Idephix Documentation

Release 0.3.0

Manuel Baldassarri

January 31, 2017

Contents

CHAPTER 1

Installing

You can install Idephix in several ways:

1.1 As a phar (Recommended)

You can download the phar directly from getidephix.com

```
$ curl -LSs http://getidephix.com/idephix.phar > idephix.phar
$ chmod a+x idephix.phar
```

We recommend you to download the phar and put it under version control with your project, so you can have the best control over used version and you'll be sure to avoid dependencies conflicts with your project.

1.2 As a composer dependency

```
$ composer require ideato/idephix --dev
```

1.3 Globally using homebrew

```
$ brew tap ideatosrl/php
$ brew install idephix
```

CHAPTER 2

Basic Usage

Idephix is a tool for running tasks. As a developer your main focus will be on writing tasks (as php functions) inside a file called `idxfile.php`. You will also need to specify some configurations inside a file called `idxrc.php`.

Fortunately you won't need to create those files manually, Idephix can generate them for you.

```
$ idx initFile
```

This will generate an `idxfile.php` and a `idxrc.php` file that you can use as a boiler plate for your automated tasks.

Basically Idephix is a tool for running tasks either remote or local. Remote tasks can be run against a chosen environment connecting to it through ssh (see *Configuration* for more information on ssh connection and environments).

Local tasks are run on the local host without any need to establish an ssh connection.

CHAPTER 3

Configuration

All Idephix configurations are defined within the `idxrc.php` file. By default Idephix will look for a file named `idxrc.php` in the root directory of your project, but you can store it wherever you want and name it whatever you want. If you want to use a custom configuration file you need to specify id by using `-c` option with Idephix CLI.

The file **must** return an array of configurations. Idephix uses 3 main configuration elements:

- environments
- ssh_client
- extensions

None of them are mandatory, you'll need environments (at least one) and ssh_client only to execute remote tasks and extensions only if you want to register some extension.

This example of `idxrc.php` file will give you and idea of how define environments, ssh clients and extensions:

```
<?php

$environments = array(
    'prod' => array(
        'hosts' => array('127.0.0.1', '33.33.33.10'),
        'ssh_params' => array(
            'user' => 'ideato'
        ),
    ),
    'stage' => array(
        'hosts' => array('192.168.169.170'),
        'ssh_params' => array(
            'user' => 'ideato'
        ),
    ),
    'test' => array(
        'hosts' => array('127.0.0.1'),
        'ssh_params' => array('user' => 'kea'),
    ),
);

return array(
    'envs' => $environments,
    'ssh_client' => new \Idephix\SSH\SshClient(),
    'extensions' => array(),
);
```

Idephix use ssh-agent to authenticate to remote computers without password. Otherwise you can specify the password in your script or use `CLISshProxy` (instead of the default `PeclSsh2Proxy`) that ask you the password.

Once you have defined several environments you can specify which one you want to run your remote task against, using `--env` CLI option.

CHAPTER 4

Defining Tasks

To define a new task you just need to define a function within the `idxfile.php` and it will be automatically mounted as an Idephix command.

```php
<?php

function myNewTask()
{
    echo 'I am a brand new task' . PHP_EOL;
}
```

Now running idx you'll get

```
$ bin/idx

$ Available commands:
$ help            Displays help for a command
$ initFile        Init idx configurations and tasks file
$ list            Lists commands
$ myNewTask
```

And you can execute it with:

```
$ bin/idx myNewTask
I am a brand new task
```

You can even execute a task within another task:

```php
<?php

function anotherTask()
{
}

function myNewTask(\Idephix\Context $context)
{
    $context->anotherTask();
}
```

Hint: Every task can define a special arguments: `$context`. If you define an argument and type hint it as `\Idephix\Context` an instance of the context object will be injected at runtime. The context object allows you to execute tasks and access configuration parameters. For more info about `Context` check out *Scripting with Idephix* section

4.1 Adding task arguments

Function parameters will be used as the task arguments.

```php
<?php

function yell($what)
{
    echo $what . PHP_EOL;
}
```

4.1.1 Mandatory Arguments

The parameter $name will be a mandatory option to be specified in the command execution.

```
$ bin/idx help yell
Usage:
    yell what

Arguments:
    what
```

You can add as many arguments as you need, just adding function parameters.

