Bldr.io Documentation

Release 0.0.2

Aaron Scherer

July 15, 2014

Contents

Bldr, in the simplest terms, is a task runner, and an awesome one at that. It was written with simpler configs in mind. If you are used to build systems, you've probably seen some pretty complicated build files, and they were probably written in xml that is clunky and a pain to maintain.

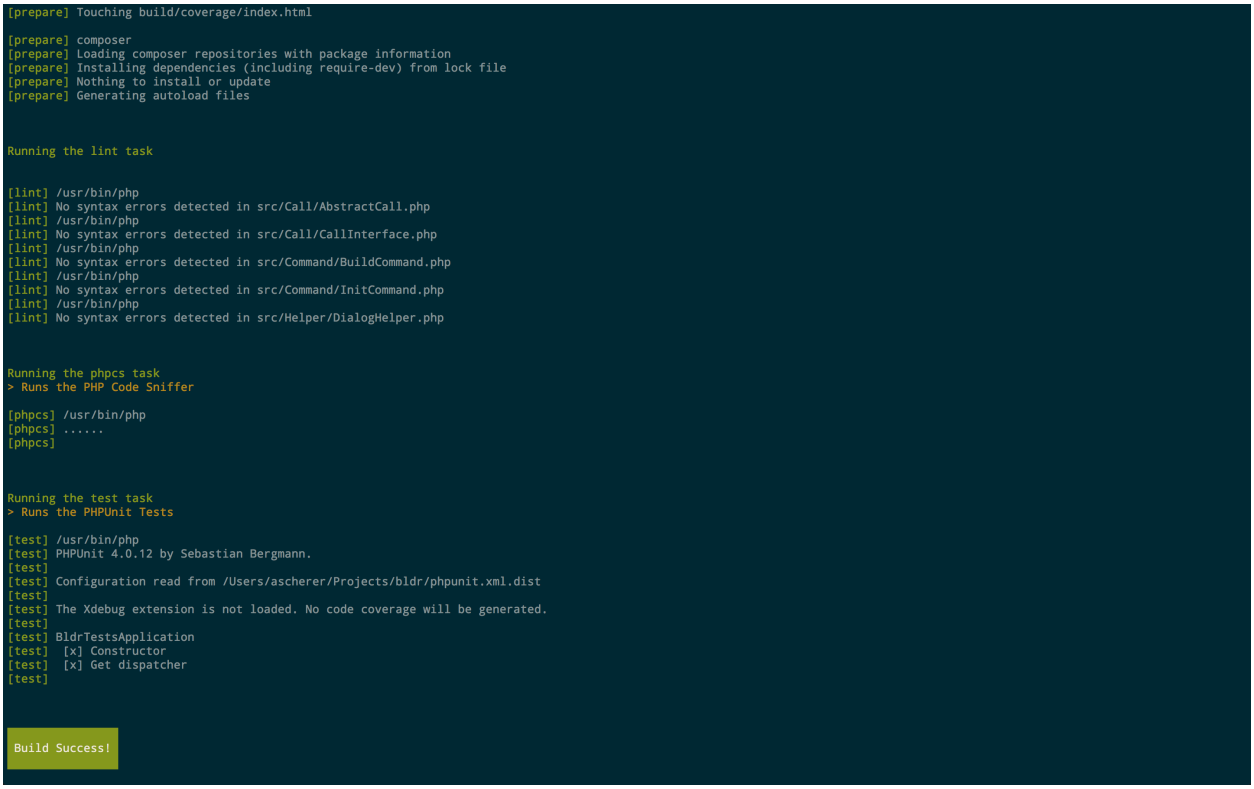
Well, here's one written for Bldr using yaml (json is also supported):

