Package: SpaDES.project (via r-universe)

October 1, 2024

Type Package

Title Project Templates Using 'SpaDES'

Description Quickly setup a 'SpaDES' project directories and add modules using templates.

URL https://spades-project.predictiveecology.org/, https://github.com/PredictiveEcology/SpaDES.project

Date 2024-08-02

Version 0.1.0.9008

Depends R (>= 4.2)

Imports data.table, fs, methods, Require (>= 0.3.1.9082), rprojroot, rstudioapi, tools, utils

Suggests covr, crayon, digest, ellipsis, filelock, geodata, gert, gh, gitcreds, googledrive, httr, igraph, knitr, pkgload, raster, remotes, reproducible, rmarkdown, sf, SpaDES.config, SpaDES.core, terra, testthat (>= 3.0.0), usethis, visNetwork, waldo, withr

Remotes PredictiveEcology/SpaDES.config@development, PredictiveEcology/SpaDES.core@development, rspatial/geodata

Encoding UTF-8

Language en-CA

License GPL-3

VignetteBuilder knitr, rmarkdown

BugReports https://github.com/PredictiveEcology/SpaDES.project/issues

ByteCompile yes

RoxygenNote 7.3.2

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

Collate 'SpaDES.projectOptions.R' 'environment.R' 'fileEdit.R' 'imports.R' 'getModule.R' 'helpers.R' 'listModules.R' 'makeDESCRIPTION.R' 'packages.R' 'paths2.R' 'pkgload2.R' 'setupProject.R' 'spades-project-package.R' 'txt.R' 'zzz.R'

Repository https://predictiveecology.r-universe.dev

RemoteUrl https://github.com/PredictiveEcology/SpaDES.project

RemoteRef development

RemoteSha c94074508fc9a3ea28459f7ffe666659c83d1815

Contents

`SpaDES.project-package` *Project templates using* `SpaDES`

Description

Quickly setup 'SpaDES' project directories and add modules using templates.

Author(s)

Maintainer: Eliot J B McIntire `<eliot.mcintire@nrcan-rncan.gc.ca>` (ORCID)

Other contributors:

- Alex M Chubaty `<achubaty@for-cast.ca>` (ORCID) [contributor]
- Ian Eddy `<ian.eddy@nrcan-rncan.gc.ca>` (ORCID) [contributor]
- Ceres Barros `<ceres.barros@nrcan-rncan.gc.ca>` [contributor]

See Also

Useful links:

- `https://spades-project.predictiveecology.org/`
- `https://github.com/PredictiveEcology/SpaDES.project`
- Report bugs at `https://github.com/PredictiveEcology/SpaDES.project/issues`

`.libPathDefault` *SpaDES.project default .libPaths() directory*

Description

For a given `name`, this will return the default library for packages.

Usage

```
.libPathDefault(name)
```

Arguments

`name`	A text string. When used in `setupProject`, this is the `projectName`

Value

A path where the packages will be installed.

`.teardownProject` *Helpers for cleanup of global state in examples and tests*

Description

1. remove project library directory created using `setupProject()`;
2. remove project paths created using `setupProject`;
3. restore original library paths.

Usage

```
.teardownProject(prjPaths, origLibPaths)
```

Arguments

`prjPaths`	character vector of paths to be removed
`origLibPaths`	character string giving the original library path to be restored

Value

NULL. Invoked for its side effects.

Note

not intended to be called by users

`findProjectPath` *Find the project root directory*

Description

Searches from current working directory for and Rstudio project file or git repository, falling back on using the current working directory.

Usage

```
findProjectPath()

findProjectName()
```

Value

`findProjectPath` returns an absolute path; `findProjectName` returns the basename of the path.

getGithubFile — *A simple way to get a Github file, authenticated*

Description

This can be used within e.g., the `options` or `params` arguments for `setupProject` to get a ready-made file for a project.

Usage

```
getGithubFile(
  gitRepoFile,
  overwrite = FALSE,
  destDir = ".",
  verbose = getOption("Require.verbose")
)
```

Arguments

`gitRepoFile`	Character string that follows the convention *GitAccount/GitRepo@Branch/File*, if @Branch is omitted, then it will be assumed to be `master` or `main`.
`overwrite`	A logical vector of same length (or length 1) `gitRepo`. If `TRUE`, then the download will delete any existing folder with the same name as the `repository` provided in `gitRepo`
`destDir`	A directory to put the file that is to be downloaded.
`verbose`	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if `1` or TRUE, more outputs; `2` even more. NOTE: in `Require` function, when `verbose >= 2`, also returns details as if `returnDetails = TRUE` (for backwards compatibility).

See Also

getModule

Examples

```
filename <- getGithubFile("PredictiveEcology/LandWeb@development/01b-options.R",
                          destDir = Require::tempdir2())
```

getModule *Simple function to download a SpaDES module as GitHub repository*

Description

Simple function to download a SpaDES module as GitHub repository

Usage

```
getModule(
  modules,
  modulePath,
  overwrite = FALSE,
  verbose = getOption("Require.verbose", 1L)
)
```

Arguments

`modules`	Character vector of one or more github repositories as character strings that contain SpaDES modules. These should be presented in the standard R way, with `account/repository@branch`. If `account` is omitted, then `"PredictiveEcology` will be assumed.
`modulePath`	A local path in which to place the full module, within a subfolder ... i.e., the source code will be downloaded to here: `file.path(modulePath, repository)`. If omitted, and `options(spades.modulePath)` is set, it will use `getOption("spades.modulePath")`, otherwise it will use `"."`.
`overwrite`	A logical vector of same length (or length 1) `gitRepo`. If `TRUE`, then the download will delete any existing folder with the same name as the `repository` provided in `gitRepo`
`verbose`	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in `Require` function, when `verbose` >= 2, also returns details as if `returnDetails` = `TRUE` (for backwards compatibility).

See Also

getGithubFile

`listModules` *Tools for examining modules on known repositories*

Description

When exploring existing modules, these tools help identify and navigate modules and their interdependencies.