4.1.2 Optional Arguments

If you want to add optional arguments, just define a default value for the parameter, as:

```php
<?php

function yell($what = 'foo')
{
    echo $what . PHP_EOL;
}
```

4.1.3 Optional arguments as task flags

A flag is a special parameter with default value false. Using flags should be useful to implement a dry-run approach in your script

```php
   <?php

   function deploy($go = false){
        if ($go) {
            //bla bla bla
        return;
    }
}
```

4.2 Documenting tasks

Tasks and arguments can have a description. You can define descriptions using simple and well known phpdoc block.

```php
<?php

/**
 * This command will yell at you
 *
 *
 * @param string $what What you want to yell
 */
function yell($what = 'foo')
{
    echo $what . PHP_EOL;
}
```

Configure a task like

```
$ bin/idx help yell
Usage:
    yell [what]

Arguments:
    what    What you want to yell (default: "foo")
```

4.3 Scripting with Idephix

With Idephix you compose your script basically:

- executing local commands
- executing remote commands
- executing other tasks you have already defined
- sending some output to the console

In order to perform such operations you will need an instance of the `Idephix\\Context` object. Idephix will inject it at runtime in each tasks that defines an argument type hinted as `Idephix\\Context`. A `Context` implements `\Idephix\TaskExecutor` and `\Idephix\DictionaryAccess` allowing you to execute commands and to access the configuration data related to the choosen `env`.

4.3.1 Executing local commands

`\Idephix\TaskExecutor::local` allows you to execute local commands. A local command will be executing without any need for a SSH connection, on your local machine.

```php
<?php

function buildDoc(\Idephix\Context $context, $open = false)
{
    $context->local('cp -r src/Idephix/Cookbook docs/');
    $context->local('make  -C ./docs html');

    if ($open) {
        $context->openDoc();
    }
}
```

If you need so you can execute the command in dry run mode

```
<?php

function buildDoc(\Idephix\Context $context, $open = false)
{
    $context->local('cp -r src/Idephix/Cookbook docs/', true);
}
```

In dry run mode the command will just be echoed to the console. This can be useful while debugging your idxfile to check the actual command that would be executed.

For local commands you can also specify a timeout:

```
<?php

function buildTravis(\Idephix\Context $context)
{
    try {
        $context->local('composer install');
        $context->local('bin/phpunit -c tests --coverage-clover=clover.xml', false, 240);
        $context->runTask('createPhar');
    } catch (\Exception $e) {
        $context->output()->writeln(sprintf("<error>Exception: \n%s</error>", $e->getMessage()));
        exit(1);
    }
};
```

4.3.2 Executing remote commands

Running remote commands is almost the same as running local commands. You can do that using `\Idephix\TaskExecutor::remote` method. Dry run mode works quite the same as for local commands, but mind that *at the moment is not possible to specify a timeout for remote commands*.

```
<?php

function switchToNextRelease(Idephix\Context $context, $remoteBaseDir, $nextRelease, $go = false)
{
    $context->remote(
        "
        cd $remoteBaseDir && \\
        ln -nfs $nextRelease current",
        !$go
    );
}
```

In order to execute a remote command you must specify an environment using `--env` option. If you fail to specify a valid env name you will get an error and the command will not be executed.