This is a sample configuration. Your project may not have the dependencies required to run this configuration.

```
bldr:
    name: bldr-io/bldr
    description: Super Extensible and Awesome Task Runner

    profiles:
        default:
            description: Development Profile
            tasks:
                - prepare
                - lint
                - phpcs
                - test

    tasks:
        prepare:
            description: Cleans up old builds and prepares the new one
            calls:
                -
                    type: filesystem:remove
                    files: [build/coverage, build/logs]
                -
                    type: filesystem:mkdir
                    files: [build/coverage, build/logs]
                -
                    type: filesystem:touch
                    files: [build/coverage/index.html]
                -
                    type: exec
                    executable: composer
                    arguments: [install, --prefer-dist]
        lint:
            description: Lints the files of the project
            calls:
                -
                    type: apply
                    failOnError: true
                    src:
                        - { path: [src, tests], files: *.php, recursive: true } # Checks src and test
                    executable: php
                    arguments: [-l]

        phpcs:
            description: Runs the PHP Code Sniffer
            calls:
                -
                    type: exec
                    executable: php
                    arguments:
                        - bin/phpcs
                        - -p
                        - --standard=build/phpcs.xml
```

```
                    - --report=checkstyle
                    - --report-file=build/logs/checkstyle.xml
                    - src/
        test:
            description: Runs the PHPUnit Tests
            calls:
                -
                    type: exec
                    failOnError: true
                    executable: php
                    arguments:
                        - bin/phpunit
                        - --testdox
                        - --coverage-text=php://stdout
```

And here's the output:



For now (while we are still working on the documentation), this will hopefully serve as ample documentation.

CHAPTER 1

Content

1.1 Installation

This should be installed via composer, for now. You can either install it globally, or in your own project. If it is the first time you globally install a dependency then make sure you include `~/.composer/vendor/bin` in `$PATH` as shown here.

Global Setup

```
$ composer global require bldr-io/bldr=~4.2.0 dflydev/embedded-composer=dev-master@dev composer/compo

# Or

$ curl -sS http://bldr.io/installer | php
$ mv bldr.phar /usr/bin/bldr
```

Project Setup

It is suggested that you use the phar, as you can get conflicts with dependencies by including it in your project or globally!

```
$ curl -sS http://bldr.io/installer | php

# Or

$ composer require bldr-io/bldr "~4.2.0"
```

And that's it! From here, you should be able to run `bldr` if you set it up globally, or `./bin/bldr` or `php ./vendor/bin/bldr` if you set it up in your project.

1.2 Usage

To start, you are going to want to generate a `.bldr.yml` file for your project. This (for now) has to be done manually, but it's pretty simple.

Create a `.bldr.yml(.dist)` file:

```
bldr:
    name: some/name
    description:  A description about the project # (Not Required)
    profiles: # A list of profiles that can be ran with `./bldr.phar build`
        someTask:
```

```
            description: Gets ran when `./bldr.phar build someTask` is called
            tasks:
                - foo
    tasks:
        foo:
            description: FooBar task
            calls:
                -
                    type: exec
                    executable: echo
                    arguments: [Hello World]
```

To view a list of available call types, run:

```
./bldr.phar task:list
```

And to get more information on a particular type, run:

```
./bldr.phar task:info <task name>
```

To run your profiles: (This has changed since version 4)

```
./bldr.phar build <profile name>
```

1.3 Extensions

When starting Bldr, I wanted it to be highly extensible, and because of that, I decided to use the Symfony2 Dependency Injection component, but Bldr takes it a step further. Its mostly just a rename with a little added functionality for ease of use.

With that, we've come up with the idea for *Bldr Blocks*. *Bldr Blocks* are basically Symfony Extensions (with a little difference), that add some functionality to bldr. There are a bunch of *Bldr Blocks* already in the core of *bldr*, and they are listed below. To find other blocks, we are working on a site where people will be able to upload links to their repositories. But in the mean time, just use the list below. If I missed anything, you can find it here: https://www.versioneye.com/php/bldr-io:bldr/references

- Frontend Block - Used for tasks like CSS/JS Minification and Less/Sass/SCSS/Coffeescript compilation
- Gush Block - Used for integrating with Gush
- [Out Of Date] Symfony Block - Used for integrating with the Symfony2 Framework
- [Out Of Date] Git Block - Used for integrating with git

To add your own task types to Bldr, you will have to write your own *Bldr Block*. By default, there are a couple blocks that already come with Bldr, but it is easy enough to add your own. For a quick baseline on how to write one, check out the documentation.

Adding third party blocks is a three step process. First, you need to create a *bldr.json* file (if you don't have one):

```
{
    "require": {
        "acme/demo-block": "@stable"
    }
}
```

Then, install/update your bldr dependencies:

```
./bldr.phar install
# OR
./bldr.phar update
```

Then, add it to your `.bldr.yml` file:

```
bldr: ~

blocks:
    - Acme\Block\Demo\AcmeDemoBlock

# If you have configs
acme_demo:
    some_setting: some_value
```

Below is some minor documentation on the core blocks.