Usage

```
listModules(
  keywords,
  accounts,
  includeForks = FALSE,
  includeArchived = FALSE,
  excludeStale = TRUE,
  omit = c("fireSense_dataPrepFitRas"),
  purge = FALSE,
  returnList = FALSE,
  verbose = getOption("Require.verbose", 1L)
)

moduleDependencies(
  modules,
  modulePath = getOption("reproducible.modulePath", ".")
)

moduleDependenciesToGraph(md)

PlotModuleGraph(graph)
```

Arguments

`keywords`	A vector of character strings that will be used as keywords for identify modules
`accounts`	A vector of character strings identifying GitHub accounts e.g., `PredictiveEcology` to search.
`includeForks`	Should the returned list include repositories that are forks (i.e., not the original repository). Default is `FALSE`.
`includeArchived`	Should the returned list include repositories that are archived (i.e., developer has retired them). Default is `FALSE`.
`excludeStale`	Logical or date. If `TRUE`, then only repositories that are still active (commits in the past 2 years) are returned. If a date (e.g., "2021-01-01"), then only repositories with commits since that date are returned. Default is `TRUE`, i.e., only include active in past 2 years.
`omit`	A vector of character strings of repositories to ignore.

`purge`	There is some internal caching that occurs. Setting this to `TRUE` will remove any cached data that is part of the requested `accounts` and `keywords`.
`returnList`	Should the function return a named list where the name is the `account` and the elements are the `repositories` selected. Default `FALSE`, i.e., return a character vector. This is included to allow a user to maintain backwards compatibility by setting `returnList = TRUE`
`verbose`	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if `1` or TRUE, more outputs; `2` even more. NOTE: in `Require` function, when `verbose >= 2`, also returns details as if `returnDetails = TRUE` (for backwards compatibility).
`modules`	Either a character vector of local module names, or a named list of character strings of short module names (i.e., the folder paths in `modulePath`).
`modulePath`	A character string indicating the path where the modules are located.
`md`	A data.table with columns `from` and `to`, showing relationships of objects in modules. Likely from `moduleDependencies`.
`graph`	An igraph object to plot. Likely returned by `moduleDependenciesToGraph`.

Value

`listModules` returns a character vector of paste0(account, "/", Repository) for all SpaDES modules in the given repositories with the `accounts` and `keywords` provided.

See Also

`metadataInModules()` helps to see different metadata elements in a folder of modules.

Examples

```
listModules(accounts = "PredictiveEcology", "none")
```

---

makeDESCRIPTIONproject *Make DESCRIPTION file(s) from SpaDES module metadata*

Description

Make DESCRIPTION file(s) from SpaDES module metadata

Usage

```
makeDESCRIPTIONproject(
  modules,
  modulePath,
  projectPath = ".",
  singleDESCRIPTION = TRUE,
  package = "Project",
  title = "Project",
  description = "Project",
  version = "1.0.0",
  authors = Sys.info()["user"],
  write = TRUE,
  verbose = getOption("Require.verbose")
)

makeDESCRIPTION(
  modules,
  modulePath,
  projectPath = ".",
  singleDESCRIPTION = FALSE,
  package,
  title,
  date,
  description,
  version,
  authors,
  write = TRUE,
  verbose,
  metadataList,
  ...
)
```

Arguments

`modules`	A character vector of module names
`modulePath`	Character. The path with modules, usually `modulePath()` or `paths$modulePath`
`projectPath`	Character. Only used if `singleDESCRIPTION` = `TRUE`
`singleDESCRIPTION`	Logical. If `TRUE`, there be only one DESCRIPTION file written for all modules, i.e., all reqdPkgs will be trimmed for redundancies and put into the single project-level DESCRIPTION file.
`package`	The name inserted into the "Package" entry in DESCRIPTION
`title`	The string inserted into the "Title" entry in DESCRIPTION
`description`	The string inserted into the "Description" entry in DESCRIPTION
`version`	The string inserted into the "Version" entry in DESCRIPTION
`authors`	The string inserted into the "Authors" entry in DESCRIPTION

`write`	Logical. If `TRUE`, then it will write the DESCRIPTION file either in the `modulePath` (if `singleDESCRIPTION = FALSE`) or `projectPath` (if `singleDESCRIPTION = TRUE`)
`verbose`	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in `Require` function, when `verbose >= 2`, also returns details as if `returnDetails = TRUE` (for backwards compatibility).
`date`	Date to enter into DESCRIPTION file. Defaults to `Sys.Date()`
`metadataList`	The parsed source code from a module. Must include `defineModule` metadata.
`...`	Currently not used.

`packagesInModules` *Extract element from SpaDES module metadata*

Description

Parses module code, looking for the `metadataItem` (default = `"reqdPkgs"`) element in the `defineModule` function.

Usage

```
packagesInModules(modules, modulePath = getOption("spades.modulePath"))

metadataInModules(
  modules,
  metadataItem = "reqdPkgs",
  modulePath = getOption("spades.modulePath"),
  needUnlist,
  verbose = getOption("Require.verbose", 1L)
)
```

Arguments

`modules`	character vector of module names
`modulePath`	path to directory containing the module(s) named in `modules`
`metadataItem`	character identifying the metadata field to extract
`needUnlist`	logical indicating whether to `unlist` the resulting metadata look up
`verbose`	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if 1 or TRUE, more outputs; 2 even more. NOTE: in `Require` function, when `verbose >= 2`, also returns details as if `returnDetails = TRUE` (for backwards compatibility).

Value

A character vector of sorted, unique packages that are identified in all named modules, or if `modules` is omitted, then all modules in `modulePath`.

---

pkgload2 *An alternative to* `pkgload::load_all` *with caching*

Description

`pkgload::load_all` does not automatically deal with dependency chains: the user must manually load the dependency chain in order with separate calls to `pkgload::load_all`. Also, it does not use caching. This function allows nested caching for a sequence of packages that depend on one another. For example, if a user has 3 packages that have dependency chain: A is a dependency of B which is a dependency of C. If a change happens in C, then pkgload::load_all will only be called on C. If a change happens in A, then pkgload::load_all will be called on A, then B, then C.

Usage

```
pkgload2(
  depsPaths = file.path("~/GitHub", c("reproducible", "SpaDES.core", "LandR")),
  envir = parent.frame()
)
```

Arguments

`depsPaths`	A character vector of paths to packages that need loading, or list of these. Each vector should be the load order sequence, based on the package dependencies, i.e., the first element in the vector should be a dependency of the second element in the vector etc. For packages that do not depend on each other, use separate list elements.
`envir`	An environment where an object called .prevDigs that will be placed and used as a cache comparison.

