creations configuration

A guide on how to configure the creations CLI – Version 0.0.0

Creation types

One of the core aspects of `creations` is creation types. These define what template will be used to scaffold your project, among other things. For example, when you run `creations new webapp/nuxt`, creations will use the template located at `~/.config/creations/templates/webapp.nuxt`.

Folder structure

Inside `~/.config/creations/templates`, each folder is the name of a creation type. Here's the structure:

```
~/.config/creations/templates/
 webapp.nuxt/
  template.toml
  new/ # Template for project creation (`creations new`)
  add/ # Templates for project additions (`creations add`)
   component/ # Template used when running `creations add component`
   page/
   store-module/
```

Let's first look at the `template.toml` file.

```
[new]
in = "/mnt/d/projects/{{ name~slug }}"

[add.component]
in = "@/components/{{ name~slug }}"

[add.store-module]
in = "@/store/{{ name~slug }}"
```

Don't be afraid of `{{ name~slug }}` or `@/`, these are explained respectively in Substitutions and Paths

---

Substitutions

creations tries its best to follow the widely-accepted {{ mustache }} templating engine. It drifs from the standard to add filters, akin to Django's templating syntax.

Filters

Due to folder and file names limitations on Windows, we can't use `|` to indicate filters. Instead, the character `~` is used (reminiscent of the "transform arrow" `~>` sometimes used).

Filters can be chained:

With `name = "Hello, World!"`

```
{{ name~slug~uppercased }}
```

will produce `HELLO-WORLD`.

> The order of filters matters. In some cases, this will affect the output. Naturally,

WARNING | the order of chaining matches the calling order: `{{ name~slug~uppercased }}` will do `uppercased(slug(name))`.

Built-in filters

slug

Produces a slug, replacing any non-alphanumeric characters by dashes, and trimming dashes from the end of the produced string.

uppercase

Converts to UPPERCASE

lowercase

Converts to lowercase

camelcase

Converts to camelCase

snakecase

Converts to snake_case

kebabcase

Converts to kebab-case

uppercasefirst

Converts the first character to uppercase

pascalcase

Shortcut for `snakecase~uppercasefirst`

constantcase

Shortcut for `snakecase~uppercase`

titlecase

Converts to Title Case

Adding your own

You'll notice that, inside of your creations directory, a `filters` folder is present.

Each file in this directory should be an executable that takes stdin as an input and ouputs the result to stdout.

Moreover, you'll notice that some files are already in this directory: these are the built-in filters. You can modify these, but it's not recommended as it hurts portability.

Variables

The following variables are defined by default.

name	The project's name, as typed in the command

type The project type

For example, when you run `creations new webapp/svelte portfolio`:

```
name = "portfolio"
type = "webapp/svelte"
```

For other variables, they will be asked when encountered for the first time during rendering. To define defaults, types and others properties, see Asking

---

Paths

Some special sequences are resolved in paths, namely:

- `@` Current project's directory.
- `~` Home folder (Supports windows user directory)
- `.` Current working directory

Running scripts

You can run custom scripts *after* and *before* the templating is done. In your `template.toml` file, do:

```
[<command>.execute]
before = [ "script-to-run-before" ]
after  = [ "script-to-run-after"  ]
```

Each command provides a hook, and the hook's name matches the command's.

For example:

```
[idea.execute]
after = [ "github-projects \"{{ project }}\" column \"{{ column }}\" add card \"{{ idea }}\"" ]
```

Reactions based on the return code

When the code is non-zero, stderr is shown to the user as an error message, the command is aborted.

If the code *is* zero, but stderr is not empty, it is shown to the user as a warning message and the commmand **continues**.

`before` scripts

For `before` scripts, the stdout must be a JSON object containing at least `name` and `type` properties, and will be used as input for the real command.

For example, assume the following setup for a type named `webapi`:

template.toml

```
[on.new]
before = [ "example.sh" ]
```

example.sh

```
return '{
 "name": "overloaded!!!",
 "type": "webapi"
}'
```

When running `new webapi my-thing`, `creations` will create a new `webapi` named `overloaded!!!` (instead of `my-thing`).

Asking

You can provide default values for variables asked to the user during the command.

Here’s an example demonstrating all the properties and their default values, as well as the section to put all of this into.

Version 0.0.0
Last updated 2020-03-08 13:09:39 +0100