

Package: SpaDES.project (via r-universe)

October 16, 2026

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 2026-10-15

Version 1.2.0.9018

Depends R (>= 4.3)

Imports data.table, filelock, fs, grDevices, magick, methods, Require (>= 1.0.0), rprojroot, rstudioapi, stats, tools, utils

Suggests callr, cli, covr, crayon, digest, ellipsis, furr, future, future.apply, geodata, gert, ggplot2, patchwork, gh, gitcreds, googledrive, googlesheets4, htmlwidgets, http, igraph, jsonlite, knitr, leaflet, leafem, parallelly, pkgload, processx, purrr (>= 1.2.1), qs2, raster, RColorBrewer, rematch2, remotes, reproducible, rlang, rmarkdown, sf, SpaDES.core (>= 3.2.1.9026), SpaDES.tools, terra, testthat (>= 3.0.0), tidyterra, usethis, visNetwork, waldo, withr

Encoding UTF-8

Language en-CA

License GPL-3

VignetteBuilder knitr, rmarkdown

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

ByteCompile yes

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

Collate 'SpaDES.projectOptions.R' 'simLists-class.R'
'as.data.table-simLists.R' 'checkProject.R' 'dataInventory.R'
'diagnostics.R' 'download.R' 'environment.R'

```
'experiment-family.R' 'experiment2.R' 'experiment-legacy.R'
'experiment.R' 'fdDiagnostics.R' 'fileEdit.R' 'future.R'
'imports.R' 'getModule.R' 'googlesheets.R' 'helpers.R'
'librarySnapshot.R' 'listModules.R' 'makeDESCRIPTION.R'
'packages.R' 'paths2.R' 'pkgload2.R' 'remoteMember.R'
'resolveDots.R' 'sbatch.R' 'scenario.R' 'setupProject.R'
'spades-project-package.R' 'status-calculators.R'
'studyAreaPlotting.R' 'tmux-queue.R' 'tmux.R' 'txt.R'
'upload.R' 'workerResources.R' 'zzz.R'
```

Remotes PredictiveEcology/Require@development,
 PredictiveEcology/reproducible@development,
 PredictiveEcology/SpaDES.core@development,
 PredictiveEcology/SpaDES.tools@development

Config/roxygen2/version 8.1.0

Config/pak/sysreqs cmake make libmagick++-dev gsfonts libuv1-dev libssl-dev

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

Date/Publication 2026-10-16 02:31:58 UTC

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

RemoteRef development

RemoteSha 81318627c1a7b89753888d6db22f39a55b50993f

Contents

SpaDES.project-package	4
.libPathDefault	5
as.data.table.simLists	5
as_path	6
as_scenario	7
as_tarname	7
assessDoneInFigure	8
awaitExperimentFuture	8
awaitExperimentSBATCH	9
checkProject	9
dataInventory	12
experiment	14
experiment_family	16
experiment2	23
experimentFuture	26
experimentFutureList	30
experimentMonitor	32
experimentSBATCH	34
experimentTmux	38
factorialDesign	44
fdSelectLimit	45
findProjectPath	46

get_sim_year_heartbeat	46
getGithubFile	47
getModule	48
initialize,simLists-method	49
killExperimentFuture	50
killExperimentSBATCH	51
lastTraceback	51
librarySnapshot	52
listModules	54
localhostLabel	56
makeDESCRIPTIONproject	56
openFds	58
openFdsReport	58
outList	59
outSave	60
outSaveTarUpload	61
outScenarios	62
outTar	63
outUpload	64
packagesInModules	64
pathBuild	65
pathParse	66
pkgload2	67
plotChangeOverTime	67
plotSAs	68
plotTimeSeriesLeaflet	70
preRunSetupProject	71
queueRead	73
queueUploadMissing	74
reGet	74
reGetUntarLoad	75
register_scenario_format	77
reIndex	78
reLoad	79
reUntar	80
runWorkerLoopFuture	81
scenario	82
scenario_family	83
scenarioFields	85
setProjPkgDir	86
setupFiles	86
setupFunctions	87
setupGitIgnore	90
setupModules	91
setupOptions	94
setupPackages	97
setupParams	99
setupPaths	102

setupProject	106
setupSideEffects	115
setupStudyArea	117
show,simLists-method	119
simInitAndExperiment	119
simLists-class	120
spadesProjectOptions	121
statusCalculate_family	122
teardownProject	124
tmuxActiveRunningPath	125
tmuxFindDuplicates	125
tmuxKillPanels	126
tmuxListPanels	127
tmuxMirrorQueueToSheets	127
tmuxPrepareQueueFromDF	128
tmuxReconcileQueueWithDF	129
tmuxRefreshQueueStatus	130
tmuxRunNextWorker	131
tmuxRunWorkerLoop	132
tmuxSetMouse	133
tmuxSetPaneTitle	134
user	134

Index 136

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](#))

Authors:

- 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>

<code>.libPathDefault</code>	<i>SpaDES.project default .libPaths() directory</i>
------------------------------	---

Description

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

Usage

```
.libPathDefault(name)
```

Arguments

name	A text string. When used in <code>setupProject</code> , this is the <code>projectName</code>
------	--

Value

A path where the packages will be installed.

<code>as.data.table.simLists</code>	<i>Coerce elements of a <code>simLists</code> object to a <code>data.table</code></i>
-------------------------------------	---

Description

This is particularly useful to build plots using the **tidyverse**, e.g., **ggplot2**. Ported here from the now-unmaintained `SpaDES.experiment` package.

Usage

```
## S3 method for class 'simLists'
as.data.table(
  x,
  keep.rownames = FALSE,
  ...,
  vals,
  objectsFromSim = NULL,
  objectsFromOutputs = NULL
)
```

Arguments

<code>x</code>	An R object.
<code>keep.rownames</code>	Default is FALSE. If TRUE, adds the input object's names as a separate column named "rn". <code>keep.rownames = "id"</code> names the column "id" instead. For lists and when calling <code>data.table()</code> , names from the first named vector are extracted and used as row names, similar to <code>data.frame()</code> behavior.
<code>...</code>	Additional arguments. Currently unused.
<code>vals</code>	A (named) list of object names to extract from each <code>simList</code> , or a named list of quoted expressions to calculate for each <code>simList</code> , or a mix of character and quoted expressions.
<code>objectsFromSim</code>	Character vector of objects to extract from the <code>simLists</code> . If omitted, it will extract all objects from each <code>simList</code> in order to calculate the <code>vals</code> . This may have a computational cost. If NA, then no objects will be accessed from the <code>simList</code> . Objects identified here will only be as they are in the <code>simList</code> , i.e., at <code>end(sim)</code> .
<code>objectsFromOutputs</code>	List of (named) character vectors of objects to load from the <code>outputs(sim)</code> prior to evaluating <code>vals</code> . If there already is an object with that same name in the <code>simList</code> , then it will be overwritten with the object loaded from <code>outputs(sim)</code> . If there are many objects with the same name, specifically from several <code>saveTime</code> values in the <code>outputs(sim)</code> , these will all be loaded, one at a time, <code>vals</code> evaluated one at a time, and each of the values will be returned from each <code>saveTime</code> . A column, <code>saveTime</code> , will be part of the returned <code>data.table</code> . For cases where more than one object is required at a given <code>saveTime</code> , all should be identified here, without time specified. This function will take all identified objects from the same time period.

Details

See examples.

Value

This returns a `data.table` class object.

<code>as_path</code>	<i>Render a scenario (or list of them) as the canonical output path.</i>
----------------------	--

Description

Render a scenario (or list of them) as the canonical output path.

Usage

```
as_path(x, pre = "outputs", withFieldLabel = .scenario_env$withFieldLabel)
```

Arguments

x	A scenario, or anything coercible via as_scenario() .
pre	Path prefix (default "outputs").
withFieldLabel	Character vector of field names whose value should be prefixed with the field name in the path (paste0(label, value) instead of bare value). Defaults to the value registered via register_scenario_format() ; pass character(0) to force the bare-value default.

as_scenario	<i>Coerce any scenario representation to a canonical record.</i>
-------------	--

Description

Method-specific arguments (mapping, name_col, fields, withFieldLabel) are forwarded via ... to the dispatched method; see the method definitions in R/scenario.R.

Usage

```
as_scenario(x, ...)
```

Arguments

x	A scenario, a character path/tarname, a named list, or a data.frame / tibble / dribble.
...	Method-specific arguments (see Details).

Value

A scenario (single input) or list of scenarios.

as_tarname	<i>Render a scenario as an upload tar filename.</i>
------------	---

Description

Render a scenario as an upload tar filename.

Usage

```
as_tarname(
  x,
  ext = ".tar.gz",
  pre = "outputs",
  withFieldLabel = .scenario_env$withFieldLabel
)
```

Arguments

x	A scenario, or anything coercible via <code>as_scenario()</code> .
ext	File extension (default ".tar.gz").
pre	Path prefix (default "outputs").
withFieldLabel	Character vector of field names whose value should be prefixed with the field name in the path (<code>paste0(label, value)</code> instead of bare value). Defaults to the value registered via <code>register_scenario_format()</code> ; pass character(0) to force the bare-value default.

assessDoneInFigure	<i>Assess simulation status from PNG outputs</i>
--------------------	--

Description

Assess simulation status from PNG outputs

Usage

```
assessDoneInFigure(
  runName,
  timeout_min = 20,
  statusCalculate = getOption("spades.statusCalculate")
)
```

Arguments

runName	Directory containing the figures/hists
timeout_min	Threshold for inactivity (e.g., 20)
statusCalculate	A quoted expression to compute job status from output files. Defaults to <code>getOption("spades.statusCalculate")</code> .

awaitExperimentFuture	<i>Wait for all workers in an experimentFuture to finish</i>
-----------------------	--

Description

Blocks the calling R session until every worker has completed. Optionally prints a summary of final queue statuses.

Usage

```
awaitExperimentFuture(ef, verbose = TRUE)
```

Arguments

ef	An "experimentFuture" object returned by experimentFuture .
verbose	If TRUE (default), print a <code>table()</code> of final queue statuses after all workers finish.

Value

The ef object, invisibly.

awaitExperimentsSBATCH *Wait for all SBATCH workers to finish*

Description

Polls `squeue -j <ids>` every `interval_s` seconds until every job ID has left the queue. Optionally prints a final queue-status summary.

Usage

```
awaitExperimentSBATCH(es, interval_s = 30, verbose = TRUE)
```

Arguments

es	An "experimentsSBATCH" object returned by experimentsSBATCH .
interval_s	Polling interval in seconds. Default 30.
verbose	If TRUE (default), print a <code>table()</code> of final queue statuses after all jobs finish.

Value

The es object, invisibly.

checkProject *Preflight check of a project set up by `setupProject()`*

Description

Looks at what [setupProject\(\)](#) returned (the modules, paths, params and times) and reports the things that are likely to go wrong in `simInit()` or `spades()`, so a new user who passes little more than modules learns about them first. It reads module metadata and code only: it downloads nothing and does not run any module. The metadata is read from the module files without `simInit()` (see [dataInventory\(\)](#)) and kept for the session, so a repeated call is fast; the checks always run against the current `inSim`. [setupProject\(\)](#) calls it at its end, only printing messages, when `options(SpaDES.project.checkProject = TRUE)` (default FALSE); it never stops and never changes what [setupProject\(\)](#) returns. Each finding is printed once per session (the same check, module and message is not repeated) unless `forceShow = TRUE`; the returned table always has all of them. A check that cannot decide (a module file that does not parse, a missing tool) says nothing.

Usage

```
checkProject(
  inSim,
  ...,
  verbose = getOption("Require.verbose", 1L),
  forceShow = FALSE
)
```

Arguments

<code>inSim</code>	A list as returned by <code>setupProject()</code> ; the elements used are modules, paths (with <code>modulePath</code> , <code>inputPath</code> , <code>cachePath</code> , <code>outputPath</code> , <code>scratchPath</code>), params, times and objects. A module that is not in <code>paths\$modulePath</code> is skipped.
<code>...</code>	Elements that replace those of <code>inSim</code> (e.g. <code>times = list(start = 2020, end = 2030)</code>).
<code>verbose</code>	Numeric. Above 0 prints the summary.
<code>forceShow</code>	Logical. TRUE prints findings already printed earlier in the session.

Details

Every check gives rows with severity "error-likely" (the run will probably fail), "warning" (the result is probably not what was intended) or "note" (for information). To silence one kind of finding, remove it with the fix given; there is no per-check switch, only the one option above for the call made by `setupProject()`.

times (check = "times"):

`times` is missing, so `simInit()` uses 0 to 10, or its start is before 1800, while a module appears to read year-indexed data: a `sourceURL` or a parameter default holding a year from 1900 to 2100, or a parameter whose name contains "Year". Such modules look data up by calendar year, so model years give no data or the wrong year. Fix: set `times = list(start = 2020, end = 2050)` (calendar years).

studyAreaOrder (check = "studyAreaOrder"):

Module A lists a study-area object X (a name starting with `studyArea` or `rasterToMatch`) in `createsOutput`, module B lists it in `expectsInput`, and B's `.inputObjects` function uses `sim$X` or `sim[["X"]]`. Unless `options(spades.allowInitDuringSimInit = TRUE)`, B's `.inputObjects` runs before A's `init` has created X, so B builds its own default X instead of using A's. Fix: set that option; see `?SpaDES.core::spadesOptions`, section "Running some init events early". A module that itself expects objects from other modules cannot run early even with the option; with the option set, such an A is reported as a "note".

googleDrive (check = "googleDrive"):

A dataset in `dataInventory()` (metadata or code layer) is on `drive.google.com`, and no Google Drive credentials are configured: no `gargle_oauth_email` option, no `GOOGLEDRIVE_AUTH` or `GARGLE_SERVICE_ACCOUNT` environment variable, and `googledrive` is not installed or `googledrive::drive_has_token()` is FALSE. A batch run fails at the download; an interactive one stops to ask. Fix: authenticate before the run (`googledrive::drive_auth()`), or set one of those settings. The row is "error-likely" when the session is not interactive, otherwise "warning".

disk (check = "disk"):

Free space on the filesystem of paths\$inputPath, cachePath, outputPath, scratchPath and getOption("reproducible.destinationPathShared") is below getOption("SpaDES.project.checkProject.minFreeSpace") GB (one row per filesystem; not checked on Windows). Fix: free space or point those paths at a larger disk, or lower the option.

linkedParams (check = "linkedParams"):

A parameter that a module reads from the other modules with SpaDES.core::paramCheckOtherMods() (found by scanning its code), defined by two or more modules whose effective values differ. The effective value of a module is its own value in params, else the one in params\$.globals, else its default in the metadata. It is reported only when a value was supplied in params for at least one of them: modules' own defaults are never compared with each other. Numbers that are equal count as equal (10L and 10). The standard SpaDES parameters (.plots, .useCache, .saveInterval and similar) are not compared. Fix: set the value once in params\$.globals, or per module.

fallbackStudyArea (check = "fallbackStudyArea"):

A module has an input named studyArea or rasterToMatch (or one starting with those names) with a sourceURL, no module in the set creates it, and it was not supplied (objects, or an element of inSim of that name). The module will use that default, which is usually a large national or test area. Fix: supply the object, or add a module that creates it.

Value

Invisibly, a data.table with columns check, severity, module, message and fix, one row per finding (zero rows when nothing was found).

See Also

[dataInventory\(\)](#), [setupProject\(\)](#)

Examples

```
if (requireNamespace("SpaDES.core", quietly = TRUE)) {
  mp <- tempfile("modules")
  dir.create(file.path(mp, "demo"), recursive = TRUE)
  writelines(c(
    'defineModule(sim, list(name = "demo", description = "d", keywords = "k",',
    '  authors = person("A", "B", email = "a@b.c", role = c("aut", "cre")),',
    '  childModules = character(0), version = list(demo = "0.0.1"),',
    '  timeframe = as.POSIXlt(c(NA, NA)), timeunit = "year", citation = list(),',
    '  documentation = list(), reqdPkgs = list(),',
    '  parameters = bindrows(defineParameter("dataYear", "numeric", 2020, NA, NA, "year")),',
    '  inputObjects = bindrows(expectsInput("studyArea", "sf", "area",',
    '    sourceURL = "https://example.org/area.shp")),',
    '  outputObjects = bindrows()))', file.path(mp, "demo", "demo.R"))
  inSim <- list(modules = "demo", paths = list(modulePath = mp),
    times = list(start = 0, end = 10))
  res <- checkProject(inSim)
  res$check
}
```

dataInventory

*Inventory of the external data a set of modules reads***Description**

Lists every external dataset (a URL) that a set of SpaDES modules reads, one row per dataset URL and module, so that a project can see which modules use which data and which data each module needs. It looks at three layers: what the modules declare, what their code does, and, if a finished run is given, what the run downloaded.

