Package 'potions'

July 23, 2025

Type Package

Title Easy Options Management

Version 0.2.0

Description Store and retrieve data from options() using syntax derived from the 'here' package. 'potions' makes it straightforward to update and retrieve options, either in the workspace or during package development, without overwriting global options.

Depends R (>= 4.1.0)

Imports jsonlite, lobstr, purrr, rrapply, rlang, stringi, yaml

Suggests knitr, pkgload, rmarkdown, testthat (>= 3.0.0)

License MPL-2.0

URL `https://potions.ala.org.au`

BugReports `https://github.com/AtlasOfLivingAustralia/potions/issues`

Encoding UTF-8

RoxygenNote 7.2.3

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation no

Author Martin Westgate [aut, cre]

Maintainer Martin Westgate <`martin.westgate@csiro.au`>

Repository CRAN

Date/Publication 2023-08-23 04:20:02 UTC

Contents

`brew` *Set up potions for easy data retrieval*

Description

Function to place a list into `options()`, or to update previously-stored data.

Usage

```
brew(
  ...,
  file,
  .slot,
  .pkg,
  method = c("modify", "merge", "overwrite", "leaves")
)

brew_package(..., file, .pkg, method)

brew_interactive(..., file, .slot, method)
```

Arguments

`...`	One or named arguments giving attributes to be stored; or alternatively a `list` containing the same.
`file`	string: optional file containing data to be stored via `options()`. Valid formats are `.yml` or `.json`.
`.slot`	string: optional name to mandate where data is stored. Defaults to a random string generated by `stringi::stri_rand_strings()`.
`.pkg`	string: package name that `potions` is being used within. Typically only used during `onLoad()`, after which later calls do not require this argument to be set.
`method`	string: How should new data be written to `options()`? See details for specifics.

Details

The default method is to use `brew` without setting either `.pkg` or `.slot` (but not both), and letting `potions` determine which slot to use. If greater control is needed, you can use `brew_package()` or `brew_interactive()`. Note that if neither `.slot` or `.pkg` are set, `potions` defaults to `.slot` , unless `.pkg` information has previously been supplied (and `.slot` information has not). This might be undesirable in a package development situation.

If both `...` and `file` arguments are empty, this function sets up an empty `potions` object in `options("potions-pkg")`; See `potions-class` for more information on this data type. If `...` and `file` arguments are provided, they will be amalgamated using `purrr::list_modify()`. If there are identical names in both lists, those in `...` are chosen.

If the user repeatedly calls `brew()`, later list entries overwrite early entries. Whole lists are not overwritten unless all top-level entry names match, or `method` is set to `"overwrite"`, which is a shortcut to using `drain()` before `brew()`. The default behaviour is `method = "modify"`, which uses `purrr::list_modify()` to do the joining. Similarly `"merge"` uses `purrr::list_merge()`. `method = "leaves"` only overwrites terminal nodes, leaving the structure of the list otherwise unaffected. For non-nested lists, this behaviour is identical to `"modify"`, but for nested lists it can be a useful shortcut.

Value

This function never returns an object; it is called for its' side- effect of caching data using `options()`.

Examples

```
# basic usage is to pass arguments using `=`
brew(x = 1)

# lists are also permitted
list(x = 2) |> brew()

# as are passing lists as objects
my_list <- list(x = 3)
my_list |> brew()

# or within a function
my_fun <- function(){list(x = 1, y = 2)}
my_fun() |> brew()

# optional clean-up
drain()
```

`drain` *Clear package options*

Description

Clear options of previously specified content. In most cases, calling `drain` with no arguments will be sufficient, but the arguments `.slot` and `.pkg`, and their corresponding functions `drain_interactive()` and `drain_package()` are provided in case greater control is needed. This is rarely needed for packages, but it is possible to manually specify the use of multiple slots when using `potions::brew()` interactively.

Usage

```
drain(.slot, .pkg)

drain_package(.pkg)

drain_interactive(.slot)
```

Arguments

`.slot`	(optional) slot to clear from `options()`
`.pkg`	(optional) package to clear from `options()`

Details

Note that this function is not vectorized, so passing multiple values to `.slot` or `.pkg` will fail (e.g. `drain(.slot = c("x", "y"))`). Similarly, passing arguments to both `.slot` and `.pkg` will fail.

Value

This function never returns an object; it is called for its' side- effect of removing data from `options()`.

`potions-class` *Methods for* `potions` *data*

Description

This package stores data in a list-like format, named class `potions`. It contains three entries: `slots` contains data stored in 'interactive' mode; `packages` contains data from packages built using `potions`; and `mapping` stores data to understand the contents of the other two slots.

Usage

```
create_potions()

## S3 method for class 'potions'
print(x, ...)
```

Arguments

`x`	An object of class `potions`
`...`	Any further arguments to `print()`

Value

In the case of `create_potions()`, an empty `potions` object. `print.potions()` displays a `potions` object using `lobstr::tree()`.

pour — *Retrieve information stored using* `potions::brew()`

Description

This is the main function that most users will call on. It retrieves data from a `potions` object stored using `brew()`. The UI for this function is based on the `here` package, in that it uses list names separated by commas to navigate through nested content. It differs from `here` in not requiring those names to be quoted.

Usage

```
pour(..., .slot, .pkg)

pour_package(..., .pkg)

pour_interactive(..., .slot)

pour_all()
```

Arguments

`...`	string: what slots should be returned
`.slot`	string: Optional manual override to default slot
`.pkg`	string: Optional manual override to default package

Details

Providing multiple arguments to `...` brings back nested values, i.e. `pour("x", "y")` is for the case of an object structured as `list(x = list(y = 1))`, rather than `list(x = 1, y = 2)`. For the latter case it would be necessary to call with either no arguments (`unlist(pour())`), or for greater control, call pour multiple times specifying different entries each time (e.g. `z <- c(pour("x"), pour("y"))`).

Additional functions are provided in case greater specificity is required. `pour_interactive(.slot = ...)` is synonymous with `pour(.slot = ...)`, while `pour_package(.pkg = ...)` is synonymous with `pour(.pkg = ...)`. `pour_all()` is a shortcut for `getOption("potions-pkg")`; i.e. to show all data stored using `potions` by any package or slot, and does not accept any arguments.

Value

If no arguments are passed to `...`, returns a `list` from the default slot. If `...` is supplied (correctly), then returns a `vector` of values matching those names.

Examples

```
# first import some data
brew(x = 1, y = list(a = 2, b = 3))

# get all data
pour()

# get only data from slot x
pour("x")

# get nested data
pour("y", "a")

# optional clean-up
drain()
```

---

read_config *Handle configuration data from a file*

Description

This is primarily an internal function for importing configuration information from a file. It is called by `brew()`, and detects `.yml` or `.json` files by their file extentions; all the actual work is done by `yaml::read_yaml` and `jsonlite::read_json` respectively. It is available as an exported function so that users can check their data is being imported correctly, and for developers who may wish to intercept configuration files for checking purposes.

Usage

```
read_config(file)
```

Arguments

`file`	string: path to file. Readable formats are `.yml` and `.json`.

Value

A `list` containing data from the specified file.

Index