1.3.1 Execute Block (Official)

The Execute Block (Included with Bldr)

This extension lets you run `exec` and `apply` tasks.

```
tasks:
    sample:
        calls:
            -
                task: exec
                executable: php
                arguments: [bin/phpcs]
            -
                task: apply
                executable: php
                output: /dev/null
                src:
                    - { path: [src, tests], files: *.php, recursive: true } # Checks src and tests di
                arguments: [-l]
```

1.3.2 Filesystem Block (Official)

The Filesystem Block (Included with Bldr)

This extension lets you run filesystem commands.

This one needs some work, as not all of the commands are there (mkdir, remove, touch, and dumpFile are).

Some examples:

```
tasks:
    sample:
        calls:
            -
                task: filesystem:mkdir
                files: [testDir]
            -
                task: filesystem:remove
```

```
            files: [testDir]
        -
            task: filesystem:touch
            files: [test.tmp]
```

1.3.3 Notify Block (Official)

The Notify Block (Included with Bldr)

This extension lets you run the `notify` commands. It will either print to the screen, or email a message.

To use this:

```
tasks:
    sample:
        calls:
            -
                task: notify
                message: Test Message
                email: test@gmail.com
```

When adding this extension, you can specify *smtp* connections:

```
blocks:
    - Bldr\Block\Notify\NotifyBlock

notify:
    smtp:
        host: smtp.google.com
        port: 465
        security: ssl
        username: google
        password: is4wesome
```

1.3.4 Watch Block (Official)

The watch Block (Included with Bldr)

This extension lets you run the `watch` commands. It will let you watch the filesystem for changes.

This one needs some work. Right now, you can only have one watch task.

```
tasks:
    sample:
        calls:
            -
                task: watch
                src:
                    - { path: [src, tests], files: *.php, recursive: true } # Checks src and tests di
                    - { path: vendor/, files: [*.php, *.yml], recursive: true } # Checks vendor/ dire
                profile: someProfile
    sample2:
        calls:
            -
                task: watch
                src:
                    - { path: [src, tests], files: *.php, recursive: true } # Checks src and tests di
```

```
            - { files: *.yml } # Checks current directory, non-recursively
        task: someTask
```

1.4 Creating A Block

Creating a *Bldr Block* for *Bldr* is fairly similar to creating a Bundle for Symfony2. Here's a quick guide:

1.4.1 1. Create a repo

Try and stay with the naming convention used by the other blocks: <name>-block

1.4.2 2. Initialize composer in the repo

```
cd your-repo && composer init
```

1.4.3 3. Add Bldr as a dev dependency

In your composer.json file, you will want to add *bldr-io/bldr* as a *require-dev* dependency. Because embedded composer and composer are unstable packages by definition or they do not have a stable release you will have to add them too into your composer.json as below:

```
{
    "require-dev": {
        "bldr-io/bldr":               "~4.2.0",
        "dflydev/embedded-composer": "dev-master@dev",
        "composer/composer":          "dev-master@dev"
    }
}
```

1.4.4 4. Create a Block class

All of bldr, and the official extensions follow PSR-4 (as well as all the other PSR's, and most, if not all, of the bylaws). With that, create your directory structure and your Block class:

```
mkdir src && vim src/AcmeDemoBlock.php
```

All blocks must extend the BldrDependencyInjectionAbstractBlock, so your class, empty, will look something like this:

src/AcmeDemoBlock.php

```
<?php

/**
 * License Information
 */

namespace Acme\Block\Demo;

use Bldr\DependencyInjection\AbstractBlock;
use Symfony\Component\DependencyInjection\ContainerBuilder;
```

```php
/**
 * @author John Doe <john@doh.com>
 */
class AcmeDemoBlock extends AbstractBlock
{
    /**
     * {@inheritDoc}
     */
    protected function assemble(array $config, ContainerBuilder $container)
    {
    }
}
```

The assemble function is where the magic happens. If you take a look at the AbstractBlock, there are some helper functions in there to make it easier to add new calls, services, and parameters to the Container.

1.4.5 5. Create your Call

As a demo, let's say we want to make a call that will output a random number to the user when running the call.