Usage

```
dataInventory(
  modules,
  paths,
  sim = NULL,
  sources = c("metadata", "code", "run"),
  verbose = getOption("Require.verbose", 1L)
)
```

Arguments

modules	Character vector of module names, as returned by <code>setupProject()</code> ("owner/repo@ref" is reduced to the module name). The children of a parent module (its <code>childModules</code>) are included.
paths	The paths list returned by <code>setupProject()</code> (its <code>modulePath</code> is used), or the module path itself as a character vector.
sim	An optional <code>simList</code> from a finished run, for layer 3.
sources	Which layers to use: any of "metadata", "code", "run". "run" is used only if <code>sim</code> is given.
verbose	Numeric. Above 0 prints a one-line summary.

Details**Layer 1: metadata:**

Every `expectsInput(..., sourceURL = )` in a module's metadata is a row (a `sourceURL` that is NA or empty is skipped). The metadata is read from the parsed `defineModule()` call of the module file, without `simInit()`; a `sourceURL` that cannot be worked out without a `simList` counts as NA. The read is kept for the session and redone when the file changes. `role` is "fallback" when another module in `modules` lists the same object name in `createsOutput`, because that is what `suppliedElsewhere()` tests at run time: the module's own download then only runs if no other module supplies the object. Otherwise `role` is "default". The metadata cannot know a parameter-dependent choice (for example `dataSource = "KNN"` against "SCANFI"), so a "default" row is the declared source, not necessarily the one a given run uses.

Layer 2: code:

A static scan of each module's .R files (the module file and its R/ folder, not tests/) with the R parser (`utils::getParseData()`) for calls to `prepInputs`, `preProcess`, `download.file`, `googledrive::drive_download`, `httr::GET`, `httr2::req_perform`, `curl::curl_download` and similar, also when wrapped in `Cache()`. Each call gives the literal URL if the `url` argument is a string, the metadata `sourceURL` for `extractURL("object")`, the file and line (where), the enclosing function (call holds the function that downloads, event the event it belongs to when the module's `doEvent` function calls it directly, and `.inputObjects` counts as "init"), and the object the result is assigned to when it is `sim$object <- .`. Calls into package helpers such as `LandR::prepInputs_CWIM()` are opaque: the row has `url = NA` and the helper name in `call`, so there is a download there that the scan cannot name. A `url` built at run time (`paste0()`, a variable) also gives `url = NA`, or the literal start of it, with the expression in `note`. A row found by both layers is one row, with `source = "metadata+code"`.

Layer 3: run:

If `sim` is given, `reproducible::urlLog(sim)` (SpaDES.core's method for a `simList`) supplies the module, event, URL and cache id of every `prepInputs/preProcess` access of the run. Rows are joined on module and URL; a match has `observed = TRUE` and the `cacheId`; a URL the run recorded that neither other layer knew becomes a row with `source = "run"`: a download the static layers missed. A ledger can miss downloads that did happen: fetches that bypass `prepInputs/preProcess` (for example `googledrive` or `download.file`); cache entries created before URL tagging, which have no tags to replay on a cache hit or a `cacheChaining` jump; and fallbacks that did not run because another module supplied the object. So `observed = FALSE` means "not seen in this run", not "not used". The ledger is kept only if `options(reproducible.urlLog)` is not `FALSE` and `options(spades.urlLog)` is not `FALSE` (both are on by default) while the run is made.

Reading the result:

`source` lists the layers that saw the row, joined by "+", in the order `metadata`, `code`, `run`. Use `grepl("run", source)` rather than `source == "run"` to select by layer.

```
inv <- dataInventory(modules, paths)
inv[!is.na(inv$url), ]           # all named datasets
split(inv$module, inv$url)      # by dataset: which modules use it
split(inv$url, inv$module)      # by module: which data it needs
inv[inv$role == "fallback", ]    # downloads that may never run
inv[is.na(inv$url), ]           # downloads the scan could not name
```

`reproducible::prepInputsLog()` is the session-wide record of every URL accessed, with no module attached, and `reproducible::urlLog(sim)` is the per-run record; `dataInventory()` adds the static view, which exists before anything has run.

Value

A data.table, one row per dataset URL and module (rows without a URL are one per call), with columns `module`, `object`, `url`, `source`, `event`, `call`, `where` (file:line for rows from code), `role` ("default" or "fallback"), `observed` (TRUE if the run's ledger has it), `cacheId` (from the run) and `note`.

See Also

[SpaDES.core::moduleMetadata\(\)](#), [metadataInModules\(\)](#), [reproducible::urlLog\(\)](#), [reproducible::prepInputsLog\(\)](#)

Examples

```
if (requireNamespace("SpaDES.core", quietly = TRUE)) {
  mp <- tempfile("modules")
  dir.create(file.path(mp, "demo"), recursive = TRUE)
  writeLines(c(
    'defineModule(sim, list(name = "demo", description = "d", keywords = "k",',
    '  authors = person("A", "B", email = "a@b.c", role = c("aut", "cre")),',
    '  childModules = character(0), version = list(demo = "0.0.1"),',
    '  timeframe = as.POSIXlt(c(NA, NA)), timeunit = "year", citation = list(),',
    '  documentation = list(), reqdPkgs = list(), parameters = bindrows(),',
    '  inputObjects = bindrows(expectsInput("dem", "SpatRaster", "elevation",',
    '    sourceURL = "https://example.org/dem.tif")),',
    '  outputObjects = bindrows()))',
    'Init <- function(sim) {',
    '  sim$dem <- prepInputs(url = extractURL("dem"), destinationPath = tempdir())',
    '  sim$roads <- prepInputs(url = "https://example.org/roads.zip")',
    '  sim',
    '}', file.path(mp, "demo", "demo.R"))
  dataInventory("demo", list(modulePath = mp))
}
```

experiment

Run an experiment using [SpaDES.core::spades\(\)](#)

Description

A wrapper around [experiment2\(\)](#) that builds a fully-factorial set of simLists from a single base simList plus alternative params / modules / inputs / objects, then runs them via [experiment2\(\)](#)'s future backend. The factorial design is built with [factorialDesign\(\)](#).

Usage

```
experiment(
  sim,
  replicates = 1,
  params,
  modules,
  objects = list(),
  inputs,
  dirPrefix = "simNum",
  substrLength = 3,
  saveExperiment = TRUE,
  experimentFile = "experiment.RData",
  clearSimEnv = FALSE,
```

```

    notOlderThan,
    cl,
    ...
  )

```

Arguments

<code>sim</code>	A <code>simList</code> , acting as the basis for the experiment.
<code>replicates</code>	The number of replicates to run of the same <code>simList</code> .
<code>params</code>	Like for <code>SpaDES.core::simInit()</code> , but for each parameter, provide a list of alternative values.
<code>modules</code>	Like for <code>SpaDES.core::simInit()</code> , but a list of module names (as strings).
<code>objects</code>	Like for <code>SpaDES.core::simInit()</code> , but a list of named lists of named objects.
<code>inputs</code>	Like for <code>SpaDES.core::simInit()</code> , but a list of inputs <code>data.frames</code> .
<code>dirPrefix</code>	String vector. This will be concatenated as a prefix on the directory names.
<code>substrLength</code>	Numeric. While making <code>outputPath</code> for each <code>spades</code> call, this is the number of characters kept from each factor level.
<code>saveExperiment</code>	Logical. Should the resulting experimental design be saved to a file. Default <code>TRUE</code> .
<code>experimentFile</code>	String. Filename if <code>saveExperiment</code> is <code>TRUE</code> ; saved to <code>outputPath(sim)</code> in <code>.RData</code> format.
<code>clearSimEnv</code>	Logical. If <code>TRUE</code> , then the <code>envir(sim)</code> of each <code>simList</code> in the return is emptied, to reduce RAM load. Default <code>FALSE</code> .
<code>notOlderThan</code>	Currently unused (kept for back-compatibility).
<code>cl</code>	Deprecated and ignored; control parallelism with <code>future::plan()</code> .
<code>...</code>	Passed to <code>experiment2()</code> and onward to <code>SpaDES.core::spades()</code> (e.g. <code>debug</code> , <code>plotInitialTime</code> , <code>cache</code> , and <code>events</code> – see Controlling events in <code>experiment2()</code>).

Details

This function (and the `simLists` class it produces) was moved here from the now-unmaintained `SpaDES.experiment` package. Two behavioural notes versus the historical version: parallelism is now controlled by `future::plan()` rather than a `cl` cluster object (the `cl` argument is accepted but ignored, with a message), and the return value is a `simLists` object (as from `experiment2()`) rather than a plain list. The experimental design table is still saved to `experimentFile` and is attached to the result (see `Value`).

Value

Invisibly, a `simLists` object. The experimental design list (`expDesign + expVals`) is attached as an attribute named "experiment" on the object's data environment, i.e. `attr(out@.xData, "experiment")`, and is also written to `experimentFile`.

Author(s)

Eliot McIntire

See Also

[experiment2\(\)](#), [factorialDesign\(\)](#), [as.data.table.simLists\(\)](#), [experiment_family](#)

experiment_family

Experiment functions: five ways to run a SpaDES experiment

Description

A SpaDES "experiment" is a way of running a simulation many times with varying inputs, parameters, paths, scenarios, or replicates. This lets you run, for example, replication of stochastic models, hypothesis testing with different data inputs, scenario analysis of different human decisions, building large datasets of alternative mechanisms to enable ensemble modeling, and other possibilities.

Details

There are five functions to choose from; these can be classified into two groups. The first group ([experiment\(\)](#) / [experiment2\(\)](#)) is conceptually simpler: it works on in-memory `simList` objects directly. The second group ([experimentTmux\(\)](#) / [experimentFuture\(\)](#) / [experimentsSBATCH\(\)](#)) is built around a project `global.R` script (typically where [setupProject\(\)](#) is run) and a shared job queue. This second group becomes more useful as the number of runs (e.g., scenarios, replicates) become numerous, long, spread across machines, or being run from a High Performance Compute cluster.

In-memory simLists

These take `simList` object(s) directly and are analogous to running `SpaDES.core::spades()`. They return results as a `simLists` ("plural") object you can post-process e.g., with [as.data.table.simLists\(\)](#) or any other custom methods. These functions are best when the run set is modest, fits in RAM, and you want the result objects back in your session. They are *not* built for resume-after-crash, cross-machine pulls, or HPC. (Moved here from the now-unmaintained `SpaDES.experiment` package.)

[experiment2\(\)](#) The core in-memory runner: give it one or more `simLists` (and optionally replicates) and it runs them all and returns a `simLists`. You build the variation yourself, e.g. with several `SpaDES.core::simInit()` calls.

[experiment\(\)](#) A light wrapper around [experiment2\(\)](#) that *builds* the variation for you: give it one base `simList` plus alternative params / modules / inputs / objects and it constructs the fully-factorial set of `simLists` (via [factorialDesign\(\)](#)) and runs them. [factorialDesign\(\)](#) is exported separately, so the same design can also seed the `df` of the second group below.

With a source file (e.g., `global.R`)

Here, the user describes the experiment using a `data.frame` (or `data.table`) in which each column name and row value defines the set of object-value pairs that will be assigned to variables in the `.GlobalEnv`. When the user runs one of these functions, the `data.frame` is translated into a queue `data.frame` that has all the same columns and rows, plus a few more (`status`, `claimed_by`,

etc.) to coordinate the run. After creating the queue, the function spawns a number of independent R "worker" sessions (according to `n_workers` or `cores`). Each worker selects a single row, assigns the values in each user-specified column to an object in the `.GlobalEnv` whose name is the column name, then `source()`s `global.R`. For example, if the `data.frame` has 2 rows and a column named `runName` with values `"trial1"` and `"trial2"`, the first worker runs `runName <- "trial1"; source(global_path)` and the second runs `runName <- "trial2"; source(global_path)`. The status column starts as `"PENDING"` for all rows; workers take the next `"PENDING"` (or `"INTERRUPTED"`) row, skipping `"DONE"` rows (and any other status, such as `"HOLD"`), and mark a row `"DONE"` when it finishes without error before moving to the next. A job that errors is marked `"ERROR"`, which no worker takes; set its status back to `"PENDING"` to run it again. A worker never overwrites a status you set while its job ran.

These three share the queue and the run-naming convention and differ only in *how* parallel workers are spawned:

`experimentTmux()` Allows the most interactivity and so is helpful when there is still debugging to perform. This will only work on a computer that has `tmux` installed. The function spawns one `tmux` pane per worker, optionally across `ssh`-reachable machines. Best for interactive use where you want to watch workers live (`tmux attach`). Workers can be stopped with `tmuxKillPanels()`.

`experimentFuture()` When there is little to no debugging necessary, this function will use background R processes using either `callr::r_bg()` if all workers are local, or `future::cluster` if some of the workers are on different machines. Best for stable scripts. Workers can be stopped with `killExperimentFuture()`.

`experimentsBATCH()` One Slurm batch job per worker. Best for HPC clusters with `sbatch` / `squeue` / `scancel`. Block with `awaitExperimentSBATCH()` (polls `squeue`) or stop with `killExperimentSBATCH()` (graceful via stop files; `force = TRUE` issues `scancel`). Inspect generated job scripts with `dry_run = TRUE`.

All three of these accept the same core arguments:

`df` The parameter grid; one row = one job. Column names become R variables in the worker's `.GlobalEnv` before `global.R` is sourced.

`global_path` Path to the R script each worker sources per job. Must be on a filesystem visible to all workers (matters for `experimentSBATCH()` and remote-host modes of the other two).

`queue_path` Path to the local RDS queue file. Workers coordinate through file-based locks on this file; remove it (or point a fresh path) to start over, leave it to resume.

`runNameLabel` Quoted expression evaluated against each row to derive a human-readable identifier (used in log messages, sentinel filenames, and `tmuxListPanels()` output). Default is the first two non-meta columns of the queue.

`statusCalculate` Optional quoted expression that inspects the job's outputs and returns up-to-date status / heartbeat metadata. `statusCalculate_LandR` and `statusCalculate_FireSenseFit` are pre-built blocks for the most common `SpaDES` module outputs.

`ss_id` Optional Google Sheets / Drive *folder* ID. When provided, workers mirror queue state to a sheet so a remote stakeholder can watch progress in a browser. With `ss_id = NULL` (default) the queue is purely local – no Google APIs are touched.

A typical usage pattern:

```
df <- expand.grid(scenario = c("A", "B"), .rep = 1:2,
                 stringsAsFactors = FALSE)
ef <- experimentFuture(df = df, global_path = "global.R",
                      n_workers = 2L, log_dir = "logs")
```

Swap `experimentFuture()` for `experimentTmux()` or `experimentSBATCH()` (adjusting cores / `n_workers` / `sbatch_opts`) and the rest of the driver script is unchanged.

Why not just run `Rscript -e ... per row?`

At its core, that is exactly what each worker does. A worker assigns the row's columns into `.GlobalEnv` and calls `source("global.R")`, which is equivalent to:

```
Rscript -e '.ELFind <- "6.3.1"; .rep <- 1; source("global.R")'
Rscript -e '.ELFind <- "6.3.1"; .rep <- 2; source("global.R")'
```

When the number of sets to run is small, this works. As you add scenarios, machines, authentication, race conditions, etc. the bookkeeping grows past what's comfortable to maintain by hand. The `experimentXXX` functions are just that bookkeeping.

`experimentXXX` functions deal with several issues that arise when running "parallel" scripts, including:

Concurrency control Two shells launched at the same second can both pick the same row. The `experimentXXX` functions take an exclusive filelock lock on the queue between read and write, so each row is claimed at most once across all workers and machines.

Resume after crash / ctrl-C If a worker dies mid-job, the row is stuck "in progress" with no record. The `experimentXXX` functions mark the row RUNNING when claimed and DONE / ERROR / INTERRUPTED when finished, so the next launch skips DONE and ERROR rows and (optionally, via `tmuxRefreshQueueStatus()` or `experimentFutureList()` (kill = TRUE)) demotes orphaned RUNNING rows back to PENDING for re-claim.

Worker-pool sizing `Rscript & Rscript & Rscript` & scales as "one process per row", which thrashes the box once you exceed the core count. The `experimentXXX` functions take `n_workers` and let each worker pull rows in sequence, so you cap parallelism explicitly.

