# Muse

## Overview

Muse, with its connotation of inspiration and creativity, is the HUBzero framework for command line tools and automation. Muse, by default, has commands for running migrations, clearing the cache, creating scaffolding, updating your hub, and more. In addition to the default commands, Muse can also be extended by individual components to provide component specific tools and command line functionality. We'll walk through many of the detailed commands below, and then give a brief description of how you can add your own commands to Muse.

Note that all commands below are assumed to come from your hub's document root.

So, to run muse, simply type:

```
php muse
```

This, like many other commands, will return your available options by default. The current list of top level commands includes:

- Cache
- Configuration
- Database
- Environment
- Extension
- Group
- Log
- Migration
- Repository
- Scaffolding
- Test
- User

As a developer, you may find yourself in a given moment as either a consumer of existing commands, or a creator of new commands,  To understand the existing commands, jump over to the commands chapter for more details about each command.  Continue below to learn more about creating your own commands.

## Structure

Muse by default will look for commands in the Commands directory within the Hubzero Console Library. If you're looking to add a new core command, this is where it will live.

The name of the command file becomes the name of the command itself. So, for example, the Database command would be found at:

```
core/libraries/Hubzero/Console/Command/Database.php
```

Within the file itself, all public methods will be considered tasks that can be called on the command. Private and protected methods will not be directly routable. To exemplify this, you'll notice that the Database command has two tasks, dump and load. These are public methods with the Database.php command class. At a minimum, commands are to implement the CommandInterface, which requires three methods:

```
public function __construct(Output $output, Arguments $arguments);
public function execute();
public function help();
```

By extending the base command, you can further simplify things to only need the execute and help methods. The execute task is the default task and is called when no task is explicity given. The help command should establish meaninful descriptions of tasks and arguments available.

Often times it will make sense to simply route the execute command to the help command, thus giving users an overview of your command and options by default

You can also namespace your commands. And by this we simply mean that you can use folders to create logical subdivisions within your commands. You'll see this, for example, in the Configuration command. The configuration command has two subcommands, aliases and hooks. To call tasks on these commands, you simply:

```
php muse configuration:hooks add ...
php muse configuration:aliases help
```

## Arguments and Output

Within the command there are two primary objects of interest on the command, the arguments and the output.

### Arguments

The primary function of the arguments class is to provide the command with access to the extra arguments passed into the command by the user. There are really two primary styles or ways of structuring a command arguments. For required commands, we typically use an ordered variable approach to these arguments. Consider the scaffolding command. It expects a task of the scaffolding action we are to perform, such as create or copy. We then expect the type of item we will be scaffolding. This ultimately will look as follows:

```
php muse scaffolding create migration
```

Then, to access these types of arguments, we simply grab them by their index order:

```
$type = $this->arguments->getOpt(3);
```

The index numbers follow the underlying values from PHP's native arguments, where the script is 0, the command is 1, the task is 2, and so on from there.

In addition to this initial style of argument, you can also accept named arguments. These are often optional sorts of arguments, such as:

```
php muse scaffolding create migration --install-dir=/altlocation
```

And these would be accessed in a similar manner:

```
$installDir = $this->arguments->getOpt('install-dir');
```

### Output

Throughout the course of your command, it's important to let the user know what you're doing, and whether or not everything was successful. To do that, we use the output object on the command. The primary methods of interest are:

```
$this->output->addLine('Hello');
```

```
$this->output->addString('hello');
$this->output->error('Something went wrong!');
```

Hopefully the method names are fairly self-explanatory. The addLine method adds the given string along with a newline, whereas the addString simply outputs the given message. The error command outputs the given message with error styling, and also stops execution immediatly (this is important!).

Both the addLine and addString methods accept a second argument specifying a style for the message. Available shortcut strings include: warning, error, info, and success. More fine-grained control can be achieved by passing an array as the second parameter. This array can have up to three arguments, specifying a format, color, and indentation. The available formats include:

- normal
- bold
- underline

And available colors include:

- black
- red
- green
- yellow
- blue
- purple
- cyan
- white

It's important to remember that care should be taken when specifying colors, as a given user's console styles may make reading certain colors more difficult.