4.3.3 Executing user defined tasks

Every task that you define will be accessible as a method of the `Idephix\Context` object. Mind that you don't have to manually inject the `Context` object, Idephix will do that for you at runtime.

```
<?php

function buildDoc(\Idephix\Context $context, $open = false)
```

```
{
    $context->local('cp -r src/Idephix/Cookbook docs/');
    $context->local('make  -C ./docs html');

    if ($open) {
        $context->openDoc();
    }
}

function openDoc(\Idephix\Context $context)
{
    $context->local('open docs/_build/html/index.html');
}
```

4.3.4 Accessing configuration from tasks

`Idephix\Context` object gives you also access to every configuration defined for the current environment. Imagine you have defined this configuration:

```
<?php

$environments = array(
    'prod' => array(
        'hosts' => array('127.0.0.1'),
        'ssh_params' => array(
            'user' => 'ideato'
        ),
        'deploy' => array(
            'repository' => './',
            'branch' => 'origin/master',
            'shared_files' => array('app/config/parameters.yml'),
            'shared_folders' => array('app/cache', 'app/logs'),
            'remote_base_dir' => '/var/www/testidx',
            'rsync_exclude' => './rsync_exclude.txt',
        )
    ),
    'test' => array(//blablabla),
);
```

While executing a command using `--env=prod` option your tasks will receive a `Context` filled up with prod data, so you can access to it. `Context` allows you to access configuration data implementing php `\ArrayAccess` interface or through get `\Idephix\DictionaryAccess::get` method.

```
<?php

function deploy(Idephix\Context $context, $go = false)
{
    $sharedFiles = $context->get('deploy.shared_files', array());
    $repository = $context['deploy.repository'];
    //cut
```

4.3.5 Writing output to the console

Idephix is based on Symfony console component so you can send output to the user using the `\Symfony\Component\Console\Output\OutputInterface`. You can get the full `OutputInterface`

component through the `\Idephix\TaskExecutor::output` method or you can use the shortcut methods: `\Idephix\TaskExecutor::write` and `\Idephix\TaskExecutor::writeln`.

Here is an example of you you can send some output to the console.

```php
<?php

/**
 * This command will yell at you
 *
 * @param string $what What you want to yell
 */
function yell(\Idephix\Context $context, $what = 'foo')
{
    $context->writeln(strtoupper($what));
    $context->write(strtoupper($what) . PHP_EOL);
    $context->output()->write(strtoupper($what) . PHP_EOL);
    $context->output()->writeln(strtoupper($what));
}
```

Hint: For more information about `OutputInterface` read the official component documentation

CHAPTER 5

Extensions

Extensions are meant to wrap reusable code into a class that you can wire to your next Idephix project. If you find yourself writing the same task over and over again you may want to put it into an Extension so you can easily reuse it in every projects.

Hint: Extensions should be used wisely, for most cases a bunch of tasks that you copy and paste across projects is the best solution. We in the first place dropped the Deploy solution to a standard recipe that we include in out idxfile for each project. This ease the readability and the hackability of the procedure. An Extension will allow you to define reusable code, but it will hide it a little bit, so take this in consideration before implementing one

An Extension is identified by a name, and is capable of:

- registering new Tasks, so they will be directly available from CLI
- registering methods that will be hooked into the Idephix instance so you can use them within other tasks

5.1 Writing Extensions

An Extension is simply a class implementing *IdephixExtension* interface. This will require you do define a name and, TasksCollection and a MethodCollection. If your extension don't need to register new tasks or methods, you can simply return an empty collection (*IdephixTaskTaskCollection::dry()* or *IdephixExtensionMethodCollection::dry()*).

If you need an instance of the current *IdephixContext* within your extension, simply implement also the *IdephixExtensionIdephixAwareInterface* and you'll get one at runtime.