Cross-machine claims Spawning N rows on each of M machines means either replicating the parameter grid by hand (and risking duplicate work) or sharding it (and losing dynamic load-balancing). With the `experimentXXX` functions, every worker on every machine pulls from the same queue, so a slow machine just claims fewer rows.

Live observability `Rscript -e` writes nothing structured – you scrape PIDs and tail logs. The `experimentXXX` functions maintain a queue with `status` / `claimed_by` / `started_at` / `process_id` / `machine_name` so `queueRead()` gives a full snapshot, and `experimentFutureList()` can enumerate live workers (and kill them) cluster-wide.

Remote-stakeholder visibility When `ss_id` is supplied, the queue is mirrored to a Google Sheet a collaborator can open in a browser; without that, "how is the run going?" requires SSH access to the runner machine.

Outputs accounting `queueUploadMissing()` / `outScenarios()` anti-join the queue against the Drive upload folder so you can see which DONE rows still need to be packaged and uploaded.

Run-name + status hooks `runNameLabel` and `statusCalculate` give one place to derive directory names and inspect output artifacts – both per-runner and in `tmuxRefreshQueueStatus()` for post-hoc rescans – without each `global.R` re-implementing them.

If you only ever run two rows on one machine and never restart, the two-line shell version is fine. The `experimentXXX` functions exist for the cases past that.

Cross-machine propagation (cluster modes)

When you launch on more than one machine – `experimentTmux(cores = c("mega", "birds"))` or `experimentFuture(cores = c("localhost", "camas"))` – `.setup_remote_machine()` runs once per unique remote host before any worker starts. It tries to make the remote R session look enough like the local one that `global.R` runs the same way. What it propagates / sets up:

Package versions `SpaDES.project` itself is rsynced from the local `.libPaths()[1]` to the remote (or, if loaded via `devtools::load_all()`, the source tree is rsynced and R CMD INSTALL-ed). `Require` is version- and `RemoteSha`-checked and rsynced if it's older or comes from a different source than locally. Then `Require::Install()` installs every package in `SpaDES.project`'s `Imports` / `Depends` / `LinkingTo`, plus any `Suggests` installed locally (so optional runtime dependencies like `googlesheets4`, `cli`, etc. follow along but the dev toolchain doesn't).

Compiled-from-source packages `terra`, `sf`, `rgdal`, `rgeos`, `lwgeom` are forced to compile from source on the remote (so they link against the remote's `libgdal.so` etc., which may be a different soversion than `localhost`'s).

System libraries A best-effort `sudo -n apt-get install -y --no-install-recommends` of the dev headers needed for the source-compiled packages (`libgdal-dev`, `libssl-dev`, `libcurl4-openssl-dev`, `libxml2-dev`, `fonts/graphics`, ...). Runs non-interactively; if passwordless `sudo` isn't configured the failure is logged and setup continues, expecting the libraries to be there already.

R startup environment The remote `~/.Rprofile` gets refreshed with `.libPaths(c(<local_lib>, .libPaths()))`, `options(repos = c("https://predictiveecology.r-universe.dev", <local_repos>))`, `options(defaultPackages = ...)` (so the remote uses the same minimal default-attached set as a fresh `Rscript`), and `Sys.setenv(CURL_CA_BUNDLE, SSL_CERT_FILE)` pointing at the system CA bundle (so `libcurl` can do HTTPS even when `/etc/profile.d/` isn't sourced under non-login SSH). The remote `$BASH_ENV`, if set, is wrapped in a subshell guard so a misbehaving `sleep $UNSET` can't kill the SSH command shell before R starts.

GitHub credentials The local `GITHUB_PAT` (read from `gitcreds::gitcreds_get()` or a caller-supplied `local_pat_file`) is written to the remote `~/.Renviro` (`chmod 0600`) and to a per-lib file `<local_lib>/spades_github_pat` that's read at the top of `~/.Rprofile`. `git credential approve` is also called so command-line `git` on the remote authenticates the same way. Required for `pak` to install private modules / dev packages from GitHub.

Google credentials The `experimentXXX` functions pass `email` + `cache_path` into each worker; the worker calls `googlesheets4::gs4_auth(email = email, cache = cache_path)` non-interactively against the same cached OAuth token directory the local session uses. The token directory itself isn't pushed (it's expected to already exist via NFS or a prior login on the remote); only the `gargle_oauth_email` / `gargle_oauth_cache` options are forwarded so the same identity is selected. If the cache isn't there, the worker prints a `gs4_auth` warning and continues without GS access.

User code: `R/ folder + modules` The directory next to `global.R` called `R/` (where project-specific helper functions live) is `rsync -a --delete-ed` to the remote, so anything `global.R source()`s from `R/` works there too. With `copyModules = TRUE` the `SpaDES` module path (`getOption("spades.modulePath")`) is also `rsynced`, so module code stays in step.

The job artifacts themselves `global.R` and the `queue.rds` are `scp'd` into the same path on the remote (or, if the path is already on NFS such as `/mnt/shared_cache/...`, they're effectively no-ops – same absolute path on both ends).

Net effect: `global.R` on `camas` sees the same packages at the same versions, the same `GITHUB_PAT`, the same `R/` helpers, the same SSL trust store, and the same Google identity as `global.R` on `mega`. Hand-rolling all of that for each remote machine before each run is the bulk of what makes "Rscript -e ... on N hosts" miserable in practice; the `experimentXXX` functions do it once per unique host per call.

Managing remote workers from the calling machine

Once `experimentFuture(cores = c("localhost", "camas", "dougfir"))` is launched, the workers on `camas` and `dougfir` are no longer reachable via local `ps / tools::pskill()` – they are R processes on other machines. `experimentFutureList()` is the cluster-wide handle for them. Pass it the `ef` object and it will:

1. Read the queue file (which is the authoritative record: every claim writes `machine_name + process_id` under a filelock, so workers on every machine appear there even when they didn't redirect their stdout to a discoverable `worker_NN.log`).
2. Probe each entry in `ef$cores` once with `ssh <core> hostname -s` to build a map from OS hostname (which is what `Sys.info()[["nodename"]]` writes to the queue) to the SSH alias the master used to reach it (e.g. `A159604 -> dougfir`). This is needed because `ssh A159604` typically fails – only `ssh dougfir` resolves via `~/.ssh/config / etc/hosts`.
3. For every `status == "RUNNING"` row, verify the worker is actually alive: `file.exists("/proc/<pid>")` for the local machine, batched `ssh <alias> "[ -d /proc/<pid> ]"` for each remote machine (one SSH connection per machine).
4. Return a `data.frame` with `pid`, `machine`, `started_at`, `queue_path`, `runName` for every live worker – local and remote, in one table.

`kill = TRUE` uses the same map to send the chosen signal (`TERM` default, `INT` or `KILL` on request): `tools::pskill()` for local PIDs and a single batched `ssh <alias> "kill -<sig> p1 p2 ..."` per remote machine. After signalling, it polls (locally via `/proc`, remotely via SSH) for up to 10 s until the workers actually exit, then runs `tmuxRefreshQueueStatus()` on each unique queue file to demote the now-orphaned `RUNNING` rows back to `PENDING`. When `ss_id` was supplied to the original `experimentFuture()` call, an `<queue_path>.ss_id` sidecar is left behind; `kill = TRUE` reads it and pushes the same demotion to the Google Sheet via `.gs_demote_after_kill()`, so the GS view converges with the local queue without a separate cleanup step.

Three usage shapes:

```
experimentFutureList(ef)                # list everything live
experimentFutureList(ef, kill = TRUE)    # graceful TERM + queue refresh
experimentFutureList(ef, kill = TRUE, signal = "KILL") # immediate
```

Across R sessions, when ef is gone, drive discovery off the queue path directly:

```
experimentFutureList(queue_paths = "/mnt/shared_cache/.../future_queue.rds")
```

Without ef, the hostname-to-alias probe is skipped, so the SSH check uses machine_name verbatim – which only works if the OS hostname is itself reachable via SSH on the calling node (i.e. it appears in ~/.ssh/config or /etc/hosts as a Host entry). If not, you’ll need to either keep ef in scope or add the OS hostnames to your SSH config.

Concretely, the things you can do post-launch from the calling machine without ever opening a terminal on camas / dougfi:

- See which row each remote worker is currently on.
- Confirm that a remote worker actually died after a crash / network blip (otherwise the queue would stay stuck at RUNNING and no one would re-claim).
- Send SIGTERM cluster-wide to abort an experiment mid-run, then immediately re-launch a fixed global.R against the same queue (any DONE rows are skipped, demoted RUNNING rows are re-claimed).
- Mirror that demotion to the Google Sheet so a stakeholder watching in a browser sees the change without needing to be told.

Resource monitoring (CPU + RAM)

`experimentMonitor()` is the read-only entry point. Discovery depends on what you pass:

- `experimentMonitor()` (no args) – enumerates every tmux pane on the calling machine across all tmux servers, same as the historical `tmuxListPanes()`.
- `experimentMonitor(ef)` – queue-driven discovery across all machines in ef\$cores (with the hostname-to-SSH-alias probe described above).
- `experimentMonitor(queue_paths = "...")` – same as ef mode, but for cross-session use when the ef handle is gone.

stats = TRUE batches ps -o pid=,%cpu=,rss=,state= (locally and via one SSH connection per remote node) to append:

- state – R (running on CPU), S (sleeping / waiting), D (uninterruptible sleep, often disk I/O – persistent D = hang), T (stopped), Z (zombie), Closed (R session exited but tmux pane still open).
- cpuAvg – percent CPU averaged over the process’s lifetime (note: *not* the instantaneous rate htop shows).
- RAM (GB) – resident memory (RSS), 1 decimal place.
- availableCores – total CPUs on the node, from nproc.
- total RAM (GB) – total RAM on the node, from /proc/meminfo.

availableCores and total RAM (GB) are constant across all rows on the same node, so each pane’s resource use is visible relative to its node capacity. Unreachable nodes get NA for all their rows; titles missing a parseable <node>-<pid> get NA too – one bad pane / unreachable host doesn’t poison the rest of the table.

Single function, three sources, same stats columns either way – so a stakeholder running `experimentMonitor(ef, stats = TRUE)` on a laptop sees the same per-worker CPU / RAM picture that `experimentMonitor(stats = TRUE)` (legacy tmux mode) gives on the master node. `tmuxListPanels()` is preserved as a thin alias that calls `experimentMonitor()` with no `ef`, so older code keeps working unchanged.

Related families:

- `scenario_family` – canonical record for one row of `df`, reversibly convertible between field values, an output directory path, and an upload tarball filename.
- `queueRead()` / `queueUploadMissing()` / `outList()` / `outScenarios()` – helpers for queues persisted to a Google Sheet plus a Drive upload folder, including the queue-vs-uploads anti-join.
- `experimentMonitor()` – read-only worker / pane lister. With no args, scans tmux panes; with `ef` or `queue_paths`, scans the queue file's RUNNING rows and verifies each PID is alive (local /proc, batched SSH for remotes). `stats = TRUE` adds per-worker CPU / RSS / state and per-node nproc / total RAM via batched ps. `tmuxListPanels()` is a thin alias for the no-args form.
- `tmuxRefreshQueueStatus()` / `tmuxFindDuplicates()` / `tmuxKillPanels()` – operational tools that work regardless of which runner produced the queue.
- `experimentFutureList()` – experimentFuture-side equivalent of `tmuxListPanels()`: discovers live workers across the cluster (driven off the queue file's RUNNING rows plus an `ssh <core> hostname -s` alias probe), and with `kill = TRUE` sends TERM / INT / KILL to all of them in one call (local via `tools::pskill()`, remote batched per machine via SSH), then refreshes the queue and demotes the matching Google-Sheet rows when an `<queue_path>.ss_id` side-car is present.

Controlling which events run

All of these honour `spades()`'s `events` argument, which restricts the events executed for each module (see `SpaDES.core::spades()`):

- `experiment()` / `experiment2()`: pass events as a named argument; it is forwarded to every `spades()` call, e.g. `experiment2(sim1, sim2, events = list(fireSpread = "init"))`. The *same* events apply to all simulations / replicates.
- `experimentTmux()` / `experimentFuture()` / `experimentSBATCH()`: there is no events argument because these functions do not call `spades()` – your `global.R` does. To control events, add an events column to `df` (each cell is the events spec for that row) and, inside `global.R`, call `spades(sim, events = events)`. Because each row carries its own value, this gives *per-scenario* control of which events run for any particular module – something the single shared events of the in-memory family cannot do.

experiment2

Run experiment, algorithm 2, using `SpaDES.core::spades()`

Description

Given one or more `simList` objects, run a series of `spades` calls in a structured, organized way. Methods are available to deal with outputs, such as `as.data.table.simLists()` which can pull out simple to complex values from every resulting `simList` or object saved by outputs in every `simList` run. This uses `future` internally, allowing for various backends and parallelism.

Usage

```
experiment2(
  ...,
  replicates = 1,
  clearSimEnv = FALSE,
  createUniquePaths = c("outputPath"),
  useCache = FALSE,
  debug = getOption("spades.debug"),
  drive_auth_account = NULL,
  meanStaggerIntervalInSecs = 1
)
```

Arguments

- | | |
|--------------------|---|
| ... | One or more <code>simList</code> objects. Additional named arguments are passed through to <code>SpaDES.core::spades()</code> (see Controlling events below). |
| replicates | The number of replicates to run of the same <code>simList</code> . See details and examples. To minimize memory overhead, currently, this must be length 1, i.e., all ... <code>simList</code> objects will receive the same number of replicates. |
| clearSimEnv | Logical. If TRUE, then the <code>envir(sim)</code> of each <code>simList</code> in the return list is emptied. This is to reduce RAM load of large return object. Default FALSE. |
| createUniquePaths | A character vector of the paths passed to <code>simInit</code> , indicating which should create a new, unique path, as a sub-path to the original paths. Currently only "outputPath" is honoured. Pass <code>character(0)</code> (or NULL) to disable this nesting entirely, e.g. when the caller (such as <code>experiment()</code>) has already set each <code>simList</code> 's <code>outputPath</code> . |
| useCache | Logical. Passed to <code>spades</code> . This will be passed with the <code>simList</code> name and replicate number, allowing each replicate and each <code>simList</code> to be seen as a non-cached call to <code>spades</code> . |
| debug | Passed to <code>SpaDES.core::spades()</code> . |
| drive_auth_account | Optional character string. If provided, it will be passed to each worker and run as <code>googledrive::drive_auth(drive_auth_account)</code> to allow a specific user account for googledrive |

meanStaggerIntervalInSecs

If used, this will use `Sys.sleep(cumsum(c(0, rnorm(nbrOfWorkers() - 1, mean = meanStaggerIntervalInSecs/10))))` and distribute these delays to the workers.

Details

This function was moved here from the now-unmaintained `SpaDES.experiment` package. See also the file-queue based [experiment_family](#) (e.g. `experimentFuture()`) for a different, script-oriented approach.

This function, because of its class formalism, allows for methods to be used. For example, `as.data.table.simLists()` allows user to pull out specific objects (in the `simList` objects or on disk saved in `outputPath(sim)`).

The `outputPath` is changed so that every simulation puts outputs in a sub-directory of the original `outputPath` of each `simList` (unless `createUniquePaths` is `character(0)/NULL`).

Value

Invisibly returns a `simLists` object. This class extends the environment class and contains `simList` objects.

Controlling events

Any named argument in ... that is not consumed by `experiment2` is passed straight to `SpaDES.core::spades()`. In particular, `spades()`'s `events` argument is honoured, so `experiment2(sim1, sim2, events = list(...))` restricts the events that run for every simulation. Note this applies the *same* events specification to all `simLists` / replicates. For per-scenario control of events, use the file-queue [experiment_family](#) with an `events` column in `df` (see [experiment_family](#)).