Value

This is called for its side effects, which are 2: pkgload::load_all on the packages that need it, and an object, `.prevDigs` that is assigned to `envir`.

setProjPkgDir — *Set the package directory for a project*

Description

This function will create a sub-folder of the `lib.loc` directory that is based on the R version and the platform, as per the standard R package directory naming convention

Usage

```
setProjPkgDir(lib.loc = "packages", verbose = getOption("Require.verbose", 1L))
```

Arguments

`lib.loc`	The folder for installing packages inside of
`verbose`	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if `1` or TRUE, more outputs; `2` even more. NOTE: in `Require` function, when `verbose >= 2`, also returns details as if `returnDetails = TRUE` (for backwards compatibility).

setupPaths — *Individual* `setup*` *functions that are contained within* `setupProject`

Description

These functions will allow more user control, though in most circumstances, it should be unnecessary to call them directly.

Usage

```
setupPaths(
  name,
  paths,
  inProject,
  standAlone = TRUE,
  libPaths = NULL,
  updateRprofile = getOption("SpaDES.project.updateRprofile", TRUE),
  Restart = getOption("SpaDES.project.Restart", FALSE),
  overwrite = FALSE,
  envir = parent.frame(),
  useGit = getOption("SpaDES.project.useGit", FALSE),
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  ...
)

setupFunctions(
  functions,
  name,
  sideEffects,
  paths,
  overwrite = FALSE,
  envir = parent.frame(),
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  ...
)

setupSideEffects(
  name,
  sideEffects,
  paths,
  times,
  overwrite = FALSE,
  envir = parent.frame(),
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  ...
)

setupOptions(
  name,
  options,
  paths,
  times,
  overwrite = FALSE,
  envir = parent.frame(),
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  useGit = getOption("SpaDES.project.useGit", FALSE),
  updateRprofile = getOption("SpaDES.project.updateRprofile", TRUE),
  ...
)

setupModules(
  name,
  paths,
  modules,
  inProject,
  useGit = getOption("SpaDES.project.useGit", FALSE),
  overwrite = FALSE,
  envir = parent.frame(),
  gitUserName,
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  updateRprofile = getOption("SpaDES.project.updateRprofile", TRUE),
  ...
)

setupPackages(
  packages,
  modulePackages = list(),
  require = list(),
  paths,
  libPaths,
  setLinuxBinaryRepo = TRUE,
  standAlone,
  envir = parent.frame(),
  verbose = getOption("Require.verbose"),
  dots,
  defaultDots,
  ...
)

setupParams(
  name,
  params,
  paths,
  modules,
  times,
  options,
  overwrite = FALSE,
  envir = parent.frame(),
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  ...
)

setupGitIgnore(
  paths,
  gitignore = getOption("SpaDES.project.gitignore", TRUE),
  verbose
)
```

```
setupStudyArea(
  studyArea,
  paths,
  envir,
  verbose = getOption("Require.verbose", 1L)
)

setupFiles(
  files,
  paths,
  envir = parent.frame(),
  verbose = getOption("Require.verbose", 1L)
)
```

Arguments

`name`	Optional. If supplied, the name of the project. If not supplied, an attempt will be made to extract the name from the `paths[["projectPath"]]`. If this is a GitHub project, then it should indicate the full Github repository and branch name, e.g., `"PredictiveEcology/WBI_forecasts@ChubatyPubNum12"`
`paths`	a list with named elements, specifically, `modulePath`, `projectPath`, `packagePath` and all others that are in `SpaDES.core::setPaths()` (i.e., `inputPath`, `outputPath`, `scratchPath`, `cachePath`, `rasterTmpDir`). Each of these has a sensible default, which will be overridden but any user supplied values. See setup.
`inProject`	A logical. If `TRUE`, then the current directory is inside the `paths[["projectPath"]]`.
`standAlone`	A logical. Passed to `Require::standAlone`. This keeps all packages installed in a project-level library, if `TRUE`. Default is `TRUE`.
`libPaths`	Deprecated. Use `paths = list(packagePath = ...)`.
`updateRprofile`	Logical. Should the `paths$packagePath` be set in the `.Rprofile` file for this project. Note: if `paths$packagePath` is within the `tempdir()`, then there will be a warning, indicating this won't persist. If the user is using `Rstudio` and the `paths$projectPath` is not the root of the current Rstudio project, then a warning will be given, indicating the .Rprofile may not be read upon restart.
`Restart`	Logical or character. If either `TRUE` or a character, and if the `projectPath` is not the current path, and the session is in RStudio and interactive, it will try to restart Rstudio in the projectPath with a new Rstudio project. If character, it should represent the filename of the script that contains the `setupProject` call that should be copied to the new folder and opened. If `TRUE`, it will use the active file as the one that should be copied to the new projectPath and opened in the Rstudio project. If successful, this will create an RStudio Project file (and .Rproj.user folder), restart with a new Rstudio session with that new project and with a root path (i.e. working directory) set to `projectPath`. Default is `FALSE`, and no RStudio Project is created.
`overwrite`	Logical vector or character vector, however, only `getModule` will respond to a vector of values. If length-one `TRUE`, then all files that were previously downloaded will be overwritten throughout the sequence of `setupProject`. If a vector of logical or character, these will be passed to `getModule`: only the named