First, lets create the call. Directory structure doesn't really matter, but the core structure is normally *src/Call/<Name>Call.php*. Similar to blocks, all calls must extend **'Bldr\Call\AbstractCall'_**.

Lets make the *src/Call* directory, and create the new Call: .. code-block:: shell

    mkdir src/Call && vim src/Call/OutputRandomNumberCall.php

Then, let's build the call class! Extending the AbstractCall, requires that we implement two methods: configure and run

src/Call/OutputRandomNumberCall.php

```php
<?php

/**
 * License Information
 */

namespace Acme\Block\Demo\Call;

use Bldr\Call\AbstractCall;

/**
 * @author John Doe <john@doh.com>
 */
class OutputRandomNumberCall extends AbstractCall
{
    /**
     * {@inheritDoc}
     */
    public function configure()
    {
        $this->setName('acme_demo:output_random_number')
            ->setDescription('This call outputs a random number. If min and max are specified, it wi
            ->addOption('min', true, 'Minimum number in range', 0)
            ->addOption('max', true, 'Maximum number in range', 100)
        ;
    }
```

```
    /**
     * {@inheritDoc}
     */
    public function run()
    {
        $random = rand($this->getOption('min'), $this->getOption('max'));
        $this->output->writeln(['', 'Random Number: '.$random, '']);

        return true;
    }
}
```

Next, we need to add the call to the container, so we can use it in .bldr.yml files:

src/AcmeDemoBlock.php

```
<?php

/**
 * License Information
 */

namespace Acme\Block\Demo;

use Bldr\DependencyInjection\AbstractBlock;
use Symfony\Component\DependencyInjection\ContainerBuilder;

/**
 * @author John Doe <john@doh.com>
 */
class AcmeDemoBlock extends AbstractBlock
{
    /**
     * {@inheritDoc}
     */
    protected function assemble(array $config, ContainerBuilder $container)
    {
        // Here's one of the shortcut methods! This method will return a Symfony DI Definition
        // that is tagged as `bldr`. If you need to, you can easily add arguments to the constructor,
        // or calls to methods.
        $call = $this->addCall('acme_demo.output_random_number', 'Acme\Block\Demo\AcmeDemoBlock');

        // If you need dependencies, you could do the following:
        // $call->setArgument(0, new Reference('some_service'));
        // or
        // $arguments = array(new Reference('some_service'));
        // $call->addMethodCall('someMethodName', $arguments);

        // If you want to add a service, that isn't a call, you can also use:
        // $this->addService($name, $class);
        // Which will also return a Symfony DI Definition
    }
}
```

With this, you should be able to add it to a .bldr.yml file:

```
blocks:
    - Acme\Block\Demo\AcmeDemoBlock
```

```
bldr:
    name: some/name
    profile:
        default:
            tasks:
                - randomize

    tasks:
        randomize:
            calls:
                -
                    type: acme_demo:output_random_number
                    min: 0
                    max: 100000
```

And run it!

```
./bldr.phar build -p default
```

There's some more advanced stuff, like being able to specify configuration:

src/AcmeDemoBlock.php

```php
<?php

/**
 * License Information
 */

namespace Acme\Block\Demo;

use Bldr\DependencyInjection\AbstractBlock;
use Symfony\Component\DependencyInjection\ContainerBuilder;

/**
 * @author John Doe <john@doh.com>
 */
class AcmeDemoBlock extends AbstractBlock
{
    // ...

    /**
     * {@inheritDoc}
     */
    protected function getConfigurationClass()
    {
        return 'Acme\Block\Demo\Configuration';
    }
}
```

1.4.6 6. Advanced Config

Then make a Configuration.php file. This config is the config from symfony. You can read their docs for more information.

src/Configuration.php

```php
<?php

/**
 * License Information
 */

namespace Acme\Block\Demo;

use Symfony\Component\Config\Definition\ConfigurationInterface;
use Symfony\Component\Config\Definition\Builder\TreeBuilder;

/**
 * @author John Doe <john@doh.com>
 */
class Configuration implements ConfigurationInterface
{
    /**
     * {@inheritDoc}
     */
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode    = $treeBuilder->root('acme_demo');

        // here you will build the configuration tree

        return $treeBuilder;
    }
}
```