Note

A `simLists` object can be made manually, if, say, many manual `spades` calls have already been run. See example, via `new("simLists")`

Author(s)

Eliot McIntire

See Also

[as.data.table.simLists\(\)](#), [SpaDES.core::spades\(\)](#), [experiment\(\)](#), [experiment_family](#)

Examples

```
## Not run:
if (require("ggplot2", quietly = TRUE) &&
    require("NLMR", quietly = TRUE) &&
    require("RColorBrewer", quietly = TRUE)) {
  library(SpaDES.core)
  library(SpaDES.project)
```

```

tmpdir <- file.path(tempdir(), "examples")
# Make 3 simLists -- set up scenarios
endTime <- 2

# Example of changing parameter values
# Make 3 simLists with some differences between them
mySim <- lapply(c(10, 20, 30), function(nFires) {
  simInit(
    times = list(start = 0.0, end = endTime, timeunit = "year"),
    params = list(
      .globals = list(stackName = "landscape", burnStats = "nPixelsBurned"),
      # Turn off interactive plotting
      fireSpread = list(.plotInitialTime = NA, spreadprob = c(0.2), nFires = c(10)),
      caribouMovement = list(.plotInitialTime = NA),
      randomLandscapes = list(.plotInitialTime = NA, .useCache = "init")
    ),
    modules = list("randomLandscapes", "fireSpread", "caribouMovement"),
    paths = list(modulePath = system.file("sampleModules", package = "SpaDES.core"),
      outputPath = tmpdir),
    # Save final state of landscape and caribou
    outputs = data.frame(
      objectName = c(rep("landscape", endTime), "caribou", "caribou"),
      saveTimes = c(seq_len(endTime), unique(c(ceiling(endTime / 2), endTime))),
      stringsAsFactors = FALSE
    )
  )
})

planTypes <- c("sequential") # try others! ?future::plan
sims <- experiment2(sim1 = mySim[[1]], sim2 = mySim[[2]], sim3 = mySim[[3]],
  replicates = 3)

# Try pulling out values from simulation experiments
# 2 variables
df1 <- as.data.table(sims, vals = c("nPixelsBurned", NCaribou = quote(length(caribou$x1))))

# Now use objects that were saved to disk at different times during spades call
df1 <- as.data.table(sims,
  vals = c("nPixelsBurned", NCaribou = quote(length(caribou$x1))),
  objectsFromOutputs = list(nPixelsBurned = NA, NCaribou = "caribou"))

# now calculate 4 different values, some from data saved at different times
# Define new function -- this calculates perimeter to area ratio
fn <- quote({
  landscape$Fires[landscape$Fires[] == 0] <- NA;
  a <- boundaries(landscape$Fires, type = "inner");
  a[landscape$Fires[] > 0 & a[] == 1] <- landscape$Fires[landscape$Fires[] > 0 & a[] == 1];
  peri <- table(a[]);
  area <- table(landscape$Fires[]);
  keep <- match(names(area), names(peri));
  mean(peri[keep]/area)
})

```

```

df1 <- as.data.table(sims,
  vals = c("nPixelsBurned",
    perimToArea = fn,
    meanFireSize = quote(mean(table(landscape$Fires[])[-1])),
    caribouPerHaFire = quote({
      NROW(caribou) /
      mean(table(landscape$Fires[])[-1])
    })),
  objectsFromOutputs = list(NA, c("landscape"), c("landscape"),
    c("landscape", "caribou")),
  objectsFromSim = "nPixelsBurned")

if (interactive()) {
  # with an unevaluated string
  library(ggplot2)
  p <- lapply(unique(df1$vals), function(var) {
    ggplot(df1[vals == var,],
      aes(x = saveTime, y = value, group = simList, color = simList)) +
    stat_summary(geom = "point", fun.y = mean) +
    stat_summary(geom = "line", fun.y = mean) +
    stat_summary(geom = "errorbar", fun.data = mean_se, width = 0.2) +
    ylab(var)
  })

  # Arrange all 4 -- could use gridExtra::grid.arrange -- easier
  pushViewport(viewport(layout = grid.layout(2, 2)))
  vlayout <- function(x, y) viewport(layout.pos.row = x, layout.pos.col = y)
  print(p[[1]], vp = vlayout(1, 1))
  print(p[[2]], vp = vlayout(1, 2))
  print(p[[3]], vp = vlayout(2, 1))
  print(p[[4]], vp = vlayout(2, 2))
}
}

## End(Not run)

```

experimentFuture

Run parallel R jobs using background processes (tmux-free)

Description

A tmux-free alternative to [experimentTmux](#) that dispatches parallel workers as background R processes via **callr**. Workers claim jobs from a GoogleSheets or local-RDS queue, source `global_path` for each job, and write all console output to per-worker log files on localhost.

The function returns immediately (non-blocking) with a handle object of class "experimentFuture". Use [awaitExperimentFuture](#) to block until all workers finish, or `print(ef)` to check live status. Because workers write to files via `callr::r_bg()`'s `stdout` / `stderr` arguments, logs appear in real time and can be followed with `tail -f`.

Usage

```

experimentFuture(
  df,
  global_path = "global.R",
  cores = NULL,
  n_workers = if (is.null(cores)) 4L else length(cores),
  queue_path = NULL,
  on_interrupt = c("requeue", "fail"),
  on_error = c("stop", "continue"),
  ss_id = NULL,
  forceLocalQueueToGS = FALSE,
  email = getOption("gargle_oauth_email"),
  cache_path = getOption("gargle_oauth_cache"),
  runNameLabel = quote(colnames(q)[1:2]),
  log_dir = "logs",
  activeRunningPath = getOption("spades.activeRunningPath"),
  sp_dev_path = NULL,
  local_pat_file = NULL,
  copyModules = FALSE,
  sync_library = NULL,
  snapshot_library = TRUE,
  ...
)

```

Arguments

<code>df</code>	data.frame of parameter combinations. Each row is one job.
<code>global_path</code>	Path to the R script each worker sources per job. Defaults to "global.R" in the current directory.
<code>cores</code>	NULL for local parallel workers, or a character vector of SSH hostnames for remote workers. When <code>cores</code> is provided, <code>.setup_remote_machine</code> is called for each unique host before workers are launched, replicating the full SpaDES environment (packages, GitHub PAT, gargle OAuth cache) on each remote machine.
<code>n_workers</code>	Number of parallel workers. Defaults to <code>length(cores)</code> for remote workers, or 4L for local.
<code>queue_path</code>	Path to the local RDS queue file. Created automatically if it does not yet exist. Defaults to <code><dirname(global_path)>/future_queue.rds</code> .
<code>on_interrupt</code>	"requeue" (default) re-queues interrupted jobs as PENDING; "fail" marks them INTERRUPTED permanently. A job that errors (rather than being interrupted) is marked ERROR and is not claimed again until its status is set back to PENDING.
<code>on_error</code>	"stop" (default) leaves a worker stopped after a job errors, with the error visible in its log; "continue" goes on to the next claimable job. The errored row is marked ERROR either way (see <code>on_interrupt</code>).
<code>ss_id</code>	Google Sheets ID (or Drive folder ID) for the shared queue. When provided workers use the GS backend instead of the local RDS file.

<code>forceLocalQueueToGS</code>	If TRUE, overwrite the GS sheet with the local df even if the sheet already contains rows.
<code>email</code>	Gargle OAuth e-mail for Google Sheets auth.
<code>cache_path</code>	Gargle OAuth cache directory.
<code>runNameLabel</code>	Quoted expression evaluated in the job environment to derive a human-readable run name (used in log messages and queue metadata).
<code>log_dir</code>	Directory for per-worker log files. Created if needed. Defaults to "logs" relative to the current working directory.
<code>activeRunningPath</code>	Directory for <code>Running_*.rds</code> marker files (file-based backend only).
<code>sp_dev_path</code>	Local path to <code>SpaDES.project</code> source tree to sync to remote workers (optional; uses installed binary if NULL).
<code>local_pat_file</code>	Path to a file containing a GitHub PAT to copy to remote workers.
<code>copyModules</code>	Logical. If TRUE and remote hosts are present, rsyncs the directory given by <code>getOption("spades.modulePath")</code> to the same absolute path on each remote host after <code>.setup_remote_machine()</code> completes. Issues a warning and skips if the option is unset. Default FALSE.
<code>sync_library</code>	NULL (default) or a character vector of package specifications for <code>Require::Install()</code> . If given, syncProjectLibrary() runs once, now, in a fresh process and only while no worker is alive, before any worker starts. Never sync per job.
<code>snapshot_library</code>	Logical (default TRUE). Each worker session takes a hardlinked snapshot of the project library before loading anything from it, so an install into the shared library cannot corrupt a running job. See librarySnapshot .
<code>...</code>	Additional named arguments stored in <code>.future_dots.rds</code> and loaded into each worker's <code>.GlobalEnv</code> before sourcing <code>global_path</code> .

Value

An object of class "experimentFuture" (a list) containing:

`procs` List of `callr::r_bg` process objects, one per local worker (or future objects for remote cluster workers).

`log_files` Character vector of log file paths.

`log_dir` Absolute path to the log directory.

`queue_path` Absolute path to the queue RDS file.

`cores` The cores argument as supplied.

See Also

[experimentTmux](#), [awaitExperimentFuture](#), [tmuxRunWorkerLoop](#)

Examples

```
## Not run:
## -- Minimal: build a tiny global.R, then run a 2 x 2 experiment -----
tdir <- file.path(tempdir(), "experimentFuture-demo")
dir.create(tdir, showWarnings = FALSE, recursive = TRUE)
writeLines(
  'message("scenario=", .scenario, " rep=", .rep); Sys.sleep(2)',
  file.path(tdir, "global.R")
)
expt <- expand.grid(.scenario = c("A", "B"), .rep = 1:2,
  stringsAsFactors = FALSE)

ef <- experimentFuture(
  df          = expt,
  global_path = file.path(tdir, "global.R"),
  n_workers   = 2L,
  queue_path  = file.path(tdir, "future_queue.rds"),
  log_dir     = file.path(tdir, "logs")
)

## -- Live inspection while workers run -----
print(ef)                                # alive/done per worker
experimentMonitor(ef)                    # pid + machine + runName
experimentMonitor(ef, stats = TRUE)      # adds CPU / RAM / state
queueRead(ef$queue_path)                 # full queue snapshot
experimentFutureList(ef)                 # cluster-wide pid list
cat(readLines(ef$log_files[[1L]]), sep = "\n") # tail one log

awaitExperimentFuture(ef)                # blocks until both workers exit

## -- Killing workers -----

# Graceful stop: workers finish their CURRENT job, then exit.
# Any remaining PENDING jobs stay in the queue and can be resumed later
# by calling experimentFuture() again with the same queue_path.
killExperimentFuture(ef)

# Immediate stop (force): workers are killed immediately.
# Jobs that were mid-execution may remain as RUNNING in the queue; reset them with:
#   tmuxRefreshQueueStatus(ef$queue_path) # file-based backend
# The GS backend reclaims stale RUNNING entries automatically before each new claim.
killExperimentFuture(ef, force = TRUE)
tmuxRefreshQueueStatus(ef$queue_path)    # clean up stale RUNNING entries

# Cluster-wide kill (works for `cores = c(...)` clusters too):
# sends SIGTERM to every worker on every machine, waits for exit, runs
#   tmuxRefreshQueueStatus(), and pushes the demotion to the Google Sheet
# if `ss_id` was used (via the <queue_path>.ss_id sidecar).
experimentFutureList(ef, kill = TRUE)

## -- Resuming after a kill -----
```

```

# Jobs left as PENDING (or INTERRUPTED with on_interrupt = "requeue") are
# automatically picked up when you call experimentFuture() again with the
# same queue_path -- no need to re-specify df.
ef2 <- experimentFuture(
  df          = expt,          # ignored if queue_path already exists
  global_path = file.path(tdir, "global.R"),
  n_workers   = 2L,
  queue_path  = file.path(tdir, "future_queue.rds"),
  log_dir     = file.path(tdir, "logs")
)
awaitExperimentFuture(ef2)  # wait for remaining jobs to finish

queueRead(ef2$queue_path)  # full snapshot (data.table)
table(queueRead(ef2$queue_path)$status)  # all DONE

cat(readLines(ef2$log_files[[1]]), sep = "\n")  # inspect worker 1 log

## -- Remote workers (pre-setup required) -----
ef <- experimentFuture(
  df          = expt,
  global_path = file.path(tdir, "global.R"),
  cores       = c("node01", "node02"),
  n_workers   = 2L,
  ss_id       = "YOUR_GOOGLE_SHEET_ID",
  email       = "you@example.com",
  cache_path  = "~/cache/gargle",
  local_pat_file = "~/github_pat"
)
killExperimentFuture(ef)    # graceful stop on remote workers too

## End(Not run)

```

experimentFutureList *Find (and optionally kill) live experimentFuture workers*

Description

Cross-session worker discovery for `experimentFuture`. Scans `/proc` for R processes whose redirected stdout points to a worker_<NN>.log file (the convention written by `callr::r_bg(stdout = log_files[[i]])` in `experimentFuture`), regardless of which R session originally spawned them. This is the right tool when:

- you re-ran the `experimentFuture` example in a new R session and a fresh `tail -f` is silent because the previous run's workers are still claiming queue rows;
- you want to clean up orphans without remembering each `ef` handle;
- you want a one-glance view of *which row* each worker is currently running (joined against the queue's status == "RUNNING" process_id).

Linux-only (uses `/proc/<pid>/fd/1` to find the log file each worker is writing). For other Unixes use `lsof -p <pid>` or `ps -ef | grep tmuxRunWorkerLoop` as a manual substitute.

Usage

```
experimentFutureList(
  ef = NULL,
  kill = FALSE,
  signal = c("TERM", "INT", "KILL"),
  queue_paths = NULL
)
```

Arguments

ef	Optional shorthand: an "experimentFuture" object (or list of them) whose queue_path will be added to the discovery set. Equivalent to passing queue_paths = ef\$queue_path and handy when the result of experimentFuture() is still in scope.
kill	If TRUE, send signal to every worker found, wait up to 10 s for the processes to exit, then call tmuxRefreshQueueStatus() on each unique queue_path to demote the now-orphaned RUNNING rows back to PENDING. Default FALSE (list-only).
signal	One of "TERM" (15, default; graceful), "INT" (2; like Ctrl-C), or "KILL" (9; immediate).
queue_paths	Optional character vector of queue .rds paths to inspect for workers. Use this across R sessions when the ef handle is no longer in scope (e.g. you restarted R but the workers from a prior experimentFuture() call are still alive on mega and camas). Each queue's status == "RUNNING" rows are verified for liveness via /proc (local) or batched SSH (remote). When NULL (default) and ef is also NULL, the function uses only queue files auto-discovered from local /proc – which in turn only finds callr::r_bg workers, not PSOCK cluster workers, so on a node with no r_bg workers it sees nothing unless ef or queue_paths is supplied.

Value

A data.frame (one row per live worker) with columns:

pid Worker process ID.

started_at Approximate process start time (ctime of /proc/<pid>).

log_file Path the worker is writing stdout/stderr to.

queue_path The first *_queue.rds found in the log directory's parent (where experimentFuture puts it by default), or NA if not located.

runName Hyphen-joined data column values of the row this worker is currently running, derived from the queue's status == "RUNNING" entry whose process_id matches. NA if the worker is between jobs.

When kill = TRUE, the same data.frame is returned (invisibly) describing the workers that were signalled.

See Also

[experimentFuture](#), [killExperimentFuture](#), [tmuxRefreshQueueStatus](#)

Examples

```
## Not run:
# Just list everything that's running (auto-discovery via /proc only)
experimentFutureList()

# Pass the ef handle to also pick up PSOCK cluster workers and remote
# workers (anything in the queue, on any machine in `cores`).
ef <- experimentFuture(df = df, global_path = "global.R",
                      cores = c("localhost", "camas"), ...)
experimentFutureList(ef)
experimentFutureList(ef, kill = TRUE)

# Across R sessions, when ef is gone, drive discovery off the queue path:
experimentFutureList(queue_paths = "/mnt/shared_cache/.../future_queue.rds")

# Hard kill (SIGKILL, no chance to update queue meta on the worker side --
# but the post-kill tmuxRefreshQueueStatus() still demotes the rows).
experimentFutureList(ef, kill = TRUE, signal = "KILL")

## End(Not run)
```

experimentMonitor	<i>Monitor live workers across an experiment (tmux panes or callr/cluster futures)</i>
-------------------	--

Description

Single read-only entry point for inspecting workers regardless of which runner spawned them. Discovery is driven by what you pass:

Usage

```
experimentMonitor(ef = NULL, queue_paths = NULL, stats = FALSE)
```

Arguments

ef	Optional "experimentFuture" object (or list of them) whose queue_path and cores will be used for discovery. Switches the function from tmux-scan mode to queue-scan mode.
queue_paths	Optional character vector of queue .rds paths. Equivalent to passing ef = NULL plus queue_paths; used when the ef handle is no longer in scope (e.g. across R sessions). When queue_paths is supplied without ef, the SSH-alias probe is skipped and machine_name from the queue is used verbatim as the SSH target

– which only works if the OS hostname is itself a Host entry in `~/ .ssh/config` / `/etc/hosts`.

stats Logical. When TRUE, queries `ps` per worker (locally or via batched SSH) to append state, `cpuAvg` (percent CPU averaged over the process's lifetime – not the instantaneous rate `htop` shows), `RAM` (GB) (resident memory), `availableCores` (total CPUs on the node, from `nproc`), and `total RAM` (GB) (total RAM on the node, from `/proc/meminfo`). Default FALSE.