Only method registered by *::methods()* will be plugged into Idephix and will be available for other tasks to use:

```
<?php
class DummyExtension implements Extension
{

    public function doStuff(IdephixInterface $idx, $foo, $go=false)
    {
        //do some stuff
    }

    /** @return array of callable */
    public function methods()
    {
        return Extension\MethodCollection::ofCallables(
            array(
```

```
                new Extension\CallableMethod('doStuff', array($this, 'doStuff'))
            )
        );
    }

//cut
}
```

```
<?php
//your idxfile.php

function deploy(IdephixInterface $idx, $go = false)
{
    //your deploy business logic here
    $idx->doStuff($foo, $go)
}
```

If you want to expose some of your methods as CLI commands you need to define them as a task:

```
<?php

class DummyExtension implements Extension
{

    /** @return TaskCollection */
    public function tasks()
    {
        return TaskCollection::ofTasks(array(
            new Task(
            'doStuff',
            'DummyExtension helps you doing stuff',
            array($this, 'doStuff'),
            Collection::createFromArray(array(
                'foo' => array('description' => 'A nice description of foo')
            )),
        ));
    }

//cut
}
```

And the you'll also get to execute it directly from cli:

```
$ idx doStuff bar
```

Hint: Check out our available extensions to see more complex examples ..

5.2 Execution priority

Idephix will always try to execute code from the idxfile first, so if some function within the idxfile conflicts with some registered method or task, the code from the idxfile will be executed and the extension code will be ignored.

CHAPTER 6

Cookbook

6.1 Deploying with Idephix

Deploying a PHP application can be done in many ways, this recipe shows you our best strategy for a generic PHP application, and it is composed of several steps:

- Preparing the local project
- Preparing the remote server
- Syncing the project to the server
- Linking shared files across releases (configuration files, cache, logs, etc)
- Switching the symlink for the new release, finalizing the deploy

The recipe organize your code on the server using a directory hierarchy borrowed from Capistrano:

```
-- current -> /var/www/my_app_name/releases/20150120114500/
-- releases
|   -- 20150080072500
|   -- 20150090083000
|   -- 20150100093500
|   -- 20150110104000
|   -- 20150120114500
-- shared
    -- <linked_files and linked_dirs>
```

So you can keep multiple releases on the server and switch the current release just creating a symlink to the actual one you want to make current. This allows you to easily rollback from one release to another.

Listing 6.1: idxfile.php

```php
<?php

function deploy(Idephix\Context $context, $go = false)
{
    $sharedFiles = $context->get('deploy.shared_files', array());
    $sharedFolders = $context->get('deploy.shared_folders', array());
    $remoteBaseDir = $context->get('deploy.remote_base_dir');
    $rsyncExclude = $context->get('deploy.rsync_exclude');
    $repository = $context->get('deploy.repository');
    $deployBranch = $context->get('deploy.branch');
    $nextRelease = "$remoteBaseDir/releases/" . time();
```

```
    $linkedRelease = "$remoteBaseDir/current";
    $localArtifact = '.deploy';
    $context->prepareArtifact($localArtifact, $repository, $deployBranch, $go);
    $context->prepareSharedFilesAndFolders($remoteBaseDir, $sharedFolders, $sharedFiles, $go);
    try {
        $context->remote("cd $remoteBaseDir && cp -pPR `readlink {$linkedRelease}` $nextRelease");
    } catch (\Exception $e) {
        $context->output()->writeln('<info>First deploy, sending the whole project</info>');
    }
    $dryRun = $go ? '' : '--dry-run';
    $context->rsyncProject($nextRelease, $localArtifact . '/', $rsyncExclude, $dryRun, $go);
    $context->linkSharedFilesAndFolders($sharedFiles, $sharedFolders, $nextRelease, $remoteBaseDir, $
    $context->switchToNextRelease($remoteBaseDir, $nextRelease, $go);
}

function prepareArtifact(Idephix\Context $context, $localArtifact, $repository, $deployBranch, $go =
{
    $context->local(
        "
        rm -Rf {$localArtifact} && \\
        git clone {$repository} {$localArtifact} && \\
        cd {$localArtifact} && \\
        git fetch && \\
        git checkout --force {$deployBranch} && \\
        composer install --no-dev --prefer-dist --no-progress --optimize-autoloader --no-interaction
        ",
        !$go
    );
}

function prepareSharedFilesAndFolders(Idephix\Context $context, $remoteBaseDir, $sharedFolders, $shar
{
    $context->remote(
        "mkdir -p {$remoteBaseDir}/releases && \\
         mkdir -p {$remoteBaseDir}/shared",
        !$go
    );
    foreach ($sharedFolders as $folder) {
        $context->remote("mkdir -p {$remoteBaseDir}/shared/{$folder}", !$go);
    }
    foreach ($sharedFiles as $file) {
        $sharedFile = "{$remoteBaseDir}/shared/{$file}";
        $context->remote("mkdir -p `dirname '{$sharedFile}'` && touch \"$sharedFile\"", !$go);
    }
}
function linkSharedFilesAndFolders(Idephix\Context $context, $sharedFiles, $sharedFolders, $nextRele
{
    foreach (array_merge($sharedFiles, $sharedFolders) as $item) {
        $context->remote("rm -r $nextRelease/$item", !$go);
        $context->remote("ln -nfs $remoteBaseDir/shared/$item $nextRelease/$item", !$go);
    }
}

function switchToNextRelease(Idephix\Context $context, $remoteBaseDir, $nextRelease, $go = false)
{
    $context->remote(
        "
        cd $remoteBaseDir && \\
```

```
        ln -nfs $nextRelease current",
        !$go
    );
}
```

