situation, the most likely culprit is one form or another of *caching*.  Here are some suggested remedies:

- Make sure you have pushed all changes to the git repo.  If your *Dockerfile* clones from a git repo, and you haven't pushed changes, you're going to keep getting the same buggy code.
- When testing locally, Docker will create a separate layer for each *RUN* directive in the *Dockerfile*.  One of these will most likely be the layer that clones from the git repo.  In this case, no matter how many times you push changes, they never appear when the image is built as Docker logically draws from its own cache.  Accordingly you'll need to get rid of the image which forces Docker to rebuild everything.
- Finally: you application code itself might have caching enabled, and you might accidentally be including this when you push changes to the repo.  Create a *.gitignore* file that excludes any application caching directories.

**NOTE**:
When you use Linux for Composer to stop a running container, you are asked if you wish to commit the change to your local Docker repo.  If you choose *yes*, the next time you use Linux for Composer to run the container, it draws from cache, saving time.

Also, very importantly, your repository source code has most likely changed.  Docker is unaware of the change, however, unless you change the *git clone* command in your Dockerfile.  Accordingly, it's not a bad idea to either (A) use *git tag*, or (B) create a new branch, to record your changes.  When you've got a new version of the code stored in the repo as either a new branch, or a new tag, you can modify the *git clone* command in your Dockerfile by adding the `--branch <NEW_BRANCH>` or `--branch <TAG>` options.  When Linux for Composer uploads your revised Dockerfile, the Docker daemon on the cloud service will recognize the change, and pull a fresh copy from the repo instead of using its Docker cache.

Now it's time to deploy to the cloud service!

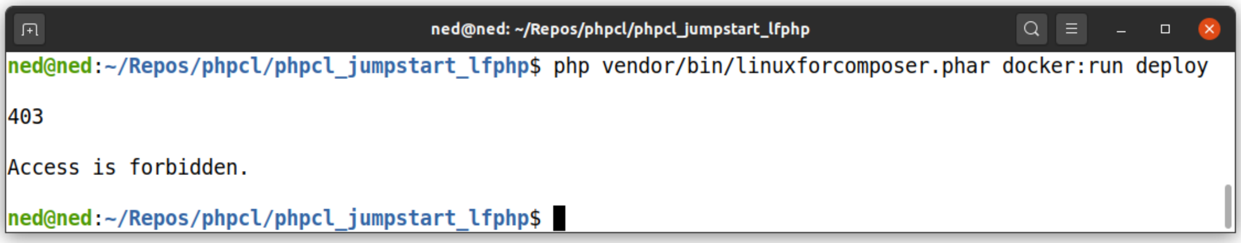
## Deploying to Linux for PHP Cloud Services

Once you have the project running successfully on your local computer, it's time to deploy to the cloud service. Double check your IP address and if it's changed for some reason, generate a new access token, as described above.

To deploy to the cloud service, from the root directory of your project where the *linuxforcomposer.json* file resides, issue this command:

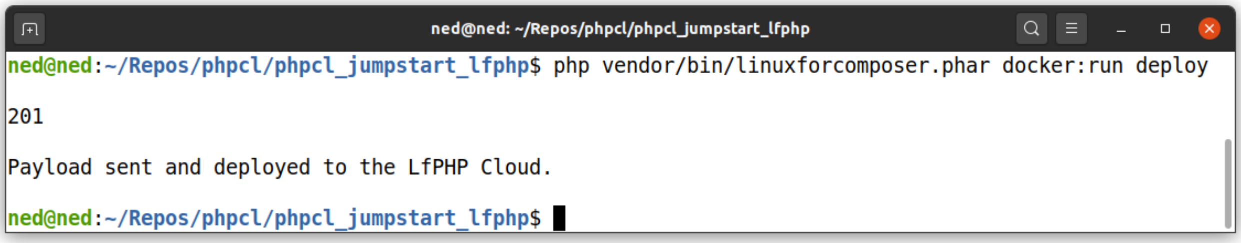
```
php vendor/bin/linuxforcomposer.phar docker:run deploy
```

If you see this message, double check to make sure your account name, username and token are correctly defined. Also, make sure that your current, valid IP address is at the top of the list:



```
ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$ php vendor/bin/linuxforcomposer.phar docker:run deploy

403

Access is forbidden.

ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$ 
```

Otherwise, you should get a success message as shown here:

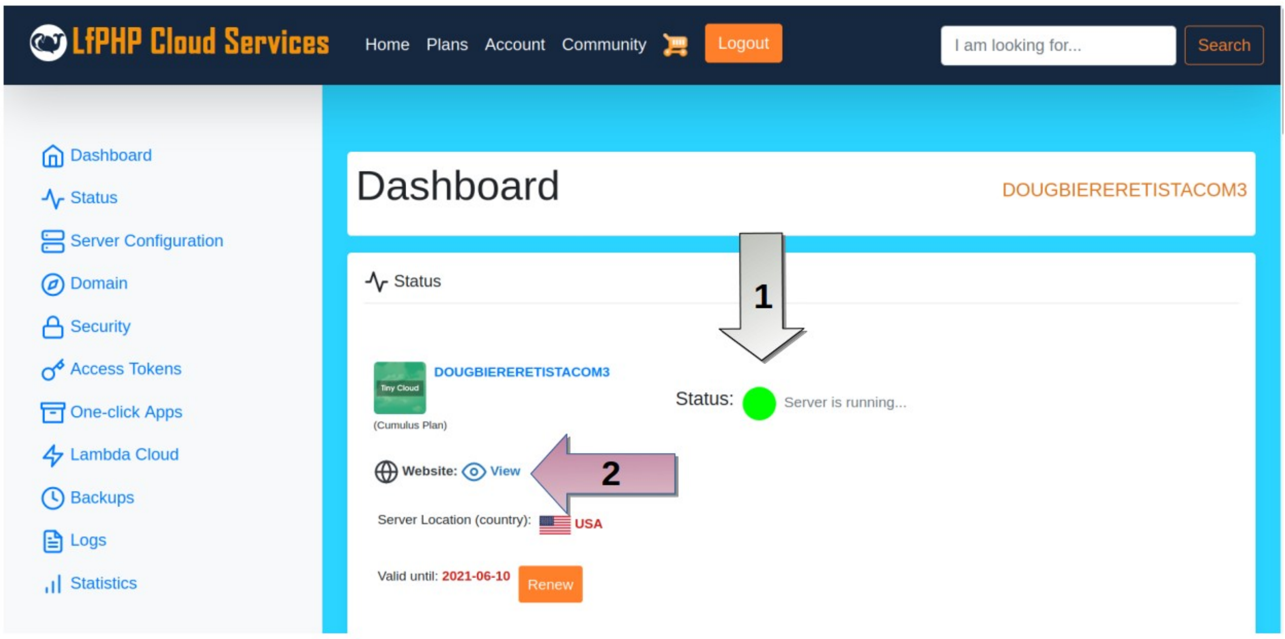


```
ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$ php vendor/bin/linuxforcomposer.phar docker:run deploy

201

Payload sent and deployed to the LfPHP Cloud.

ned@ned:~/Repos/phpcl/phpcl_jumpstart_lfphp$ 
```

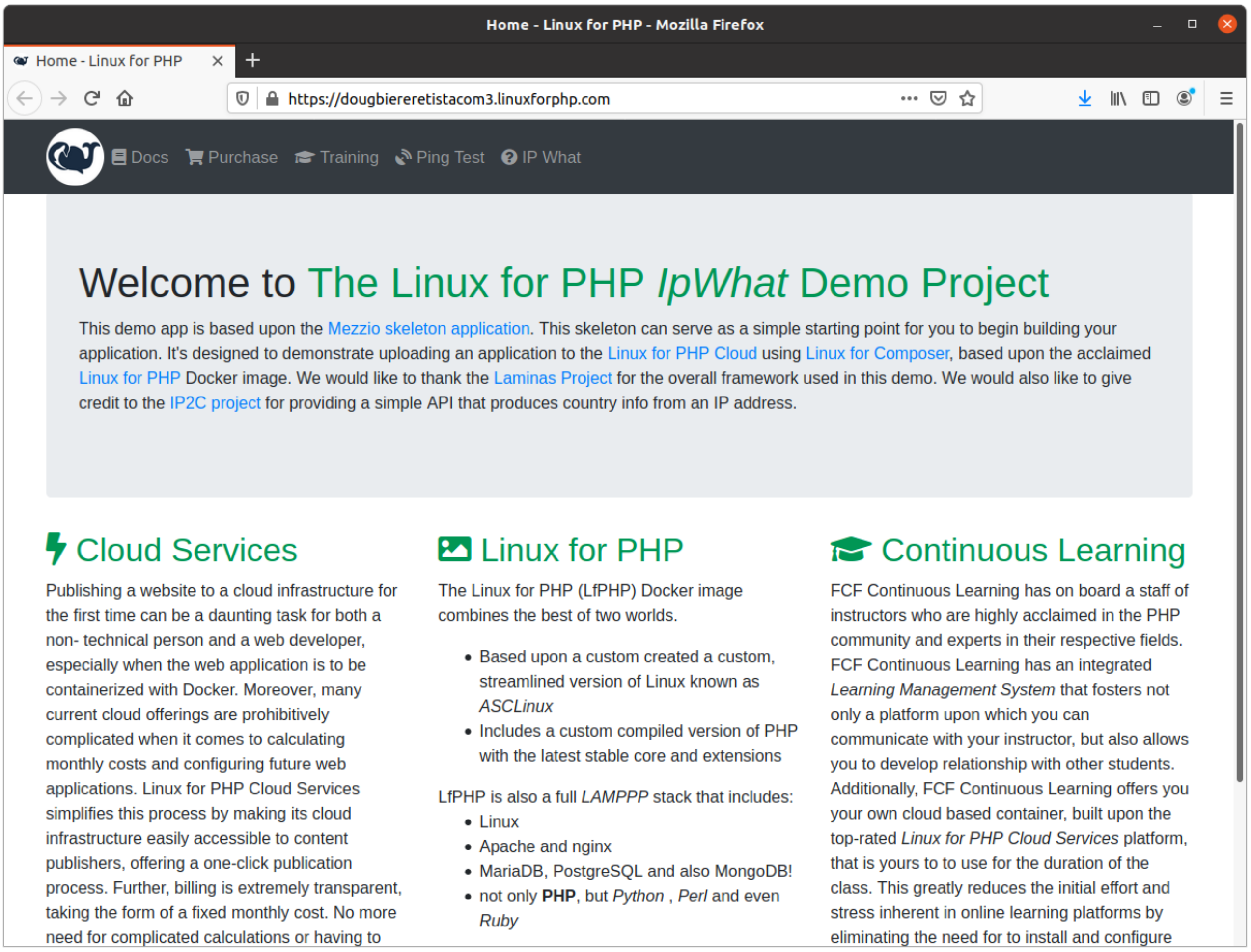
You will then need to wait until your service has been deployed. The time it takes to run locally should give you an indication how long this will take. In the meantime, monitor your dashboard from your cloud services account.