Details

- **Default** (`ef = NULL`, `queue_paths = NULL`) – enumerates tmux panes via `tmux -S <socket> list-panes -a` across every tmux server under `$TMUX_TMPDIR/tmux-<uid>/`. Same behaviour the historical `tmuxListPanes()` had. Per-socket failures are swallowed so one broken socket cannot poison the rest; works outside a tmux pane and across multiple tmux servers (e.g. sessions started under different `-L` names). `Cluster_Monitor` panes are filtered out.
- **ef supplied (or queue_paths)** – reads each queue file's `status == "RUNNING"` rows, probes `ssh <core> hostname -s` once per non-local entry in `ef$cores` to map OS hostnames (which is what `Sys.info()[["nodename"]]` writes to the queue) back to SSH aliases (`~/ .ssh/config` / `/etc/hosts` entries), and verifies each PID is alive (`/proc/<pid>` locally, batched `ssh <alias> "[ -d /proc/<pid> ]"` remotely). This is the `experimentFuture()` / `experimentSBATCH()` equivalent of the tmux pane scan – workers there don't necessarily live in a tmux pane, so the queue file is the authoritative record.

Either way, `stats = TRUE` runs the same `ps -o pid=,%cpu=,rss=,state= batch` (locally and via one SSH connection per remote node) to append CPU / RSS / state plus per-node `nproc` / total RAM.

Value

`Data.frame` whose columns depend on the discovery mode:

- **tmux mode** – `session`, `window`, `pane`, `pane_id`, `pane_ref` (the "session:window.pane" string), `title`, `node` (first dash-separated token in `title` that matches a cluster alias from `/etc/hosts`; falls back to `localHostLabel()` when the title contains only the raw local hostname; NA if no match).
- **queue mode** – `pid`, `machine`, `started_at`, `log_file` (NA when the worker isn't a `callr::r_bg` writer), `queue_path`, `runName`.

With `stats = TRUE`, five additional columns appear in either mode: `state`, `cpuAvg`, `RAM` (GB), `availableCores`, `total RAM` (GB). Returns an empty `data.frame` (0 rows, same columns) if no workers are found.

State codes

The `state` column is the best single signal for hang-detection because it is a snapshot (no time window needed). Values:

State	Meaning
R	running on CPU right now

S	sleeping (waiting on I/O, timer, or lock)
D	uninterruptible sleep (usually disk I/O; persistent D can indicate a hang)
T	stopped (SIGSTOP or similar)
Z	zombie (dead but not yet reaped)
Closed	worker process has exited – PID no longer exists
NA	could not determine (machine unreachable, or no parseable <node>--<pid> in title)

See Also

[experimentFutureList\(\)](#) for the same queue-mode discovery plus cluster-wide kill / queue refresh / GS demotion. [tmuxListPanels\(\)](#) is preserved as a thin alias that calls this function with no ef.

experimentSBATCH	<i>Run parallel R jobs on a Slurm cluster (SBATCH-based)</i>
------------------	--

Description

A Slurm-native sibling of [experimentTmux](#) and [experimentFuture](#). Submits `n_workers` long-lived SBATCH jobs that each call [tmuxRunWorkerLoop](#) against the shared queue, claiming and running rows until the queue is empty (or the worker's stop file appears). Same queue / `global.R` / `runNameLabel` / `statusCalculate` semantics as the other two runners.

Returns a non-blocking handle of class "experimentSBATCH" carrying the Slurm job IDs. Use [awaitExperimentSBATCH](#) to poll squeue until all jobs leave the queue, or [killExperimentSBATCH](#) to stop them (gracefully via stop files, or immediately via `scancel`).

Usage

```
experimentSBATCH(
  df,
  global_path = "global.R",
  n_workers = 4L,
  queue_path = NULL,
  on_interrupt = c("requeue", "fail"),
  on_error = c("stop", "continue"),
  ss_id = NULL,
  forceLocalQueueToGS = FALSE,
  email = getOption("gargle_oauth_email"),
  cache_path = getOption("gargle_oauth_cache"),
  runNameLabel = quote(colnames(q)[1:2]),
  log_dir = "logs",
  activeRunningPath = getOption("spades.activeRunningPath"),
  sbatch_opts = list(),
  sbatch_cmd = "sbatch",
  r_cmd = file.path(R.home("bin"), "Rscript"),
  r_libs = .libPaths(),
```

```

    dry_run = FALSE,
    sync_library = NULL,
    snapshot_library = TRUE,
    ...
)

```

Arguments

<code>df</code>	data.frame of parameter combinations. Each row is one job. Ignored if <code>queue_path</code> already exists (workers resume from the existing queue).
<code>global_path</code>	Path to the R script each worker sources per job. Defaults to "global.R" in the current directory. Must be on a filesystem visible to compute nodes (e.g. shared NFS / Lustre).
<code>n_workers</code>	Number of SBATCH jobs to submit. Defaults to 4L.
<code>queue_path</code>	Path to the local RDS queue file. Created automatically if it does not yet exist. Defaults to <code><dirname(global_path)>/sbatch_queue.rds</code> . Must be on shared storage so all worker nodes can read/write it.
<code>on_interrupt</code>	"requeue" (default) re-queues interrupted jobs as PENDING; "fail" marks them INTERRUPTED permanently. A job that errors (rather than being interrupted) is marked ERROR and is not claimed again until its status is set back to PENDING.
<code>on_error</code>	"stop" (default) leaves a worker stopped after a job errors, with the error visible in its log; "continue" goes on to the next claimable job. The errored row is marked ERROR either way (see <code>on_interrupt</code>).
<code>ss_id</code>	Google Sheets ID (or Drive folder ID) for the shared queue. When provided workers use the GS backend in addition to the local RDS file (mirroring <code>experimentFuture</code>).
<code>forceLocalQueueToGS</code>	If TRUE, overwrite the GS sheet with the local df even if the sheet already contains rows.
<code>email</code>	Gargle OAuth e-mail for Google Sheets auth (only used when <code>ss_id</code> is non-NULL).
<code>cache_path</code>	Gargle OAuth cache directory.
<code>runNameLabel</code>	Quoted expression evaluated in the job environment to derive a human-readable run name (used in log messages and queue metadata). Defaults to <code>quote(colnames(q)[1:2])</code> .
<code>log_dir</code>	Directory for per-worker log files, generated job scripts, and stop files. Created if needed. Defaults to "logs" relative to the current working directory. Must be on shared storage.
<code>activeRunningPath</code>	Directory for <code>Running_*.rds</code> marker files (file-based backend only).
<code>sbatch_opts</code>	Named list of SBATCH directives. Each name = value becomes <code>#SBATCH --<name>=<value></code> in the generated job script. Underscores in names are translated to hyphens, so <code>cpus_per_task = 4</code> becomes <code>#SBATCH --cpus-per-task=4</code> . Set a value to NULL or TRUE for flag-only directives (<code>#SBATCH --<name></code>). Common keys: <code>partition</code> , <code>time</code> , <code>mem</code> , <code>cpus_per_task</code> , <code>account</code> , <code>nodes</code> , <code>ntasks_per_node</code> , <code>constraint</code> , <code>gres</code> .

<code>sbatch_cmd</code>	Path to the sbatch executable. Defaults to "sbatch" (resolved on \$PATH). Override on systems where sbatch is wrapped or non-standard.
<code>r_cmd</code>	Path to the R interpreter to invoke on compute nodes. Defaults to <code>file.path(R.home("bin"), "Rscript")</code> .
<code>r_libs</code>	Character vector of library paths to set via <code>.libPaths()</code> inside each worker. Defaults to the master's <code>.libPaths()</code> so the worker sees the same package set; override when compute nodes have a different filesystem layout.
<code>dry_run</code>	If TRUE, generate the job scripts but do not submit them. Returns a handle whose <code>job_ids</code> are all NA. Useful for inspecting what would be submitted.
<code>sync_library</code>	NULL (default) or a character vector of package specifications for <code>Require::Install()</code> . If given, syncProjectLibrary() runs once, now, in a fresh process and only while no worker is alive, before any worker starts. Never sync per job.
<code>snapshot_library</code>	Logical (default TRUE). Each worker session takes a hardlinked snapshot of the project library before loading anything from it, so an install into the shared library cannot corrupt a running job. See librarySnapshot .
<code>...</code>	Additional named arguments stored in <code>.sbatch_dots.rds</code> and loaded into each worker's <code>.GlobalEnv</code> before sourcing <code>global_path</code> .

Value

An object of class "experimentSBATCH" (a list) containing:

`job_ids` Integer vector of Slurm job IDs (or NA under `dry_run = TRUE`).

`job_scripts` Character vector of generated SBATCH script paths.

`log_files` Character vector of log file paths.

`stop_files` Character vector of stop-file paths.

`log_dir` Absolute path to the log directory.

`queue_path` Absolute path to the queue RDS file.

See Also

[experimentTmux](#), [experimentFuture](#), [awaitExperimentsSBATCH](#), [killExperimentsSBATCH](#)

Examples

```
## Not run:
## -- Minimal: build a tiny global.R, then run a 2 x 2 experiment -----
# Use a directory on your shared HPC filesystem (NFS / Lustre / BeeGFS).
tdir <- file.path(tempdir(), "experimentSBATCH-demo")
dir.create(tdir, showWarnings = FALSE, recursive = TRUE)
writeLines(
  'message("scenario=", .scenario, " rep=", .rep); Sys.sleep(2)',
  file.path(tdir, "global.R")
)
expt <- expand.grid(.scenario = c("A", "B"), .rep = 1:2,
  stringsAsFactors = FALSE)
```

```

es <- experimentSBATCH(
  df      = expt,
  global_path = file.path(tdir, "global.R"),
  n_workers = 2L,
  queue_path = file.path(tdir, "sbatch_queue.rds"),
  log_dir    = file.path(tdir, "logs"),
  sbatch_opts = list(partition = "compute", time = "00:30:00", mem = "1G")
)

## -- Live inspection while jobs run -----
print(es)                                # job IDs + squeue status
queueRead(es$queue_path)                 # full queue snapshot
experimentMonitor(queue_paths = es$queue_path) # cluster-wide pid + machine
experimentMonitor(queue_paths = es$queue_path, stats = TRUE) # + CPU/RAM

awaitExperimentsSBATCH(es)               # block until squeue empty

## -- Larger experiment with full sbatch_opts -----
es <- experimentSBATCH(
  df      = expt,
  global_path = file.path(tdir, "global.R"),
  n_workers = 4L,
  queue_path = file.path(tdir, "sbatch_queue.rds"),
  log_dir    = file.path(tdir, "logs"),
  sbatch_opts = list(
    partition = "compute",
    time      = "24:00:00",
    mem       = "16G",
    cpus_per_task = 4,
    account   = "my_alloc"
  )
)

print(es)                                # job IDs + squeue status per worker
awaitExperimentsSBATCH(es)               # blocks until every job ID leaves squeue

# Graceful stop (workers exit between jobs, queue rows stay PENDING):
killExperimentsSBATCH(es)

# Immediate stop (scancel; stale RUNNING entries can be cleaned up via:
#   tmuxRefreshQueueStatus(es$queue_path)):
killExperimentsSBATCH(es, force = TRUE)
tmuxRefreshQueueStatus(es$queue_path)

## -- Resume after stop -----
# Same `queue_path` -> DONE rows are skipped, demoted PENDING rows are
# re-claimed by the new sbatch jobs.
es2 <- experimentSBATCH(
  df      = expt,                                # ignored if queue exists
  global_path = file.path(tdir, "global.R"),
  n_workers = 2L,
  queue_path = file.path(tdir, "sbatch_queue.rds"),

```

```

log_dir      = file.path(tdir, "logs"),
sbatch_opts = list(partition = "compute", time = "00:30:00", mem = "1G")
)
awaitExperimentSBATCH(es2)
table(queueRead(es2$queue_path)$status)          # all DONE

## End(Not run)

```

experimentTmux

Spawn tmux worker panes and process a job queue

Description

Creates `n_workers` tmux panes in the current window, tiles them, and starts a worker loop in each one that claims and runs jobs from a file-backed queue (`queue_path`). Control returns immediately to the **master pane**; all work happens asynchronously inside the worker panes.

Worker loop modes (`pane_mode`):

"killAndNewPane" (*default*):

Each worker runs **one job per R session**, then exits. A fresh R session starts automatically for the next job, freeing all memory between runs.

- **localhost panes**: After each job, `tmuxRunWorkerLoop()` calls `tmux respawn-pane -k`, which replaces the current pane's process in-place with a new Rscript invocation. No retiling needed.
- **Remote panes** (`cores = "hostname"`): The local pane runs a bash while-loop that repeatedly calls `ssh -t host bash -c 'exec env R_PROFILE_USER=<script> R --interactive'`. `ssh -t` allocates a PTY so R runs interactively (readline, OSC 2 title updates, Ctrl+C propagation). A startup script injected via `R_PROFILE_USER` runs one job then exits; `q(status = 1L)` (job done or queue empty) lets the while-loop start a fresh R session, `q()` (status 0) stops the loop. `R_PROFILE_USER` is unset inside R immediately after startup so workers spawned by `makeClusterPSOCK()` do not inherit it and inadvertently re-run the startup script.

"reuse":

Each worker loops inside a single R session (`repeat { tmuxRunNextWorker() }`). Memory accumulates across jobs – useful for lightweight simulations.

Remote machine setup (`cores`):

Supplying a hostname in `cores` triggers `.setup_remote_machine()` once per unique host before any workers start. Steps run in this order:

1. **Guard BASH_ENV** – wraps the remote `$BASH_ENV` file's existing content in a subshell (`(( ... ) 2>/dev/null || true)`) so that any exit or failing command inside it cannot abort the non-interactive SSH shell that carries setup commands.
2. **Create remote directory; copy files** – `mkdir -p` the remote working directory (same relative path from `~` as on localhost), then `scp` `global_path`, `queue_path`, and `dots_path` (if supplied) into it.

3. **Rsync project R/ folder** – syncs the R/ subdirectory next to `global_path` to the remote with `rsync --delete` so user-defined helper functions sourced by `global.R` are up to date.
4. **Write ~/.Rprofile on remote** – injects three lines (replacing any previous versions): `.libPaths(c(local_lib, ...))` so the project library takes precedence over system libraries; `options(repos = ...)` including the PredictiveEcology r-universe; and an SSL block that sets `CURL_CA_BUNDLE/SSL_CERT_FILE` so HTTPS downloads work in non-login SSH sessions where `/etc/profile.d/` is not sourced.
5. **Verify/install Require** – compares the remote Require version and git commit SHA to the local installation. If they differ, rsyncs the installed directory (GitHub source) or runs `install.packages("Require")` (CRAN source).
6. **Install usethis** on the remote via `Require::Install()`.
7. **Propagate GitHub credentials** – reads the local token via `gitcreds::gitcreds_get()` and pipes it into `git credential approve` on the remote so private GitHub packages can be installed without interactive setup. Falls back to checking whether the remote already has credentials; errors if neither is true.
8. **Install system libraries** via `sudo -n apt-get install -y --no-install-recommends` (non-interactive; fails gracefully if passwordless sudo is not configured). Libraries installed: `spatial` (`libgdal-dev`, `libgeos-dev`, `libproj-dev`, `libsqlite3-dev`, `libudunits2-dev`), `HTTP/TLS` (`libssl-dev`, `libcurl4-openssl-dev`), `XML` (`libxml2-dev`), `archive` (`libarchive-dev`), `git` (`libgit2-dev`), `fonts/graphics` (`libfontconfig1-dev`, `libharfbuzz-dev`, `libfribidi-dev`, `libpng-dev`, `libjpeg-dev`, `libtiff-dev`, `libfreetype6-dev`), `protobuf` (`libabsl-dev`), and R compilation headers (`r-base-dev`).
9. **Ensure remote lib path exists** – `mkdir -p` the project library path on the remote (must match localhost exactly so installed file paths are identical).
10. **Rsync SpaDES.project** – copies the locally installed SpaDES.project directory to the same path on the remote. Both machines must share the same platform and R version so compiled lazy-load databases are compatible.
11. **Install SpaDES.project dependencies** via `Require::Install()`. Spatial packages (`terra`, `sf`, `rgdal`, `rgeos`, `lwgeom`) are compiled from source so they link against the remote's actual GDAL/GEOS/PROJ versions. All other hard dependencies (`Imports/Depends/LinkingTo`) plus any `Suggests` packages installed locally are installed as binaries via `Require::setLinuxBinaryRepo()`. Common packages with strict version requirements (`purrr >= 1.2.1`, `rlang >= 1.1.7`, `cli >= 3.6.0`, `vctrs >= 0.6.0`) are pre-installed to the project library to avoid stale system-library versions being picked up during compilation.
12. **Rsync Require package cache** (`Require::cachePkgDir()`) to the remote to accelerate future package installations.
13. **Rsync gargle OAuth cache** (`cache_path` or `getOption("gargle_oauth_cache")`) to the remote so the worker can authenticate with Google APIs (Sheets, Drive) without a browser prompt.