Here are some examples of using the message styles:

```
$this->output->addLine('All done here', 'success');
$this->output->addLine('Something went wrong!', ['color' => 'red', 'fo
rmat' => 'bold']);
```

## Documentation

Documenting your commands is a good practice, both for you and for those that will be using your commands. All commands are required to have a help function. That function will be used to output the appropriate help info for the command. A typical help method will look something like this:

```
public function help()
{
    $this
         ->output
         ->addOverview(
             'This is my command for doing great things'
         )
         ->addTasks($this)
         ->addArgument(
             '--awesome-level: Set the awesomeness level',
             'Specify the desired level of awesomeness',
             'Example: --awesome-level=7'
         );
}
```

The methods available for help documentation fairly straight-forward. The overview section, generated by addOverview, is the main description of the command. The addTasks method is used generate a list of available tasks within the command. Finally, the addArgument method can be used to specify the available arguments that your command accepts.

The addTasks method generates the available tasks list based on public methods, as mentioned above. To define the description for the method, include the @museDescription tag in the method docblock, as shown below.

```
/**
 * Creates awesomeness
 *
 * @museDescription Constructs and does important things
 *
 * @return void
**/
```

The result of the above examples would render like this:

```
me@myhub.org:~# muse mycommand help
Overview:
  This is my command for doing great things

Tasks:
  create   Constructs and does important things

Arguments:
  --awesome-level: Set the awesomeness level
    Specify the desired level of awesomeness
    Example: --awesome-level=7
```

## Interactivity

Interactivity is a cool feature of Muse. This allows a more guided experience for users. For example, instead of requiring users to provide four arguments, you can prompt for them, or even tailor them based on previous arguments. An example of this can be found in the extension command.

```
me@myhub.org:~# muse extension
What do you want to do? [add|delete|install|enable|disable] add
What extension were you wanting to add? com_awesome
Successfully added com_awesome!
```

To display a prompt to the user, simply use the getResponse method on the output object.

```
$name = $this->output->getResponse("What extension were you wanting to
 add?");
```

This will wait for a response and enter from the user.

**When not to be interactive?**

Interactivity is not always desired. If a user has set the non-interactive flag, or the current output mode is non-standard, it becomes important to not wait for user input. To ensure proper functionality in different environments and output formats, you should wrap all interactive calls in

the isInteractive check and provide an appropriate alternative (likely just checking for a given argument).

```
// Check for interactivity
if ($this->output->isInteractive())
{
    // Prompt for action
    $action  = $this->output->getResponse('What do you want to do?');
}
else
{
    // Otherwise show help output so user knows available options
 $this->output = $this->output->getHelpOutput();
 $this->help();
 $this->output->render();
 return;
}
```

## Component Commands

In addition to the basic command library, individual commponents can contain commands as well. This makes adding site-specific commands easier (without modifying core HUBzero), as well as allowing for a more logical grouping of functionality with other component-specific models.

Site commands work in exactly the same manner as library commands, but are simply located in an alternate place.

`app/components/mycomponent/cli/commands/mycommand.php`

Commands must still implement the command interface, and should function the same way as library commands. They will not however, show up in the master command list obtained when calling the global muse help command

# Commands

## Cache

The cache command is a helper for clearing your sites cache files. You can clear the entire cache, or just the CSS cache. Those commands, respectively, are:

```
php muse cache clear

php muse cache:css clear
```

## Configuration

The configuration command is used to personalize and customize your Muse experience. It's also used to store variables for repeated use. For example, the scaffolding command will ask you, if you haven't already, to set your name and email to be used when generating files.

```
muse configuration set --user_name="John Doe"
muse configuration set --user_email=john.doe@gmail.com
```

Configuration can also be used to store hooks and aliases. Hooks are additional commands that are run at pre-defined points. Aliases are command shortcuts. Here are some examples:

```
# run permissions fix after updating the repository
muse configuration:hooks add repository.afterUpdate "chmod -R g+w /www
/docroot"

# Add a shortcut for the environment command
muse configuration:aliases add env environment
```

## Database

The database command was added for two primary reasons - the first backups, and the second, reverse content migration. Backups are fairly straight-forward, but a little more detail is in order for reverse content migration.