	modules will be overwritten or the logical vector of the modules. NOTE: if a vector, no other file specified anywhere in `setupProject` will be overwritten except a module that/those names, because only `setupModules` is currently responsive to a vector. To have fine grained control, a user can just manually delete a file, then rerun.
`envir`	An environment within which to look for objects. If called alone, the function should use its own internal environment. If called from another function, e.g., `setupProject`, then the `envir` should be the internal transient environment of that function.
`useGit`	(if not FALSE, then experimental still). There are two levels at which a project can use GitHub, either the `projectPath` and/or the `modules`. Any given project can have one or the other, or both of these under git control. If "both", then this function will assume that git submodules will be used for the `modules`. A logical or `"sub"` for *submodule*. If `"sub"`, then this function will attempt to clone the identified `modules` *as git submodules*. This will only work if the `projectPath` is a git repository. If the project is already a git repository because the user has set that up externally to this function call, then this function will add the `modules` as git submodules. If it is not already, it will use `git clone` for each module. After git clone or submodule add are run, it will run `git checkout` for the named branch and then `git pull` to get and change branch for each module, according to its specification in `modules`. If `FALSE`, this function will download modules with `getModules`. NOTE: *CREATING* A GIT REPOSITORY AT THE PROJECT LEVEL AND SETTING MODULES AS GIT SUBMODULES IS EXPERIMENTAL. IT IS FINE IF THE PROJECT HAS BEEN MANUALLY SET UP TO BE A GIT REPOSITORY WITH SUBMODULES: THIS FUNCTION WILL ONLY EVALUTE PATHS. This can be set with the `option(SpaDES.project.useGit = xxx)`.
`verbose`	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if `1` or TRUE, more outputs; `2` even more. NOTE: in `Require` function, when `verbose >= 2`, also returns details as if `returnDetails = TRUE` (for backwards compatibility).
`dots`	Any other named objects passed as a list a user might want for other elements.
`defaultDots`	A named list of any arbitrary R objects. These can be supplied to give default values to objects that are otherwise passed in with the `...`, i.e., not specifically named for these `setup*` functions. If named objects are supplied as top-level arguments, then the `defaultDots` will be overridden. This can be particularly useful if the arguments passed to `...` do not always exist, but rely on external e.g., batch processing to optionally fill them. See examples.
`...`	further named arguments that acts like `objects`, but a different way to specify them. These can be anything. The general use case is to create the `objects` that are would be passed to `SpaDES.core::simInit`, or `SpaDES.core::simInitAndSpades`, (e.g. `studyAreaName` or `objects`) or additional objects to be passed to the simulation (in older versions of `SpaDES.core`, these were passed as a named list to the `objects` argument). **Order matters**. These are sequentially evaluated, and also any arguments that are specified before the named arguments e.g., `name`, `paths`, will be evaluated prior to any of the named arguments, i.e., "at the start"

	of the `setupProject`. If placed after the first named argument, then they will be evaluated at the end of the `setupProject`, so can access all the packages, objects, etc.
`functions`	A set of function definitions to be used within `setupProject`. These will be returned as a list element. If function definitions require non-base packages, prefix the function call with the package e.g., `terra::rast`. When using `setupProject`, the `functions` argument is evaluated after `paths`, so it cannot be used to define functions that help specify `paths`.
`sideEffects`	Optional. This can be an expression or one or more file names or a code chunk surrounded by `{...}`. If a non-text file name is specified (e.g., *not .txt or .R* currently), these files will simply be downloaded, using their relative path as specified in the github notation. They will be downloaded or accessed locally at that relative path. If these file names represent scripts (*.txt or .R), this/these will be parsed and evaluated, but nothing is returned (i.e., any assigned objects are not returned). This is intended to be used for operations like cloud authentication or configuration functions that are run for their side effects only.
`times`	Optional. This will be returned if supplied; if supplied, the values can be used in e.g., `params`, e.g., `params = list(mod = list(startTime = times$start))`. See help for `SpaDES.core::simInit`.
`options`	Optional. Either a named list to be passed to `options` or a character vector indicating one or more file(s) to source, in the order provided. These will be parsed locally (not the `.GlobalEnv`), so they will not create globally accessible objects. NOTE: `options` is run 2x within `setupProject`, once before `setupPaths` and once after `setupPackages`. This occurs because many packages use options for their behaviour (need them set before e.g., `Require::require` is run; but many packages also change `options` at startup. See details. See setup.
`modules`	a character vector of modules to pass to `getModule`. These should be one of: simple name (e.g., `fireSense`) which will be searched for locally in the `paths[["modulePath"]]`; or a GitHub repo with branch (`GitHubAccount/Repo@branch` e.g., `"PredictiveEcology/Biomass_core@development"`); or a character vector that identifies one or more module folders (local or GitHub) (not the module .R script). If the entire project is a git repository, then it will not try to re-get these modules; instead it will rely on the user managing their git status outside of this function. See setup.
`gitUserName`	The GitHub account name. Used with git clone git@github.com:*gitHuserName*/name
`packages`	Optional. A vector of packages that must exist in the `libPaths`. This will be passed to `Require::Install`, i.e., these will be installed, but not attached to the search path. See also the `require` argument. To force skip of package installation (without assessing modules), set `packages = NULL`
`modulePackages`	A named list, where names are the module names, and the elements of the list are packages in a form that `Require::Require` accepts.
`require`	Optional. A character vector of packages to install *and* attach (with `Require::Require`). These will be installed and attached at the start of `setupProject` so that a user can use these during `setupProject`. See setup
`setLinuxBinaryRepo`	Logical. Should the binary RStudio Package Manager be used on Linux (ignored if Windows)

params	Optional. Similar to options, however, this named list will be returned, i.e., there are no side effects. See setup.
gitignore	Logical. Only has an effect if the paths$projectPath is a git repositories without submodules. This case is ambiguous what a user wants. If TRUE, the default, then paths$modulePath will be added to the .gitignore file. Can be controled with options(SpadES.project.gitignore = ...).
studyArea	Optional. If a list, it will be passed to geodata::gadm. To specify a country other than the default "CAN", the list must have a named element, "country". All other named elements will be passed to gadm. 2 additional named elements can be passed for convenience, subregion = "...", which will be grepped with the column NAME_1, and epsg = "...", so a user can pass an epsg.io code to reproject the studyArea. See examples.
files	A vector or list of files to parse. These can be remote github.com files.

Details

setPaths will fill in any paths that are not explicitly supplied by the user as a named list. These paths that can be set are: projectPath, packagePath, cachePath, inputPath, modulePath, outputPath, rasterPath, scratchPath, terraPath. These are grouped thematically into three groups of paths: projectPath and packagePath affect the project, regardless of whether a user uses SpaDES modules. cachePath, inputPath, outputPath and modulePath are all used by SpaDES within module contexts. scratchPath, rasterPath and terraPath are all "temporary" or "scratch" directories.

setupFunctions will source the functions supplied, with a parent environment being the internal temporary environment of the setupProject, i.e., they will have access to all the objects in the call.