Staggered starts:

Pane 1 starts immediately. Pane $i > 1$ waits `delay_before_source + (i - 2) * stagger_by` seconds inside R before claiming its first job, avoiding simultaneous queue contention at startup. For remote workers in `killAndNewPane` mode the stagger only applies to the first R session; subsequent while-loop iterations start immediately.

Restarting a broken pane:

If a worker pane is manually interrupted (e.g. Ctrl+C) and drops to a shell prompt, restart it by pressing (up-arrow) (up-arrow) in that pane and hitting Enter. The full command is always in the pane's bash history:

- **localhost:** Rscript -e "... " (re-enters tmuxRunWorkerLoop; in killAndNewPane mode respawn-pane takes over from the first job onward).
- **remote:** if setup && scp; then first_run; _st=\$?; while [\$_st -ne 0]; do sleep 2; loop_run; _st=\$?; command (restarts the sh loop from scratch; plain POSIX – works in bash, dash, and sh).

ANSI colour support:

At startup, experimentTmux sets the tmux session option `default-terminal = "tmux-256color"`. This ensures that all subsequently created panes advertise a full-colour ANSI terminal, which is required for R packages such as cli and crayon to render coloured/dynamic output correctly. Without this, connections that arrive via Windows PowerShell -> SSH -> tmux often inherit TERM=screen or no TERM at all, causing R to fall back to plain-text output. The setting is applied globally to the session (-g) and persists for the session's lifetime; it does not modify ~/.tmux.conf.

Usage

```
experimentTmux(
  df,
  global_path = "global.R",
  cores = NULL,
  n_workers = if (is.null(cores)) 4L else length(cores),
  delay_after_split = 0.4,
  delay_after_layout = 0.4,
  delay_between_R_start = 0,
  delay_before_source = 60,
  stagger_by = delay_before_source,
  set_mouse = TRUE,
  statusCalculate = getOption("spades.statusCalculate"),
  folderWithIterInFilename = getOption("spades.folderWithIterInFilename"),
  activeRunningPath = getOption("spades.activeRunningPath"),
  continue = TRUE,
  queue_path = NULL,
  on_interrupt = c("requeue", "fail"),
  on_error = c("stop", "continue"),
  pane_mode = c("killAndNewPane", "reuse"),
  ss_id = NULL,
  forceLocalQueueToGS = FALSE,
  onExistingQueue = c("resume", "append", "rebuild"),
  enableGSSync = FALSE,
  email = getOption("gargle_oauth_email"),
  cache_path = getOption("gargle_oauth_cache"),
  workersToMonitor = unique(if (is.null(cores)) "localhost" else cores),
  runNameLabel = quote(colnames(q)[1:2]),
  copyModules = FALSE,
  sync_library = NULL,
```

```

    snapshot_library = TRUE,
    ...
)

```

Arguments

<code>df</code>	A data.frame of parameter combinations. Each row is one job. Column names become object names in worker panes; values from each row are assigned prior to sourcing <code>global_path</code> .
<code>global_path</code>	Character scalar. Absolute path to the script sourced for each job.
<code>cores</code>	Character vector of machine hostnames, recycled to <code>n_workers</code> . Use "localhost" for the local machine or a bare hostname (e.g. "sbw") for a remote machine reachable via passwordless SSH. When any remote hosts are listed, <code>.setup_remote_machine()</code> is called for each unique hostname before workers start. Default NULL (all local-host).
<code>n_workers</code>	Integer. Number of worker panes to spawn. Defaults to <code>length(cores)</code> if <code>cores</code> is supplied, otherwise 4.
<code>delay_after_split</code>	Numeric. Seconds to wait after each split-window. Default 2.
<code>delay_after_layout</code>	Numeric. Seconds to wait after select-layout. Default 0.2.
<code>delay_between_R_start</code>	Numeric. Seconds to wait after starting R in each pane. Default 0.1.
<code>delay_before_source</code>	Numeric. Seconds panes 2..n wait before claiming their first job. Default 60.
<code>stagger_by</code>	Numeric. Additional seconds per pane beyond pane 2: pane <code>i > 1</code> waits <code>delay_before_source + (i - 2) * stagger_by</code> . Default <code>delay_before_source</code> .
<code>set_mouse</code>	Logical. Enable tmux mouse support (pane selection, scroll). Default TRUE.
<code>statusCalculate</code>	A quoted expression (optionally using <code>runName</code>) that evaluates to a path containing job-status output files. Currently used by <code>fireSense_SpreadFit</code> . Default <code>getOption("spades.statusCalculate", NULL)</code> .
<code>folderWithIterInFilename</code>	A quoted expression (optionally using <code>runName</code>) for a folder whose filenames encode iteration info. Currently used by <code>fireSense_SpreadFit</code> . Default <code>getOption("spades.folderWithIterInFilename", NULL)</code> .
<code>activeRunningPath</code>	Directory for "running" flag files written while a job is active. Must be cleaned up manually if a job crashes without removing its flag. Default: <code>file.path("logs/", basename(queue_path))</code> .
<code>continue</code>	Logical. Reserved for future single-shot mode; currently ignored.
<code>queue_path</code>	Character. Path to the .rds queue file. Defaults to <code>file.path(dirname(global_path), "tmux_queue.rds")</code> .

on_interrupt	"requeue" (default) or "fail". Action when a job is interrupted (Ctrl-C): requeue it for another worker, or mark it INTERRUPTED and stop this worker. A job that errors is marked "ERROR" and is not claimed again until its status is set back to "PENDING"; a status set by the user while the job ran (e.g. "HOLD") is kept.
on_error	"stop" (default) or "continue". What a worker does after a job errors. "stop" leaves the worker at the prompt with the error, call stack and traceback (<code>lastTraceback()</code>) available. "continue" prints the error and goes on to the next claimable job. Either way the row is marked "ERROR" with <code>last_error</code> .
pane_mode	"killAndNewPane" (default) or "reuse". See Worker loop modes above.
ss_id	Optional Google Drive spreadsheet/folder ID for live status syncing via googlesheets4. NULL disables syncing.
forceLocalQueueToGS	Logical. If TRUE, overwrite the Google Sheet queue with the local df even if the sheet already contains rows. Default FALSE.
onExistingQueue	What to do when <code>queue_path</code> already exists and <code>df</code> is supplied. "resume" (default) keeps the existing queue – preserving its DONE/RUNNING state – and warns if <code>df</code> holds rows it does not; "append" adds those rows as PENDING; "rebuild" discards the existing queue and starts from <code>df</code> . See tmuxReconcileQueueWithDF() .
enableGSSync	Logical. If TRUE, start an additional tmux pane that periodically syncs the local queue file to a Google Sheet (requires <code>ss_id</code>). Default FALSE.
email	Optional email address for gargle/Google OAuth authentication.
cache_path	Optional path to the gargle OAuth token cache directory.
workersToMonitor	Character vector of pane titles to monitor (currently unused).
runNameLabel	A quoted expression evaluated against the <code>queue_data.frame</code> to produce a human-readable job label used in log files and Google Sheet status updates.
copyModules	Logical. If TRUE and remote hosts are present, rsyncs the directory given by <code>getOption("spades.modulePath")</code> to the same absolute path on each remote host before workers start. Issues a warning and skips if the option is unset. Default FALSE.
sync_library	NULL (default) or a character vector of package specifications for <code>Require::Install()</code> . If given, syncProjectLibrary() runs once, now, in a fresh process and only while no worker is alive, before any worker starts. Never sync per job.
snapshot_library	Logical (default TRUE). Each worker session takes a hardlinked snapshot of the project library before loading anything from it, so an install into the shared library cannot corrupt a running job. See librarySnapshot .
...	Additional arguments passed to <code>.setup_remote_machine()</code> .

Value

Invisibly returns a character vector of tmux pane IDs for the spawned workers. Pass these to `tmuxKillPanes()` to tear down all workers at once.

Related tmux helpers

Function	Purpose
<code>tmuxPrepareQueueFromDF()</code>	Build a file-backed queue RDS from a data.frame of runs
<code>tmuxRunNextWorker()</code>	Claim and run one queued job in the current R session
<code>tmuxRunWorkerLoop()</code>	Loop of <code>tmuxRunNextWorker()</code> inside a worker pane
<code>tmuxRefreshQueueStatus()</code>	Re-evaluate job status from output files and heartbeats
<code>tmuxMirrorQueueToSheets()</code>	Mirror a local queue RDS to a Google Sheet
<code>tmuxListPanes()</code>	List every pane across every tmux server on this machine
<code>tmuxFindDuplicates()</code>	Surface panes running the same job (duplicate claims)
<code>tmuxSetPaneTitle()</code>	Rewrite a pane's title by matching its current title
<code>tmuxKillPanes()</code>	Kill a set of panes by ID (tear-down)
<code>tmuxSetMouse()</code>	Enable or disable tmux mouse mode
<code>tmuxActiveRunningPath()</code>	Default path for per-run "active" flag files
<code>localHostLabel()</code>	Short cluster alias for this machine (/etc/hosts lookup)

Examples

```
## Not run:
# --- Minimal: build a tiny global.R, then run a 2 x 2 experiment ---
tdir <- file.path(tempdir(), "experimentTmux-demo")
dir.create(tdir, showWarnings = FALSE, recursive = TRUE)
writeLines(
  'message("scenario=", .scenario, " rep=", .rep); Sys.sleep(2)',
  file.path(tdir, "global.R")
)
expt <- expand.grid(scenario = c("A", "B"), .rep = 1:2,
  stringsAsFactors = FALSE)

workers <- experimentTmux(
  df          = expt,
  global_path = file.path(tdir, "global.R"),
  cores       = rep("localhost", 2L),
  queue_path  = file.path(tdir, "queue.rds")
)

# --- Live inspection while panes run ---
experimentMonitor()          # tmux pane scan (no args)
experimentMonitor(stats = TRUE) # adds CPU / RAM / state per pane
tmuxListPanes()              # alias of experimentMonitor()
queueRead(file.path(tdir, "queue.rds")) # full queue snapshot
tmuxFindDuplicates(workers)    # any double-claimed jobs?
tmuxRefreshQueueStatus(file.path(tdir, "queue.rds")) # reset stuck rows

# --- Basic local usage with explicit pane sizing ---
workers <- experimentTmux(
  global_path = "/abs/path/to/global.R",
  queue_path  = "/abs/path/to/queue.rds",
```

```

    n_workers      = 4,
    pane_mode      = "killAndNewPane",
    delay_before_source = 60,
    stagger_by     = 60,
    set_mouse      = TRUE
  )

# --- Mixed local + remote ---
# Runs 2 workers on localhost and 2 on remote host "sbw".
# .setup_remote_machine("sbw", ...) is called automatically before workers start.
workers <- experimentTmux(
  global_path = "/abs/path/to/global.R",
  queue_path  = "/abs/path/to/queue.rds",
  cores       = c("localhost", "localhost", "sbw", "sbw"),
  pane_mode   = "killAndNewPane",
  email       = "you@example.com",
  cache_path  = "/abs/path/to/.secret",
  ss_id       = "your-google-sheet-id"
)

# --- Tear down all workers ---
tmuxKillPanes(workers)

# --- Restart a single broken pane ---
# In the broken pane, press Up then Enter to re-run the last command.

## End(Not run)

```

factorialDesign

Build a factorial experiment design

Description

Extracts the "all meaningful combinations" factorial-design logic that used to live inside `experiment()`. Given a `simList` plus lists of alternative params / modules / inputs / objects, it returns one row per run. Values are stored as *indices* into the supplied alternatives (because an alternative may itself be a vector and so cannot live in a single data.frame cell); column names are `module.parameter`, plus a modules index, an `expLevel`, and (when relevant) input, object and replicate columns.

Usage

```
factorialDesign(sim, params, modules, objects = list(), inputs, replicates = 1)
```

Arguments

<code>sim</code>	A <code>simList</code> , acting as the basis for the experiment.
<code>params</code>	Like for <code>SpaDES.core::simInit()</code> , but for each parameter, provide a list of alternative values.
<code>modules</code>	Like for <code>SpaDES.core::simInit()</code> , but a list of module names (as strings).

objects	Like for SpaDES.core::simInit() , but a list of named lists of named objects.
inputs	Like for SpaDES.core::simInit() , but a list of inputs data.frames.
replicates	The number of replicates to run of the same simList.

Details

This is the engine behind [experiment\(\)](#). It is exported so the same design can also seed the file-queue [experiment_family](#) ([experimentFuture\(\)](#) etc.): map each row's indices back to values to build their df.

Value

A data.frame, one row per run.

See Also

[experiment\(\)](#), [experiment_family](#)

fdSelectLimit	<i>FD_SETSIZE on this platform (the select() ceiling).</i>
---------------	--

Description

Hard-coded to 1024 – the value compiled into glibc and used by R's socket layer on Linux, macOS, and most BSDs. Not user-configurable without rebuilding R.

Usage

```
fdSelectLimit()
```

Value

integer scalar.

findProjectPath	<i>Find the project root directory</i>
-----------------	--

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.

get_sim_year_heartbeat	<i>Heartbeat for year-checkpoint SpaDES simulations</i>
------------------------	---

Description

Scans an output directory for files matching the pattern <file_prefix>_year<XXXX>.<ext> (e.g. cohortData_year2920.rds) and returns the furthest simulation year reached, the wall-clock elapsed time since the first checkpoint, and a percentage-complete estimate.

Usage

```
get_sim_year_heartbeat(  
  output_path,  
  start_year = NULL,  
  end_year = NULL,  
  file_prefix = "cohortData"  
)
```

Arguments

output_path	Character. Directory to scan for checkpoint files.
start_year	Integer or NULL. Expected start year of the simulation. If NULL (default), inferred as the minimum year found in output_path.
end_year	Integer or NULL. Expected end year. If NULL (default), inferred as the maximum year found in output_path (i.e. 100 % is reported only once the final checkpoint exists). Supply a value (e.g. 3020L) to get a meaningful percentage before the run completes.

file_prefix Character. Only files whose basename begins with this prefix are used as checkpoint indicators. Defaults to "cohortData" because that file is written **after** all others at each SpaDES save event, making it the most reliable completion signal.

Value

A named list with elements:

ts Character. Modification timestamp of the latest checkpoint file.

iter Integer. Simulation year of the latest checkpoint.

started Character. Modification timestamp of the first checkpoint file.

elapsed difftime. Wall-clock time between first and latest checkpoint.

pct_complete Numeric 0-100. Percentage of the simulation completed, or NA if start_year == end_year.

All elements are NA / NA_character_ when no matching files are found.

Examples

```
## Not run:
hb <- get_sim_year_heartbeat(
  output_path = "outputs/6.5/1991-2020/NRV_ssp370/rep1",
  end_year     = 3020L
)
message("Year: ", hb$iter, " (", hb$pct_complete, "%) -- last checkpoint: ", hb$ts)

## End(Not run)
```

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 <i>GitAccount/GitRepo@Branch/File</i> , 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	<i>Simple function to download a SpaDES module as GitHub repository</i>
-----------	---

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: <code>file.path(modulePath, repository)</code> . If omitted, and <code>options(spades.modulePath)</code> is set, it will use <code>getOption("spades.modulePath")</code> , otherwise it will use <code>"."</code> .
overwrite	A logical vector of same length (or length 1) <code>gitRepo</code> . If <code>TRUE</code> , then the download will delete any existing folder with the same name as the repository provided in <code>gitRepo</code>
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or <code>FALSE</code> , then minimal outputs; if 1 or <code>TRUE</code> , more outputs; 2 even more. NOTE: in <code>Require</code> function, when <code>verbose >= 2</code> , also returns details as if <code>returnDetails = TRUE</code> (for backwards compatibility).