These tasks are based on several configuration options that you can define in your idxrc.php file:

Listing 6.2: idxrc.php

```
<?php

$environments = array(
    'prod' => array(
        'hosts' => array('127.0.0.1'),
        'ssh_params' => array(
            'user' => 'ideato',
//              'password'              => '',
//              'public_key_file'       => '',
//              'private_key_file'      => '',
//              'private_key_file_pwd' => '',
//              'ssh_port'              => '22'
        ),
        'deploy' => array(
            'repository' => './',
            'branch' => 'origin/master',
            'shared_files' => array('app/config/parameters.yml'),
            'shared_folders' => array('app/cache', 'app/logs'),
            'remote_base_dir' => '/var/www/testidx',
            'rsync_exclude' => './rsync_exclude.txt',
        )
    ),
);

return
    array(
        'envs' => $environments,
        'ssh_client' => new \Idephix\SSH\SshClient(),
        'extensions' => array(
            'rsync' => new \Idephix\Extension\Project\Rsync(),
        ),
    );
```

CHAPTER 7

Migrating your idx file

Since version 0.2.0 we started supporting a new idxfile format. We think the new format is more easy to write and we will drop support for the old format soon. If you're still using the old idxfile format you can avoid migrating right now, just be sure to create an instance of *IdephixConfig* to construct your Idephix instance instead of the array you're using right now.

Implementing this should be easy enough as the *IdephixConfig* object can be created from an array, see *Configuration* for more information.

If you're brave enough and want to jump on the innovation wagon right now read writing_tasks on how to update your idxfile.

CHAPTER 8

Welcome to Idephix's documentation!

Idephix is a PHP automation tool useful to perform **remote** and **local** tasks. It can be used to deploy applications, rotate logs, synchronize data repository across server or create a build system. The choice is up to you. Idephix is still in alpha, so things will change. You can report issues and submit PRs (greatly appreciated :-)) on the github repo

Basically what you're going to do is define a bunch of function in a php file and execute them from the command line.

```php
<?php
/**
* This command will yell at you
*
* @param string $what What you want to yell
*/
function yell(\Idephix\Context $context, $what = 'foo')
{
   $context->writeln(strtoupper($what));
}
```

```
$ bin/idx yell "say my name"
SAY MY NAME
```

CHAPTER 9

Requirements

PHP 5.3.2 or above, at least 5.3.12 recommended

CHAPTER 10

Authors

Manuel ‘Kea’ Baldassarri, Michele ‘Orso’ Orselli, Filippo De Santis and other contributors

CHAPTER 11

License

Idephix is mantained by ideato, licensed under the MIT License.