Most arguments in the family of setup* functions are run *sequentially*, even within the argument. Since most arguments take lists, the user can set values at a first value of a list, then use it in calculation of the 2nd value and so on. See examples. This "sequential" evaluation occurs in the ..., setupSideEffects, setupOptions, setupParams (this does not work for setupPaths) can handle sequentially specified values, meaning a user can first create a list of default options, then a list of user-desired options that may or may not replace individual values. This can create hierarchies, *based on order*.

setupOptions can handle sequentially specified values, meaning a user can first create a list of default options, then a list of user-desired options that may or may not replace individual values. Thus final values will be based on the order that they are provided.

setupModules will download all modules do not yet exist locally. The current test for "exists locally" is simply whether the directory exists. If a user wants to update the module, overwrite = TRUE must be set, or else the user can remove the folder manually.

setupPackages will read the modules' metadata reqdPkgs element. It will combine these with any packages passed manually by the user to packages, and pass all these packages to Require::Install(...).

setupGitIgnore will add.

setupStudyArea only uses inputPath within its paths argument, which will be passed to path argument of gadm. User can pass any named list element that matches the columns in the sf object, including e.g., NAME_1 and, if level = 2, is specified, then NAME_2.

```
setupStudyArea(list(NAME_1 = "Alberta", "NAME_2" = "Division No. 17", level = 2))
```

`setupFiles` is a convenience function intended for interactive use to verify the files being parsed. This is similar to `parse`, but each element must be a named list or a named object, such as a function. It uses the same specification for `https://github.com` files as `setupProject`, i.e., using `@` for branch.

```
setupFiles("PredictiveEcology/PredictiveEcology.org@main/tutos/castorExample/params.R")
```

Value

`setupPaths` returns a list of paths that are created. `projectPath` will be assumed to be the base of other non-temporary and non-R-library paths. This means that all paths that are directly used by `simInit` are assumed to be relative to the `projectPath`. If a user chooses to specify absolute paths, then they will be returned as is. It is also called for its side effect which is to call `setPaths`, with each of these paths as an argument. See table for details. If a user supplies extra paths not useable by `SpaDES.core::simInit`, these will added as an attribute ("extraPaths") to the `paths` element in the returned object. These will still exist directly in the returned list if a user uses `setupPaths` directly, but these will not be returned with `setupProject` because `setupProject` is intended to be used with `SpaDES.core::simInit`. In addition, three paths will be added to this same attribute automatically: `projectPath`, `packagePath`, and `.prevLibPaths` which is the previous value for `.libPaths()` before changing to `packagePath`.

`setupFunctions` returns NULL. All functions will be placed in `envir`.

`setupSideEffects` is run for its side effects (e.g., web authentication, custom package options that cannot use `base::options`), with deliberately nothing returned to user. This, like other parts of this function, attempts to prevent unwanted outcomes that occur when a user uses e.g., `source` without being very careful about what and where the objects are sourced to.

`setupOptions` is run for its side effects, namely, changes to the `options()`. The list of modified options will be added as an attribute (`attr(out, "projectOptions")`), e.g., so they can be "unset" by user later.

`setupModules` is run for its side effects, i.e., downloads modules and puts them into the `paths[["modulePath"]]`. It will return a named list, where the names are the full module names and the list elemen.ts are the R packages that the module depends on (`reqsPkgs`)

`setupPackages` is run for its side effects, i.e., installing packages to `paths[["packagePath"]]`.

`setupParams` prepares a named list of named lists, suitable to be passed to the `params` argument of `simInit`.

`setupGitIgnore` is run for its side effects, i.e., adding either `paths$packagePath` and/or `paths$modulePath` to the `.gitignore` file. It will check whether `packagePath` is located inside the `paths$projectPath` and will add this folder to the `.gitignore` if `TRUE`. If the project is a git repository with git submodules, then it will add nothing else. If the project is a git repository without git submodules, then the `paths$modulePath` will be added to the `.gitignore` file. It is assumed that these modules are used in a `read only` manner.

`setupStudyArea` will return an `sf` class object coming from `geodata::gadm`, with subregion specification as described in the `studyArea` argument.fsu

`setupFiles` a named list with each element that was parsed.

Paths

Path	Default if not supplied by user
	Project Level Paths
`projectPath`	if `getwd()` is name, then just `getwd`; if not `file.path(getwd(), name)`
`packagePath`	`file.path(tools::R_user_dir("data"), name, "packages", version$platform, substr(getRversio`
———	————
	Module Level Paths
`cachePath`	`file.path(projectPath, "cache")`
`inputPath`	`file.path(projectPath, "inputs")`
`modulePath`	`file.path(projectPath, "modules")`
`outputPath`	`file.path(projectPath, "outputs")`
———	————
	Temporary Paths
`scratchPath`	`file.path(tempdir(), name)`
`rasterPath`	`file.path(scratchPath, "raster")`
`terraPath`	`file.path(scratchPath, "terra")`
———	————
	Other Paths
`logPath`	`file.path(outputPath(sim), "log")`
`tilePath`	Not implemented yet

Examples

```
 ## simplest case; just creates folders
out <- setupProject(
  paths = list(projectPath = ".") #
)
# specifying functions argument, with a local file and a definition here
tf <- tempfile(fileext = ".R")
fnDefs <- c("fn <- function(x) x\n",
            "fn2 <- function(x) x\n",
            "fn3 <- function(x) terra::rast(x)")
cat(text = fnDefs, file = tf)
funHere <- function(y) y
out <- setupProject(functions = list(a = function(x) return(x),
                                     tf,
                                     funHere = funHere), # have to name it
                    # now use the functions when creating objects
                    drr = 1,
                    b = a(drr),
                    q = funHere(22),
                    ddd = fn3(terra::ext(0,b,0,b)))
```

setupProject *Sets up a new or existing SpaDES project*

Description

`setupProject` calls a sequence of functions in this order: `setupOptions` (first time), `setupPaths`, `setupRestart`, `setupFunctions`, `setupModules`, `setupPackages`, `setupSideEffects`, `setupOptions` (second time), `setupParams`, and `setupGitIgnore`.

This sequence will create folder structures, install missing packages from those listed in either the `packages`, `require` arguments or in the modules `reqdPkgs` fields, load packages (only those in the `require` argument), set options, download or confirm the existence of modules. It will also return elements that can be passed directly to `simInit` or `simInitAndSpades`, specifically, `modules`, `params`, `paths`, `times`, and any named elements passed to `...`. This function will also , if desired, change the .Rprofile file for this project so that every time the project is opened, it has a specific `.libPaths()`.