Details

A parent module lists its children in `childModules`. After a module is fetched (or found locally), its children are fetched too, recursively: an entry written as `"owner/repo@branch"` is fetched as written; a plain name (or `"name@branch"`) is fetched from the parent's GitHub account (and branch, unless one is given). When the parent is fetched at a version tag (e.g. `"owner/parent@v1.1.0"`), a plain-named child is fetched at `v<version>` instead, its version taken from the parent's own version list at that tag (e.g. `version = list(parent = "1.1.0", child = "2.2.0")` gives `child@v2.2.0`), so one parent release names the release of every child. The same holds when the parent has no ref (`"owner/parent"`): it comes from its default branch, normally its latest release, and its children at the releases its list names. A child missing from that list, or listed at a development version (four components, e.g. `"2.2.0.9000"`), falls back to the parent's ref. A module also named in `modules` is fetched only as it is written there, never from a parent's entry.

Value

A list with success and failed, the module specifications (children included) that are, or are not, available locally.

See Also

[getGithubFile](#)

```
initialize,simLists-method
```

Generate a simLists object

Description

Given the name or the definition of a class, plus optionally data to be included in the object, `new` returns an object from that class.

Usage

```
## S4 method for signature 'simLists'
initialize(.Object, ...)
```

Arguments

<code>.Object</code>	A <code>simList</code> object.
<code>...</code>	Optional Values passed to any or all slot

`killExperimentFuture` *Stop workers launched by `experimentFuture`*

Description

Two modes are available:

Graceful (`force = FALSE`, default): creates a per-worker sentinel file. Each worker checks for this file between jobs and exits cleanly once its current job finishes. Remaining PENDING jobs stay in the queue and are picked up automatically when `experimentFuture` is called again with the same `queue_path`.

Immediate (`force = TRUE`): sends SIGTERM to each live worker, causing the process to exit as soon as possible. Because callr workers run non-interactively, the process typically exits before R's interrupt handler has a chance to update the queue. Any jobs that were RUNNING at the time of the kill will remain as RUNNING in the queue until the next reclaim pass. Call `tmuxRefreshQueueStatus(ef$queue_path)` afterwards to reset stale RUNNING entries to PENDING (or INTERRUPTED if the run had already written a heartbeat), or use the GS backend which reclaims dead workers automatically before each new claim.

Usage

```
killExperimentFuture(ef, force = FALSE)
```

Arguments

<code>ef</code>	An "experimentFuture" object returned by <code>experimentFuture</code> .
<code>force</code>	If FALSE (default), signal workers via stop files so they exit after their current job completes. If TRUE, send SIGINT to each live worker for an immediate but clean stop.

Value

`ef`, invisibly.

See Also

`experimentFuture`, `awaitExperimentFuture`

killExperimentSBATCH *Stop SBATCH workers launched by experimentSBATCH*

Description

Graceful (force = FALSE, default): creates per-worker stop files; each worker exits cleanly between jobs once it observes its file. Slurm jobs end normally; remaining PENDING rows stay in the queue and can be resumed with another [experimentSBATCH](#) call against the same queue_path.

Usage

```
killExperimentSBATCH(es, force = FALSE, scancel_cmd = "scancel")
```

Arguments

es	An "experimentSBATCH" object.
force	FALSE (graceful) or TRUE (scancel).
scancel_cmd	Path to scancel; defaults to "scancel" on \$PATH.

Details

Immediate (force = TRUE): runs scancel <ids> to kill the Slurm jobs straight away. Any rows that were RUNNING at the time of cancellation will remain RUNNING in the queue until the next reclaim pass; clean them up with `tmuxRefreshQueueStatus(es$queue_path)`.

Value

es, invisibly.

See Also

[experimentSBATCH](#), [awaitExperimentSBATCH](#)

lastTraceback *Inspect the call stack from the most recent worker error*

Description

Worker panes started by `experimentTmux()` / `tmuxRunNextWorker()` evaluate the user's global .R inside a fresh scenario environment (not .GlobalEnv). When the source call errors, the package captures `sys.calls()` and stashes it on that scenario env so a post-mortem traceback is still possible without polluting the user's global state. Use this accessor to retrieve it.

Usage

```
lastTraceback()
```

Value

A list of calls (as from `sys.calls()`) suitable for passing to `base::traceback()`; NULL if no error has been captured in the current session.

Examples

```
## Not run:
# After a worker pane errors:
traceback(SpaDES.project::lastTraceback())

## End(Not run)
```

librarySnapshot	<i>Keep a shared project library safe under concurrent experiment workers</i>
-----------------	---

Description

Every experiment backend (`experimentTmux()`, `experimentFuture()`, `experimentSBATCH()`) runs many jobs against one project library. Nothing stops that library from being modified while jobs use it, and R lazy-loads package objects from `R/<pkg>.rdb` by file offset, so replacing a package under a running session ends in lazy-load database `'.../R/<pkg>.rdb'` is corrupt. Two functions make the library safe:

- `syncProjectLibrary()` installs or updates packages **once per launch**, in a fresh R process (this session may itself have them loaded), and refuses to run while any worker is alive. The backends call it when given `sync_library = <package specs>`.
- `snapshotLibrary()` gives **one job** its own copy of the library, made of hardlinks (`file.link()`, so it costs seconds and no disk), falling back to `file.copy()` across filesystems. A hardlink keeps the old inode alive: an install into the shared library replaces files *there* and cannot touch what the running job has open. Every job thereby also records exactly which package versions it ran with. The backends take the snapshot in the worker startup script, before any package is loaded, when `snapshot_library = TRUE` (the default).

`libraryInUse()` reports the workers that make a sync unsafe; `releaseLibrarySnapshot()` removes one snapshot (the worker loop does this at exit) and `sweepLibrarySnapshots()` removes those whose owning process is gone.

Usage

```
syncProjectLibrary(
  packages,
  libPath = .libPaths()[1],
  activeRunningPath = getOption("spades.activeRunningPath"),
  requireIdle = TRUE,
  installFun = NULL,
  verbose = getOption("Require.verbose", 1)
)
```

```

libraryInUse(activeRunningPath = getOption("spades.activeRunningPath"))

snapshotLibrary(
  libPath = .libPaths()[1],
  where = getOption("spades.activeRunningPath"),
  id = paste0(Sys.info()[["nodename"]], "_", Sys.getpid()),
  setLibPaths = TRUE
)

releaseLibrarySnapshot(snapshot = Sys.getenv("SPADES_PROJECT_LIB_SNAPSHOT"))

sweepLibrarySnapshots(where = getOption("spades.activeRunningPath"))

```

Arguments

packages	Character vector of package specifications as accepted by <code>Require::Install()</code> , e.g. <code>"PredictiveEcology/reproducible@development (>= 3.2.1.9010)"</code> . Usually the vector given to <code>setupProject(packages =)</code> , so the version floors declared there are the single source of truth.
libPath	The shared library. Defaults to <code>.libPaths()[1]</code> .
activeRunningPath	Where the experiment keeps its <code>Running_*</code> sentinels; snapshots are created there as <code>lib_<id>/</code> unless <code>where</code> says otherwise.
requireIdle	Logical. Refuse to install while <code>libraryInUse()</code> finds a worker. Set to <code>FALSE</code> only when you know the workers are idle.
installFun	Function of <code>(packages, libPath)</code> performing the install; defaults to <code>Require::Install()</code> . Exposed for tests.
verbose	Numeric.
where	Directory holding snapshots; defaults to <code>activeRunningPath</code> .
id	Snapshot id; defaults to <code><nodename>_<pid></code> so <code>sweepLibrarySnapshots()</code> can tell a live owner from a dead one.
setLibPaths	Logical. Put the snapshot first on <code>.libPaths()</code> in place of <code>libPath</code> , and record it in <code>SPADES_PROJECT_LIB_SNAPSHOT</code> so the worker loop can release it at exit.
snapshot	A directory returned by <code>snapshotLibrary()</code> .

Value

`syncProjectLibrary()` returns `TRUE` invisibly after installing, `FALSE` when `packages` is empty. `libraryInUse()` returns a character vector describing live workers (sentinel files and `ps` lines), empty when none. `snapshotLibrary()` returns the snapshot directory. `releaseLibrarySnapshot()` returns `TRUE` invisibly if it removed one. `sweepLibrarySnapshots()` returns the directories it removed.

Examples

```
## Not run:
# before launching an experiment, with no worker running:
syncProjectLibrary(c("PredictiveEcology/reproducible@development (>= 3.2.1.9010)",
                    "PredictiveEcology/SpaDES.core@development"))
# inside a worker, before loading anything from the library:
snapshotLibrary(where = "logs/my_queue.rds")

## End(Not run)
```

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")
```

localHostLabel	<i>Short friendly name for the local machine</i>
----------------	--

Description

Resolves the cluster-facing short name for this host, trying in order:

1. /etc/hosts lookup by a local IP (shortest alias wins);
2. ~/.ssh/config Host entry whose Hostname is a local IP (CRLF-safe);
3. hostname -s.

Usage

```
localHostLabel()
```

Details

Useful for deriving the pane-title host prefix when the cluster knows this machine by a name different from `hostname -s` (e.g. mega, whose raw hostname is the node id but whose cluster alias is mega via /etc/hosts).

Value

Character(1) short name, or NULL if none could be determined.

makeDESCRIPTIONproject	<i>Make DESCRIPTION file(s) from SpaDES module metadata</i>
------------------------	---

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 <code>modulePath()</code> or <code>paths\$modulePath</code>
projectPath	Character. Only used if <code>singleDESCRIPTION = TRUE</code>
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 <code>modulePath</code> (if <code>singleDESCRIPTION = FALSE</code>) or <code>projectPath</code> (if <code>singleDESCRIPTION = TRUE</code>)
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 <code>Require</code> function, when <code>verbose >= 2</code> , also returns details as if <code>returnDetails = TRUE</code> (for backwards compatibility).
date	Date to enter into DESCRIPTION file. Defaults to <code>Sys.Date()</code>
metadataList	The parsed source code from a module. Must include <code>defineModule</code> metadata.
...	Currently not used.

Value

Invisibly, the path(s) of the DESCRIPTION file(s) written.

openFds	<i>Inspect open file descriptors.</i>
---------	---------------------------------------

Description

On Linux, reads `/proc/self/fd` and resolves each symlink to identify what the fd points to. Returns a `data.frame` with one row per open fd; useful for diagnosing the cryptic "file descriptor is too large for select()" failure from `parallelly::makeClusterPSOCK()`.

Usage

```
openFds()
```

Details

Returns an empty `data.frame` on non-Linux systems or if `/proc/self/fd` is unreadable.

Value

`data.frame` with columns `fd` (integer), `target` (character, with any Linux (deleted) suffix preserved), and `bucket` (character holder category: "socket", "pipe", "terra scratch", "terra scratch (deleted)", "tif (other)", "vrt", "sqlite", "qs/qs2", "anon_inode", "other file", "unknown").

See Also

`openFdsReport()` for a printable summary; `fdSelectLimit()`.

openFdsReport	<i>Printable summary of open file descriptors.</i>
---------------	--

Description

Wraps `openFds()` and returns a multi-line string suitable for warning or error messages. By default summarizes only fds at or above `fdSelectLimit()` (the `select()` failure threshold); pass `threshold = 0L` to summarize everything.

Usage

```
openFdsReport(threshold = fdSelectLimit())
```

Arguments

threshold integer. Only fds at or above this number are bucketed. Defaults to `fdSelectLimit()`.

Value

character scalar; "" if /proc/self/fd is not available.

Examples

```
cat(openFdsReport())
cat(openFdsReport(threshold = 0L))
```

outList	<i>List uploaded scenario output archives.</i>
---------	--

Description

List uploaded scenario output archives.

Usage

```
outList(folder, pattern = "\\\\.tar\\.gz$")
```

Arguments

folder Folder URL or dribble of the upload folder.

pattern Regex matched against the name column. NULL disables filtering.

Value

A dribble of the matching files.

See Also

`outScenarios()`, `queueUploadMissing()`

outSave

*Save a SpaDES simulation to an RDS file***Description**

Saves a `simList` to an RDS file via `SpaDES.core::saveSimList()`. Heavy ancillary data (inputs, outputs, cache, files) are excluded so the file contains only the simulation state; pair with `outTar()` to bundle the output files separately.

Usage

```
outSave(
  sim,
  runName,
  simFilename = NULL,
  lazy = TRUE,
  verbose = getOption("reproducible.verbose", 1)
)
```

Arguments

<code>sim</code>	A <code>simList</code> object.
<code>runName</code>	Character scalar. Used as the base name for the saved sim file and tarball.
<code>simFilename</code>	Character scalar. Full path for the <code>.rds</code> file. Defaults to <code>SpaDES.core::simFile(name = runName, path = outputPath(sim), time = end(sim), ext = "rds")</code> .
<code>lazy</code>	Logical. Passed to <code>SpaDES.core::saveSimList()</code> . When TRUE (default), each user object in <code>sim@.xData</code> and each module's mod objects in <code>sim@.xData\$.modObjs</code> are written to their own file in a sibling <code><simFilename sans ext>_lazy/</code> directory, and lazily restored on load via <code>delayedAssign</code> . <code>outTar()</code> picks up that directory automatically.
<code>verbose</code>	Numeric message level, forwarded to <code>SpaDES.core::saveSimList()</code> . Defaults to <code>getOption("reproducible.verbose", 1)</code> – note the fallback: <code>saveSimList()</code> resolves its own default as a bare <code>getOption("reproducible.verbose")</code> , and passes the result to <code>Require::messageVerbose()</code> , whose if (<code>verbose >= verboseLevel</code>) fails with "argument is of length zero" whenever that option happens to be unset.

Value

Invisibly returns `simFilename`.

See Also

`outTar()`, `outUpload()`, `outSaveTarUpload()`

outSaveTarUpload	<i>Save, tar, and upload a SpaDES simulation to Google Drive</i>
------------------	--

Description

Convenience wrapper that calls [outSave\(\)](#), [outTar\(\)](#), and [outUpload\(\)](#) in sequence. The sim is saved to an RDS file, bundled with its output files into a `.tar.gz` archive, and the archive is uploaded to a Google Drive folder.

Usage

```
outSaveTarUpload(
  runName,
  sim,
  gFolder = NULL,
  simFilename = NULL,
  tarDir = NULL,
  tarball = NULL,
  overwrite = TRUE,
  cleanup = FALSE,
  verbose = TRUE,
  lazy = TRUE
)
```

Arguments

runName	Character scalar. Used as the base name for the saved sim file and tarball.
sim	A <code>simList</code> object.
gFolder	A Google Drive folder identifier accepted by googledrive::drive_upload() – a dribble, a Drive URL, or a bare folder ID from googledrive::as_id() .
simFilename	Character scalar. Full path for the <code>.rds</code> file. Defaults to <code>SpaDES.core::simFile(name = runName, path = outputPath(sim), time = end(sim), ext = "rds")</code> .
tarDir	Character scalar. Directory in which to create the tarball. Defaults to <code>dirname(simFilename)</code> .
tarball	Character scalar. Path to the local file to upload.
overwrite	Logical. Overwrite an existing file of the same name in the Drive folder. Default <code>TRUE</code> .
cleanup	Logical. Delete the local tarball after a successful upload. Default <code>FALSE</code> .
verbose	Numeric message level, forwarded to SpaDES.core::saveSimList() . Defaults to <code>getOption("reproducible.verbose", 1)</code> – note the fallback: <code>saveSimList()</code> resolves its own default as a bare <code>getOption("reproducible.verbose")</code> , and passes the result to <code>Require::messageVerbose()</code> , whose <code>if (verbose >= verboseLevel)</code> fails with "argument is of length zero" whenever that option happens to be unset.

`lazy` Logical. Passed to `SpaDES.core::saveSimList()`. When TRUE (default), each user object in `sim@.xData` and each module's mod objects in `sim@.xData$.modObjs` are written to their own file in a sibling `<simFilename sans ext>_lazy/` directory, and lazily restored on load via `delayedAssign`. `outTar()` picks up that directory automatically.

Value

Invisibly returns the dribble from `googledrive::drive_upload()`.

See Also

`outSave()`, `outTar()`, `outUpload()`

outScenarios	<i>Uploaded outputs as scenario records.</i>
--------------	--

Description

Uploaded outputs as scenario records.

Usage

