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