There are a number of convenience elements described in the section below. See Details. Because of this sequence, users can take advantage of settings (i.e., objects) that happen (are created) before others. For example, users can set `paths` then use the `paths` list to set `options` that will can update/change `paths`, or set `times` and use the `times` list for certain entries in `params`.

Usage

```
setupProject(
  name,
  paths,
  modules,
  packages,
  times,
  options,
  params,
  sideEffects,
  functions,
  config,
  require = NULL,
  studyArea = NULL,
  Restart = getOption("SpaDES.project.Restart"),
  useGit = getOption("SpaDES.project.useGit"),
  setLinuxBinaryRepo = getOption("SpaDES.project.setLinuxBinaryRepo"),
  standAlone = getOption("SpaDES.project.standAlone"),
  libPaths = NULL,
  updateRprofile = getOption("SpaDES.project.updateRprofile"),
  overwrite = getOption("SpaDES.project.overwrite"),
  verbose = getOption("Require.verbose", 1L),
  defaultDots,
  envir = parent.frame(),
  dots,
  ...
)
```

Arguments

`name`	Optional. If supplied, the name of the project. If not supplied, an attempt will be made to extract the name from the `paths[["projectPath"]]`. If this is a GitHub project, then it should indicate the full Github repository and branch name, e.g., `"PredictiveEcology/WBI_forecasts@ChubatyPubNum12"`
`paths`	a list with named elements, specifically, `modulePath`, `projectPath`, `packagePath` and all others that are in `SpaDES.core::setPaths()` (i.e., `inputPath`, `outputPath`, `scratchPath`, `cachePath`, `rasterTmpDir`). Each of these has a sensible default, which will be overridden but any user supplied values. See setup.
`modules`	a character vector of modules to pass to `getModule`. These should be one of: simple name (e.g., `fireSense`) which will be searched for locally in the `paths[["modulePath"]]`; or a GitHub repo with branch (`GitHubAccount/Repo@branch` e.g., `"PredictiveEcology/Biomass_core@development"`); or a character vector that identifies one or more module folders (local or GitHub) (not the module .R script). If the entire project is a git repository, then it will not try to re-get these modules; instead it will rely on the user managing their git status outside of this function. See setup.
`packages`	Optional. A vector of packages that must exist in the `libPaths`. This will be passed to `Require::Install`, i.e., these will be installed, but not attached to the search path. See also the `require` argument. To force skip of package installation (without assessing modules), set `packages = NULL`
`times`	Optional. This will be returned if supplied; if supplied, the values can be used in e.g., `params`, e.g., `params = list(mod = list(startTime = times$start))`. See help for `SpaDES.core::simInit`.
`options`	Optional. Either a named list to be passed to `options` or a character vector indicating one or more file(s) to source, in the order provided. These will be parsed locally (not the `.GlobalEnv`), so they will not create globally accessible objects. NOTE: `options` is run 2x within `setupProject`, once before `setupPaths` and once after `setupPackages`. This occurs because many packages use options for their behaviour (need them set before e.g., `Require::require` is run; but many packages also change `options` at startup. See details. See setup.
`params`	Optional. Similar to `options`, however, this named list will be returned, i.e., there are no side effects. See setup.
`sideEffects`	Optional. This can be an expression or one or more file names or a code chunk surrounded by `{...}`. If a non-text file name is specified (e.g., *not .txt or .R* currently), these files will simply be downloaded, using their relative path as specified in the github notation. They will be downloaded or accessed locally at that relative path. If these file names represent scripts (*.txt or .R), this/these will be parsed and evaluated, but nothing is returned (i.e., any assigned objects are not returned). This is intended to be used for operations like cloud authentication or configuration functions that are run for their side effects only.

`functions`	A set of function definitions to be used within `setupProject`. These will be returned as a list element. If function definitions require non-base packages, prefix the function call with the package e.g., `terra::rast`. When using `setupProject`, the `functions` argument is evaluated after `paths`, so it cannot be used to define functions that help specify `paths`.
`config`	Still experimental linkage to the `SpaDES.config` package. Currently not working.
`require`	Optional. A character vector of packages to install *and* attach (with `Require::Require`). These will be installed and attached at the start of `setupProject` so that a user can use these during `setupProject`. See setup
`studyArea`	Optional. If a list, it will be passed to `geodata::gadm`. To specify a country other than the default `"CAN"`, the list must have a named element, `"country"`. All other named elements will be passed to `gadm`. 2 additional named elements can be passed for convenience, `subregion = "..."`, which will be grepped with the column `NAME_1`, and `epsg = "..."`, so a user can pass an `epsg.io` code to reproject the `studyArea`. See examples.
`Restart`	Logical or character. If either `TRUE` or a character, and if the `projectPath` is not the current path, and the session is in RStudio and interactive, it will try to restart Rstudio in the projectPath with a new Rstudio project. If character, it should represent the filename of the script that contains the `setupProject` call that should be copied to the new folder and opened. If `TRUE`, it will use the active file as the one that should be copied to the new projectPath and opened in the Rstudio project. If successful, this will create an RStudio Project file (and .Rproj.user folder), restart with a new Rstudio session with that new project and with a root path (i.e. working directory) set to `projectPath`. Default is `FALSE`, and no RStudio Project is created.
`useGit`	(if not FALSE, then experimental still). There are two levels at which a project can use GitHub, either the `projectPath` and/or the `modules`. Any given project can have one or the other, or both of these under git control. If "both", then this function will assume that git submodules will be used for the `modules`. A logical or `"sub"` for *submodule*. If `"sub"`, then this function will attempt to clone the identified `modules` *as git submodules*. This will only work if the `projectPath` is a git repository. If the project is already a git repository because the user has set that up externally to this function call, then this function will add the `modules` as git submodules. If it is not already, it will use `git clone` for each module. After git clone or submodule add are run, it will run `git checkout` for the named branch and then `git pull` to get and change branch for each module, according to its specification in `modules`. If `FALSE`, this function will download modules with `getModules`. NOTE: *CREATING* A GIT REPOSITORY AT THE PROJECT LEVEL AND SETTING MODULES AS GIT SUBMODULES IS EXPERIMENTAL. IT IS FINE IF THE PROJECT HAS BEEN MANUALLY SET UP TO BE A GIT REPOSITORY WITH SUBMODULES: THIS FUNCTION WILL ONLY EVALUTE PATHS. This can be set with the `option(SpaDES.project.useGit = xxx)`.
`setLinuxBinaryRepo`	Logical. Should the binary RStudio Package Manager be used on Linux (ignored if Windows)