If you have an environment with more than one stop in your production flow, you've likely run into the problem of wanting to move data from prod to dev for testing purposes. But in so doing, you often overwrite some site-specific configuation on dev. So get around this, we perform a dump and load using the database command to move only those things that should move between environments.

```
# dump the database
muse database dump

# then make sure you copy to your dev environment
# then from dev, load the dump back up (it will have a different name)
muse database load filenamefromabovecommand
```

## Environment

The environment command simply outputs the current environment variables.

```
Current user     : Mr Awesome <awesome@gmail.com>
Current database : example
```

## Extension

If you don't already know, extensions are the general name for all of the 'apps' allowed by the HUBzero framework. They include (amoung some others), templates, components, modules, and plugins. When adding a new extension, you will often want to add it to the extensions database table and enable it. This command can help save you trips directly to the database.

The nice thing to about the extension command is that it will prompt you for what it needs, you don't really need to remember the syntax.

```
me@me.org:~# muse extension
What do you want to do? [add|delete|install|enable|disable] add
What extension were you wanting to add? com_awesome
Successfully added com_awesome!
```

Or, as another example. Let's delete that entry we added above using the written out syntax

```
me@me.org:~# muse extension delete --name=com_awesome
Successfully deleted com_awesome!
```

Note that if you're in a production environment and using migrations, this command is redundent. Use migrations! But if you're just testing and need a quick way to enable or disable something, this is the way to go.

## Group

The group commands are simply wrappers on existing commands to be used within the super group context. Please review the super group documentation for more details.

## Log

The log command is great for following and filtering log entries. There are currently two log types available, the profile log and the query log. To start, simply:

```
muse log follow profile
```

You have to having logging enabled for new entries to be displayed!

Once started, you'll see info on the log fields being displayed.

```
me@me.org:~# muse log follow profile
The profile log has the following format (* indicates visible field):
<0:*timestamp> <1:*hubname> <2:*ip> <3:*app> <4:*uri> <5:*quer
y> <6:*memory> <7:*querycount> <8:*timeinqueries> <9:*totaltime>
```

To toggle a fields visibility, simply press the number next to the field of interest. For example, pressing 2, and then f to show the fields again, results in:

```
> Hiding ip
> The profile log has the
following format (* indicates visible field):
<0:*timestamp> <1:*hubname> <2:ip> <3:*app> <4:*uri> <5:*quer
y> <6:*memory> <7:*querycount> <8:*timeinqueries> <9:*totaltime>
```

To show the available commands, simply type h.

```
> q: quit, h: help, i: input mode, p: pause/play, b: beep on/off, f: f
ields, r: rerender last 100 lines
```

## Migration

For more info on the migration command, see the dedicated migrations section under the database chapter.

## Repository

The repository command offers an abstraction on top of the mechanism used to manage and update the CMS. This could include GIT, HTTP-based package installs, or Debian packages. Currently, GIT is the only supported mechanism, but more are to come in the future.

To start, simply see if the repository command is supported in your environment.

```
me@me.org:~# muse repository
This repository is managed by GIT and is clean
```

If you environment is not currently supported, you'll receive a message like this:

```
me@me.org:~# muse repository
Sorry, this command currently only supports setups managed by GIT
```

To start the update process, use the update task. Depending on your current state, you'll either see that you're up-to-date, or see what's coming in the next update.

```
me@me.org:~# muse repository update
The repository is already up-to-date
```

or...

```
me@me.org:~# muse repository update
The repository is behind by 747 update(s):
...
```

Then, to perform the actual update, add the -f flag.

```
me@me.org:~# muse repository update -f
Updating the repository...complete
```

If something goes wrong, the update mechanism will automatically roll back to it's state prior to attempting the update. Then you'll have to go in a manually perform the update depending on the mechanism.

**Spring Cleaning**

In addition to performing updates, the repository command also offers some help doing periodic cleanup. Using the clean command will allow you to prune rollback points and stashes.

```
me@me.org:~# muse repository clean
Do you want to purge all rollback points except the latest? [y|n] y
Purging rollback points.
Do you want to purge all stashed changes? [y|n] y
Purging repository stash.
Clean up complete. Performed (2/2) cleanup operations available.
```

## Scaffolding

Scaffolding was create to help developers get started quickly. Let's be honest, developers rarely start from a blank file. We copy something existing and modify. With scaffolding, we give you a template a pre-fill known values to make this process even easier.