NOTE: although an initial byte count shows up, possibly in the gigabyte range, this represents what goes to the Docker daemon: **not** what is uploaded. The only thing uploaded is the *Dockerfile*.

During deployment the server status button [1] will be either yellow, indicating status unknown, or red, indicating server stopped:



Click on *Website: View* [2] to have a look at your newly deployed application:


Home - Linux for PHP - Mozilla Firefox
Home - Linux for PHP
https://dougbiereretistacom3.linuxforphp.com
Docs
Purchase
Training
Ping Test
IP What

Welcome to The Linux for PHP *IpWhat* Demo Project

This demo app is based upon the Mezzio skeleton application. This skeleton can serve as a simple starting point for you to begin building your application. It's designed to demonstrate uploading an application to the Linux for PHP Cloud using Linux for Composer, based upon the acclaimed Linux for PHP Docker image. We would like to thank the Laminas Project for the overall framework used in this demo. We would also like to give credit to the IP2C project for providing a simple API that produces country info from an IP address.

Cloud Services

Publishing a website to a cloud infrastructure for the first time can be a daunting task for both a non- technical person and a web developer, especially when the web application is to be containerized with Docker. Moreover, many current cloud offerings are prohibitively complicated when it comes to calculating monthly costs and configuring future web applications. Linux for PHP Cloud Services simplifies this process by making its cloud infrastructure easily accessible to content publishers, offering a one-click publication process. Further, billing is extremely transparent, taking the form of a fixed monthly cost. No more need for complicated calculations or having to

Linux for PHP

The Linux for PHP (LfPHP) Docker image combines the best of two worlds.

- Based upon a custom created a custom, streamlined version of Linux known as *ASCLinux*
- Includes a custom compiled version of PHP with the latest stable core and extensions

LfPHP is also a full *LAMPPP* stack that includes:

- Linux
- Apache and nginx
- MariaDB, PostgreSQL and also MongoDB!
- not only **PHP**, but *Python* , *Perl* and even *Ruby*

Continuous Learning

FCF Continuous Learning has on board a staff of instructors who are highly acclaimed in the PHP community and experts in their respective fields. FCF Continuous Learning has an integrated *Learning Management System* that fosters not only a platform upon which you can communicate with your instructor, but also allows you to develop relationship with other students. Additionally, FCF Continuous Learning offers you your own cloud based container, built upon the top-rated *Linux for PHP Cloud Services* platform, that is yours to to use for the duration of the class. This greatly reduces the initial effort and stress inherent in online learning platforms by eliminating the need for to install and configure

## Summary

In this guide you learned how to deploy an application using Linux for Composer.  You learned about the major configuration settings in the *linuxforcomposer.json* file, including how to define a single image using either the LfPHP Docker image, or a custom image built from a *Dockerfile*.  In addition you learned settings that correspond to various *docker run* directives include port and volume mappings.

You then went through a sample deployment using a custom *Dockerfile* deploying directly to Linux for PHP Cloud Services.  There was also a discussion on troubleshooting and various error conditions that might arise.

As the saying goes … *that's all folks*.  I hope you enjoy using this technology as much as do I.  Happy coding and even happier deployment!