`standAlone`	A logical. Passed to `Require::standAlone`. This keeps all packages installed in a project-level library, if `TRUE`. Default is `TRUE`.
`libPaths`	Deprecated. Use `paths = list(packagePath = ...)`.
`updateRprofile`	Logical. Should the `paths$packagePath` be set in the `.Rprofile` file for this project. Note: if `paths$packagePath` is within the `tempdir()`, then there will be a warning, indicating this won't persist. If the user is using `Rstudio` and the `paths$projectPath` is not the root of the current Rstudio project, then a warning will be given, indicating the .Rprofile may not be read upon restart.
`overwrite`	Logical vector or character vector, however, only `getModule` will respond to a vector of values. If length-one `TRUE`, then all files that were previously downloaded will be overwritten throughout the sequence of `setupProject`. If a vector of logical or character, these will be passed to `getModule`: only the named modules will be overwritten or the logical vector of the modules. NOTE: if a vector, no other file specified anywhere in `setupProject` will be overwritten except a module that/those names, because only `setupModules` is currently responsive to a vector. To have fine grained control, a user can just manually delete a file, then rerun.
`verbose`	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or FALSE, then minimal outputs; if `1` or TRUE, more outputs; `2` even more. NOTE: in `Require` function, when `verbose >= 2`, also returns details as if `returnDetails = TRUE` (for backwards compatibility).
`defaultDots`	A named list of any arbitrary R objects. These can be supplied to give default values to objects that are otherwise passed in with the `...`, i.e., not specifically named for these `setup*` functions. If named objects are supplied as top-level arguments, then the `defaultDots` will be overridden. This can be particularly useful if the arguments passed to `...` do not always exist, but rely on external e.g., batch processing to optionally fill them. See examples.
`envir`	The environment where `setupProject` is called from. Defaults to `parent.frame()` which should be fine in most cases and user shouldn't need to set this
`dots`	Any other named objects passed as a list a user might want for other elements.
`...`	further named arguments that acts like `objects`, but a different way to specify them. These can be anything. The general use case is to create the `objects` that are would be passed to `SpaDES.core::simInit`, or `SpaDES.core::simInitAndSpades`, (e.g. `studyAreaName` or `objects`) or additional objects to be passed to the simulation (in older versions of `SpaDES.core`, these were passed as a named list to the `objects` argument). **Order matters**. These are sequentially evaluated, and also any arguments that are specified before the named arguments e.g., `name`, `paths`, will be evaluated prior to any of the named arguments, i.e., "at the start" of the `setupProject`. If placed after the first named argument, then they will be evaluated at the end of the `setupProject`, so can access all the packages, objects, etc.

Value

`setupProject` will return a named list with elements `modules`, `paths`, `params`, and `times`. The goal of this list is to contain list elements that can be passed directly to `simInit`.

It will also append all elements passed by the user in the `...`. This list can be passed directly to `SpaDES.core::simInit()` or `SpaDES.core::simInitAndSpades()` using a `do.call()`. See example.

NOTE: both `projectPath` and `packagePath` will be omitted in the `paths` list as they are used to set current directory (found with `getwd()`) and `.libPaths()[1]`, but are not accepted by `simInit`. `setupPaths` will still return these two paths as its outputs are not expected to be passed directly to `simInit` (unlike `setupProject` outputs).

Faster runtime after project is set up

There are a number of checks that occur during `setupProject`. These take time, particularly after an R restart (there is some caching in RAM that occurs, but this will only speed things up if there is no restart of R). To get the "fastest", these options or settings will speed things up, at the expense of not being completely re-runnable. You can add one or more of these to the arguments. These will only be useful after a project is set up, i.e., `setupProject` and `SpaDES.core::simInit` has/have been run at least once to completion (so packages are installed).

```
options = c(
  reproducible.useMemoise = TRUE,               # For caching, use memory objects
  Require.cloneFrom = Sys.getenv("R_LIBS_USER"),# Use personal library as possible source of packages
  spades.useRequire = FALSE,              # Won't install packages/update versions
  spades.moduleCodeChecks = FALSE,          # moduleCodeChecks checks for metadata mismatches
  reproducible.inputPaths = "~/allData"),     # For sharing data files across projects
packages = NULL,                        # Prevents any packages installs with setupProject
useGit = FALSE                                       # Prevents checks using git
```

These will be set early in `setupProject`, so will affect the running of `setupProject`. If the user manually sets one of these in addition to setting these, the user options will override these. The remining causes of `setupProject` being "slow" will be loading the required packages.

These options/arguments can now be set all at once (with caution as these changes will affect how your script will be run) with `options(SpaDES.project.fast = TRUE)` or in the `options` argument.

Objective

The overarching objectives for these functions are:

1. To prepare what is needed for `simInit`.
2. To help a user eliminate virtually all assignments to the `.GlobalEnv`, as these create and encourage spaghetti code that becomes unreproducible as the project increases in complexity.
3. Be very simple for beginners, but powerful enough to expand to almost any needs of arbitrarily complex projects, using the same structure
4. Deal with the complexities of R package installation and loading when working with modules that may have been created by many users
5. Create a common SpaDES project structure, allowing easy transition from one project to another, regardless of complexity.

Convenience elements

Sequential evaluation: Throughout these functions, efforts have been made to implement sequential evaluation, within files and within lists. This means that a user can *use* the values from an upstream element in the list. For example, the following where projectPath is part of the list that will be assigned to the paths argument and it is then used in the subsequent list element is valid:

```
setupPaths(paths = list(projectPath = "here",
                    modulePath = file.path(paths[["projectPath"]], "modules")))
```

Because of such sequential evaluation, paths, options, and params files can be sequential lists that have impose a hierarchy specified by the order. For example, a user can first create a list of *default* options, then several lists of user-desired options behind an if (user("emcintir")) block that add new or override existing elements, followed by machine specific values, such as paths.

```
setupOptions(
  maxMemory <- 5e+9 # if (grepl("LandWeb", runName)) 5e+12 else 5e+9

 # Example -- Use any arbitrary object that can be passed in the `...` of `setupOptions`
  #  or `setupProject`
  if (.mode == "development") {
     list(test = 2)
  }
  if (machine("A127")) {
    list(test = 3)
  }
)
```

Values and/or files: The arguments, paths, options, and params, can all understand lists of named values, character vectors, or a mixture by using a list where named elements are values and unnamed elements are character strings/vectors. Any unnamed character string/vector will be treated as a file path. If that file path has an @ symbol, it will be assumed to be a file that exists on a GitHub repository in https://github.com. So a user can pass values, or pointers to remote and/or local paths that themselves have values.