At this time, scaffolding knows how to create:

- Commands
- Components
- Migrations
- Tests

So, for example, to create a new component, simply:

```
me@me.org:~# muse scaffolding create component com_awesome
Creating /var/www/example/core/components/com_awesome/awesome.xml
Creating /var/www/example/core/components/com_awesome/admin/awesome.ph
p
Creating /var/www/example/core/components/com_awesome/admin/controller
s/awesome.php
Creating /var/www/example/core/components/com_awesome/admin/language/e
n-GB/en-GB.com_awesome.ini
Creating /var/www/example/core/components/com_awesome/admin/language/e
n-GB/en-GB.com_awesome.sys.ini
Creating /var/www/example/core/components/com_awesome/admin/views/awes
ome/tmpl/display.php
Creating /var/www/example/core/components/com_awesome/api/controllers/
api.php
Creating /var/www/example/core/components/com_awesome/config/access.xm
l
Creating /var/www/example/core/components/com_awesome/config/config.xm
l
Creating /var/www/example/core/components/com_awesome/models/awesomes.
php
Creating /var/www/example/core/components/com_awesome/site/awesome.php
Creating /var/www/example/core/components/com_awesome/site/assets/css/
awesome.css
Creating /var/www/example/core/components/com_awesome/site/assets/js/a
wesome.js
Creating /var/www/example/core/components/com_awesome/site/controllers
/awesome.php
Creating /var/www/example/core/components/com_awesome/site/language/en-
GB/en-GB.com_awesome.ini
Creating /var/www/example/core/components/com_awesome/site/router.php
Creating /var/www/example/core/components/com_awesome/site/views/aweso
mes/tmpl/display.php
Creating /var/www/example/core/components/com_awesome/site/views/aweso
```

```
mes/tmpl/edit.php
```

As you can see, this automatically generates all of the core files and views you're likely to need. It also names them appropriately, as well as using the provided component name to even tweak the contents of these files.

## Test

Testing is critical to both deploying a new extension, and updating existing extensions without too much heartache. To facilitate testing, muse offers a framework and wrapper around the popular PHP Unit testing infrustucture.

To see the current extensions with tests, run:

```
me@me.org:~# muse test show
lib_database
```

Then, to run a specific extensions tests, you can use the run command.

```
me@me.org:~# muse test run lib_database
PHPUnit 4.6.2 by Sebastian Bergmann and contributors.

..................................................

Time: 2.26 seconds, Memory: 17.5Mb

OK (51 tests, 73 assertions)
```

## User

The final command available at this time is the user command. It offers some advances administrative functionality for merging and unmerging users.

This command is experimental!

Occasionally, on a hub, one person will create two accounts and not realize it. They later ask

you to merge the accounts and move the contributions from one to the other. This isn't a simple task, and involves updating many, many references in the database. Fortunately for you, we've been working on a solution.

```
me@me.org:~# muse user merge 1042 into 1003
Updating (1) item(s) in jos_collections.object_id
Updating (1) item(s) in jos_collections.created_by
Updating (1) item(s) in jos_collections_items.created_by
Updating (8) item(s) in jos_courses_asset_groups.created_by
Updating (15) item(s) in jos_courses_assets.created_by
Updating (1) item(s) in jos_courses_members.user_id
Updating (2) item(s) in jos_courses_offering_section_dates.created_by
Updating (2) item(s) in jos_courses_units.created_by
Updating (76) item(s) in jos_developer_access_tokens.uidNumber
Updating (1) item(s) in jos_developer_applications.created_by
Updating (1) item(s) in jos_developer_rate_limit.uidNumber
Updating (9) item(s) in jos_users_log_auth.user_id
Ignoring jos_users_password.user_id due to integrity constraint violat
ion
Updating (1) item(s) in jos_users_points.uid
Ignoring jos_xprofiles_bio.uidNumber due to integrity constraint viola
tion
Updating (3) item(s) in jos_xprofiles_tokens.user_id
```

Then, if needed, you can reverse the merge.

```
me@me.org:~# muse user unmerge 1042 from 1003
Unmerged (122/122) records successfully!
```