The following will set an option as declared, plus read the local file (with relative path), plus download and read the cloud-hosted file.

```
setupProject(
   options = list(reproducible.useTerra = TRUE,
                  "inst/options.R",
                 "PredictiveEcology/SpaDES.project@transition/inst/options.R")
                 )
   )
```

This approach allows for an organic growth of complexity, e.g., a user begins with only named lists of values, but then as the number of values increases, it may be helpful to put some in an external file.

NOTE: if the GitHub repository is *private* the user *must* configure their GitHub token by setting the GITHUB_PAT environment variable – unfortunately, the usethis approach to setting the token will not work at this moment.

Specifying `paths`**,** `options`**,** `params`**:** If `paths`, `options`, and/or `params` are a character string or character vector (or part of an unnamed list element) the string(s) will be interpreted as files to parse. These files should contain R code that specifies *named lists*, where the names are one or more `paths`, `options`, or are module names, each with a named list of parameters for that named module. This last named list for `params` follows the convention used for the `params` argument in `simInit(..., params = )`.

These files can use `paths`, `times`, plus any previous list in the sequence of `params` or `options` specified. Any functions that are used must be available, e.g., prefixed `Require::normPath` if the package has not been loaded (as recommended).

If passing a file to `options`, it should **not set** `options()` explicitly; only create named lists. This enables options checking/validating to occur within `setupOptions` and `setupParams`. A simplest case would be a file with this: `opts <- list(reproducible.destinationPath = "~/destPath")`.

All named lists will be parsed into their own environment, and then will be sequentially evaluated (i.e., subsequent lists will have access to previous lists), with each named elements setting or replacing the previously named element of the same name, creating a single list. This final list will be assigned to, e.g., `options()` inside `setupOptions`.

Because each list is parsed separately, they to not need to be assigned objects; if they are, the object name can be any name, even if similar to another object's name used to built the same argument's (i.e. `paths`, `params`, `options`) final list. Hence, in an file to passed to `options`, instead of incrementing the list as:

```
a <- list(optA = 1)
b <- append(a, list(optB = 2))
c <- append(b, list(optC = 2.5))
d <- append(c, list(optD = 3))
```

one can do:

```
a <- list(optA = 1)
a <- list(optB = 2)
c <- list(optC = 2.5)
list(optD = 3)
```

NOTE: only atomics (i.e., character, numeric, etc.), named lists, or either of these that are protected by 1 level of "if" are parsed. This will not work, therefore, for other side-effect elements, like authenticating with a cloud service.

Several helper functions exist within `SpaDES.project` that may be useful, such as `user(...)`, `machine(...)`

Can hard code arguments that may be missing: To allow for batch submission, a user can specify code `argument = value` even if `value` is missing. This type of specification will not work in normal parsing of arguments, but it is designed to work here. In the next example, `.mode = .mode` can be specified, but if R cannot find `.mode` for the right hand side, it will just skip with no error. Thus a user can source a script with the following line from batch script where `.mode` is specified. When running this line without that batch script specification, then this will assign no value to `.mode`. We include `.nodes` which shows an example of passing a value that does exist. The non-existent `.mode` will be returned in the `out`, but as an unevaluated, captured list element.

```
.nodes <- 2
out <- setupProject(.mode = .mode,
                    .nodes = .nodes,
                    options = "inst/options.R"
                    )
```

See Also

setupPaths(), setupOptions(), setupPackages(), setupModules(), setupGitIgnore(). Also, helpful functions such as user(), machine(), node()

`vignette("i-getting-started", package = "SpaDES.project")`

Examples

```
## For more examples:
vignette("i-getting-started", package = "SpaDES.project")

library(SpaDES.project)



 ## simplest case; just creates folders
out <- setupProject(
  paths = list(projectPath = ".") #
)
```

---

spadesProjectOptions   SpaDES.project *options*

Description

These demonstrate default values for some options that can be set in SpaDES.project. To see defaults, run `spadesProjectOptions()`. See Details below.

Usage

```
spadesProjectOptions()
```

Details

Below are options that can be set with `options("spades.xxx" = newValue)`, where `xxx` is one of the values below, and `newValue` is a new value to give the option. Sometimes these options can be placed in the user's `.Rprofile` file so they persist between sessions.

The following options are used, using the prefix: `spades`

OPTION	*DEFAULT VALUE*	*DESCRIP*
`reproducible.cachePath`	NOTE: uses `reproducible`. Defaults is within projectPath, with subfolder "cache"	
`spades.inputPath`	Default is within projectPath, with subfolder "inputs"	
`spades.modulePath`	Default is within projectPath, with subfolder "modules"	

`spades.outputPath`	Default is within projectPath, with subfolder "outputs"
`spades.packagePath`	Default to `.libPathDefault(<projectPath>)`
`spades.projectPath`	Default "."
`spades.scratchPath`	Default is within `tempdir()`, with subfolder
`SpaDES.project.Restart`	Default is FALSE. Passed to `Restart` argument in `setupProject`
`SpaDES.project.useGit`	Default is FALSE. Passed to `useGit` argument in `setupProject`

Value

named list of the *default* options currently available.

`user` *Helpers to develop easier to understand code.*

Description

A set of lightweight helpers that are often not strictly necessary, but they make code easier to read.

Usage

```
user(username = NULL)

machine(machinename = NULL)

node(machinename = NULL)
```

Arguments

`username`	A character string of a username.
`machinename`	A character string, which will be used as a partial match via `grep`, so the entire machine name is not necessary. A user can use `regex` if needed, e.g., `"^machine1"` will match `"machine15"` and `"machine12"`, but not `"thisIs_machine1"`.

Details

`node` is an alias for `machine`

Value

if `username` is non-NULL, returns a logical indicating whether the current user matches the supplied `username`. Otherwise returns a character string with the value of the current user.

`machine` returns a logical indicating whether the current machine name `Sys.info()[["nodename"]]` is matched by `machinename`.

Index