```
outScenarios(folder, pattern = "\\..tar\\.gz$")
```

Arguments

folder	Folder URL or dribble of the upload folder.
pattern	Regex matched against the name column. NULL disables filtering.

Value

A list of scenario objects.

See Also

`outList()`, `as_scenario()`

outTar	<i>Bundle a sim file and output files into a tar.gz archive</i>
--------	---

Description

Creates a `.tar.gz` archive containing `simFilename` and any additional `outputFiles`. Files that do not exist are silently skipped so a partially completed simulation can still be archived.

Usage

```
outTar(
  simFilename,
  outputFiles = character(0),
  runName,
  tarDir = dirname(simFilename),
  verbose = TRUE
)
```

Arguments

<code>simFilename</code>	Character scalar. Full path for the <code>.rds</code> file. Defaults to <code>SpaDES.core::simFile(name = runName, path = outputPath(sim), time = end(sim), ext = "rds")</code> .
<code>outputFiles</code>	Character vector of additional files to include (e.g. <code>SpaDES.core::outputs(sim)\$file</code>). Non-existent paths are dropped. Default <code>character(0)</code> .
<code>runName</code>	Character scalar. Used as the base name for the saved sim file and tarball.
<code>tarDir</code>	Character scalar. Directory in which to create the tarball. Defaults to <code>dirname(simFilename)</code> .
<code>verbose</code>	Logical. Pass <code>-v</code> to <code>tar</code> for file-by-file progress. Default <code>TRUE</code> .

Value

Invisibly returns the path to the created tarball.

See Also

[outSave\(\)](#), [outUpload\(\)](#), [outSaveTarUpload\(\)](#)

outUpload	<i>Upload a file to Google Drive</i>
-----------	--------------------------------------

Description

Uploads a local file (typically a tarball produced by `outTar()`) to a Google Drive folder via `googledrive::drive_upload()`.

Usage

```
outUpload(tarball, gFolder, overwrite = TRUE, cleanup = FALSE)
```

Arguments

tarball	Character scalar. Path to the local file to upload.
gFolder	A Google Drive folder identifier accepted by <code>googledrive::drive_upload()</code> – a dribble, a Drive URL, or a bare folder ID from <code>googledrive::as_id()</code> .
overwrite	Logical. Overwrite an existing file of the same name in the Drive folder. Default TRUE.
cleanup	Logical. Delete the local tarball after a successful upload. Default FALSE.

Value

Invisibly returns the dribble returned by `googledrive::drive_upload()`.

See Also

`outSave()`, `outTar()`, `outSaveTarUpload()`

packagesInModules	<i>Extract element from SpaDES module metadata</i>
-------------------	--

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.

pathBuild	<i>Default scenario path builder.</i>
-----------	---------------------------------------

Description

Generic format: each non-empty field's value becomes one path segment. Field order is taken from the order of the input. A positional call is named by its own argument symbols when those name the cached `scenarioFields()` in any order, and by `scenarioFields()` otherwise. Integer-and-contiguous vectors are encoded as start-end. Empty / NA fields are dropped entirely (yielding one fewer segment); for round-tripping see `pathParse()`.

Usage

```
pathBuild(..., pre = "outputs", withFieldLabel = .scenario_env$withFieldLabel)
```

Arguments

...	Either a single named-list / scenario, or named/positional field/value pairs.
pre	Path prefix (default "outputs").
withFieldLabel	Character vector of field names whose segment should carry a paste0(label, value) prefix. Defaults to the registered value (see <code>register_scenario_format()</code>).

Details

Fields whose name appears in withFieldLabel get their segment prefixed by the field name itself (e.g. .rep with value 5 renders as .rep5 instead of bare 5). Useful when path readers must distinguish two integer fields, or when round-tripping with mid-list NAs (the label disambiguates which segments are present).

Accepts three calling styles, all equivalent:

- `pathBuild(scenarioObj)` — a single named-list / scenario;
- `pathBuild(.fieldA = vA, .fieldB = vB, ...)` — explicit named args;
- `pathBuild(vA, vB, ...)` — positional, in cached-field order.

Value

Character scalar.

pathParse	<i>Default scenario path parser.</i>
-----------	--------------------------------------

Description

Inverse of the default `pathBuild()`: splits the path on / (or, for tarname inputs, on `_`), strips archive extensions and the pre prefix, then matches segments **positionally** to `scenarioFields()`. Integer ranges of the form start-end decode to integer vectors.

Usage

```
pathParse(
  path,
  fields = scenarioFields(),
  pre = "outputs",
  withFieldLabel = .scenario_env$withFieldLabel
)
```

Arguments

<code>path</code>	A single character string (path or tarname).
<code>fields</code>	Field labels in scenario order; defaults to <code>scenarioFields()</code> (set by <code>queueRead()</code>).
<code>pre</code>	Path prefix to strip (default "outputs").
<code>withFieldLabel</code>	Character vector of field names that were built with <code>paste0(label, value)</code> prefixing.

Details

Without per-segment labels there is no way to recover which field a missing segment corresponds to, so when the path has fewer segments than there are fields, the *trailing* fields are treated as NA. Round-trip is therefore only lossless when NA-bearing fields are last in the field order *unless* you label the ambiguous fields through `withFieldLabel`: any field named there has its label prefix stripped from the segment, and segments not starting with a labeled field's name are assigned positionally to the next unlabeled field. With every potentially-NA field labeled, mid-list NAs round-trip cleanly.

Value

Named list of field values (in fields order).

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.

plotChangeOverTime

Difference between two layers of a time-series, as a leaflet map

Description

Subtracts the from layer from the to layer and plots the result on a single-layer leaflet map with a diverging palette centred on zero. The "change from start to finish" view that pairs with [plotTimeSeriesLeaflet\(\)](#).

Usage

```
plotChangeOverTime(
  x,
  from = NULL,
  to = NULL,
  palette = "differences",
  rev = TRUE,
  layerName = "change",
  legendPosition = "bottomright"
)
```

Arguments

x	One of: a multi-layer SpatRaster, a list of single-layer SpatRasters, a <code>simList</code> (reads <code><name>*.tif</code> from <code>SpaDES.core::outputPath(x)</code>), or a length-1 character directory path.
from, to	Names (or, if missing, first/last) of the layers to subtract. <code>result = x[[to]] - x[[from]]</code> .
palette	Palette name. Default "differences" – the dedicated blue->white->red diverging palette from <code>terra::map.pal()</code> , purpose-built for difference maps. Any other <code>terra::map.pal()</code> name (e.g. "viridis") is also accepted; if <code>terra::map.pal()</code> doesn't recognise the name, we fall back to <code>grDevices::hcl.colors()</code> (e.g. "RdBu", "Spectral").
rev	Reverse the palette? Default TRUE – with the "differences" palette this gives red = negative, blue = positive. Set FALSE to flip (the <code>terra::map.pal("differences")</code> native orientation: red = positive, blue = negative).
layerName	Short prefix for the per-layer GeoTIFF filename.
legendPosition	Where to place the legend on the map. One of "bottomright" (default), "bottomleft", "topright", "topleft".

Value

A leaflet htmlwidget showing the difference layer with a symmetric (zero-centred) colour scale. Static-safe.

See Also

`plotTimeSeriesLeaflet()` for the time-step viewer that complements this difference view.

plotSAs

*Plot studyArea** and rasterToMatch** with ggplot2 or leaflet*

Description

Plot all `studyArea**` and `rasterToMatch**` objects within a list-like object.

Usage

```

plotSAs(
  ll,
  ...,
  include = TRUE,
  exclude,
  saCols = c("purple", "blue", "green", "red"),
  title,
  rasterToMatchLabel = "Stand Age",
  rasterToMatchPalette = c("Set1", "Set2", "Set3"),
  country = "CAN",
  latlong = FALSE,
  minArea = 7e+11
)

plotSAsLeaflet(
  ll,
  ...,
  include = TRUE,
  exclude,
  saCols = c("purple", "blue", "green", "red"),
  title = "Study Areas",
  rasterToMatchLabel = "Stand Age",
  rasterToMatchPalette = c("Set1", "Set2", "Set3"),
  labelCols = c("ID", "Name", "Names")
)

```

Arguments

<code>ll</code>	Any list-like object with named elements. Names must include at least one that starts with <code>studyArea</code> or <code>rasterToMatch</code> . Thus any of the permutations like <code>studyAreaLarge</code> or <code>rasterToMatchPSP</code> all are fine.
<code>...</code>	Any objects to plot. Currently, they must be named arguments, and they must have prefixes <code>studyArea</code> or <code>rasterToMatch</code> to be visualized.
<code>include</code>	Either logical or a character vector. If logical, this indicates whether all maps in the <code>ll</code> object should be plotted (if <code>TRUE</code>) or, if <code>FALSE</code> , no extra maps (on top of the defaults listed in <code>ll</code> argument description. If a character vector, then the objects indicated will also be plotted. Default is <code>FALSE</code> to prevent inadvertent (slow) plotting of potentially many layers.
<code>exclude</code>	A character vector of spatial objects contained within <code>ll</code> to exclude from plotting. This is run after <code>include</code> , so it will override any named objects specified in <code>include</code> .
<code>saCols</code>	A vector of same length as number of <code>studyArea**</code> objects, that defines the <code>studyArea</code> polygon boundary colours. These will be used in sequence from largest to smallest in polygon area.
<code>title</code>	The main title for the <code>ggplot2</code> object. Defaults to one or both of <code>"studyArea"</code> and <code>"rasterToMatch"</code> or their plurals.

rasterToMatchLabel	Used in rasterToMatch legend
rasterToMatchPalette	A palette to be used for colour scheme in rasterToMatch plotting. Can be any that work with <code>tidyterra::whitebox.colors</code> .
country	The country for jurisdiction boundaries; defaults to "CAN". Passed to <code>geodata::gadm</code>
latlong	Logical. Should all layers be converted to latlong for plotSAs prior to plotting. This means that "North will be up"; this could be slow for large rasters. This happens by default with plotSAsLeaflet and can't be turned off.
minArea	In m^2 . This is the minimum area for the entire plot. If this is too small then the legislative boundaries may not appear. The area covered by the plot will be the maximum of the <code>studyArea**</code> or <code>rasterToMatch**</code> and this <code>minArea</code> value.
labelCols	plotSAsLeaflet only. Character vector of attribute-column names to look for, in order, when labelling the polygons of each <code>studyArea**</code> layer; the first one the layer actually has supplies the per-feature hover labels. A layer with none of them (or with no attribute table at all) falls back to its own name, as before.

Value

Run primarily for side effects. `plotSAs` plots (and returns) a `ggplot2` object. `plotSAsLeaflet` creates a leaflet page in a viewer (if using Rstudio).

`plotTimeSeriesLeaflet` *Time-series of SpatRaster layers as an interactive leaflet map with a slider*

Description

Takes a multi-layer `SpatRaster` (or a list of single-layer `SpatRasters`) and produces a self-contained leaflet `htmlwidget`. Each layer becomes a radio-selected base group; a draggable range slider is added that drives the radio buttons, so the user steps through years by dragging.

Usage

```
plotTimeSeriesLeaflet(
  x,
  palette = "viridis",
  rev = FALSE,
  layerName = "raster",
  sliderPosition = "bottomleft",
  legendPosition = "bottomright"
)
```

Arguments

x	One of: • a multi-layer SpatRaster – becomes one "object" named "raster" • a list of single-layer SpatRasters – same • a simList – reads ALL <obj>*.tif time-series from <code>SpaDES.core::outputPath(x)</code> • a length-1 character path to a directory of GeoTIFFs – same scan For the disk-scan cases, files are grouped by base name (last regex match = time tag) into separate "objects". Each object becomes a radio-selectable layer in the UI.
palette	Palette name. Default "viridis". Tried first with <code>terra::map.pal()</code> , falling back to <code>grDevices::hcl.colors()</code> .
rev	Reverse the palette? Default FALSE.
layerName	Short prefix used for each per-layer GeoTIFF filename (avoids collisions when multiple time-series sit on the same page).
sliderPosition	Leaflet control position for the object-radio + year-slider UI: "bottomleft" (default), "bottomright", "topleft", "topright".
legendPosition	Leaflet control position for the per-object colour legend. Default "bottomright".

Details

Designed to ship in a Quarto / knitr static render – uses `.leafletGeoTiffPath()` internally so the per-layer GeoTIFFs are written into the qmd's `_files/figure-html/` folder rather than `tempfile()`.

Value

A leaflet htmlwidget with the slider injected via `htmlwidgets::onRender()`. Ships as static HTML – no Shiny server needed.

See Also

`plotSAsLeaflet()` for the single-snapshot case.

preRunSetupProject	<i>Partially or Fully Run</i> setupProject
--------------------	--

Description

`preRunSetupProject` parses an R script (default: "global.R") and evaluates its contents up to the `setupProject()` call, either fully or partially based on the `upTo` argument. This is useful for initializing only certain parts of a project without executing the entire setup.

Usage

```
preRunSetupProject(file = "global.R", upTo = TRUE, envir = parent.frame())
```

Arguments

<code>file</code>	Character string. Path to the R script containing the setup code. Defaults to "global.R".
<code>upTo</code>	Character or logical. If TRUE, evaluates all code up to and including the first <code>setupProject()</code> call within <code>file</code> . If a character string, only evaluates the code up to the <code>setupProject</code> plus the arguments up to the <code>upTo</code> named argument. Defaults to "paths" so that paths will be evaluated and available to use.
<code>envir</code>	The environment where the function should be finding objects. Defaults to <code>parent.frame()</code> so it can find them in the calling frame.

Details

The function:

1. Parses the specified file using `parse()`.
2. Identifies the line where `setupProject()` is called.
3. Evaluates all code before the `setupProject()` call.
4. Depending on `upTo`, evaluates either the full call or a subset of its arguments.

This allows selective initialization of project components for debugging or partial setup in large projects.

Value

The evaluated result of the executed portion of `setupProject()`. i.e., a list returned by `setupProject()`.

See Also

[setupProject](#)

Examples

```
## Not run:
# Run file up to and including the setupProject, but only to the 'paths' argument
result <- preRunSetupProject(file = "global.R", upTo = "paths")

# Run file up to and including full setupProject()
result <- preRunSetupProject(file = "global.R", upTo = TRUE)

## End(Not run)
```

queueRead

*Read the driver queue (local RDS or Google Sheet).***Description**

Two call shapes:

Usage

```
queueRead(folder, name, sheet = NULL, col_types = "c")
```

Arguments

folder	Either a local path to an .rds queue file (when name is missing), or a Drive folder URL / dribble of the parent folder containing the queue spreadsheet.
name	Spreadsheet name (exact match) within folder. Omit for local-RDS reads.
sheet	Optional worksheet/tab name (passed to read_sheet()). Ignored for local-RDS reads.
col_types	Column-types spec for read_sheet() (default "c"). Ignored for local-RDS reads.

Details

Local: queueRead("path/to/queue.rds") When the first argument is an existing local .rds file and name is not supplied, the queue is loaded via readRDS(). Useful for the file-backed queues written by experimentTmux() / experimentFuture() / experimentSBATCH() when no ss_id was supplied.

Google Sheet: queueRead(folder, name) Convenience wrapper around googledrive::drive_ls() + googlesheets4::read_sheet(). folder is the Drive folder URL/id, name is the spreadsheet name within it.

Either way the result is passed through revertDotNames() so callers see canonical .ELFind/.GCM/... column names rather than the dotELFind/dotGCM/... names Google Sheets forces. As a side effect, the non-meta column names are cached as the active scenario field set (see [scenarioFields\(\)](#)).

Value

A data.table. Pipe through [as_scenario\(\)](#) for scenario records.

See Also

[queueUploadMissing\(\)](#), [outList\(\)](#), [outScenarios\(\)](#), [experimentFuture\(\)](#), [experimentTmux\(\)](#), [experimentSBATCH\(\)](#)

queueUploadMissing	<i>Queue rows whose tarball is missing from the upload folder.</i>
--------------------	--

Description

Anti-join of the driver queue against the upload folder's .tar.gz listing, keyed on rendered tarname (see [as_tarname\(\)](#)). Independent of the queue's status column.

Usage

```
queueUploadMissing(folder, name, uploadFolder, ...)
```

Arguments

folder	Folder URL of the <i>queue</i> (driver) Drive folder.
name	Queue spreadsheet name within folder.
uploadFolder	Folder URL of the <i>upload</i> Drive folder.
...	Extra args forwarded to queueRead() .

Value

Subset of the queue data.table for rows whose expected tarball is not present in uploadFolder.

See Also

[queueRead\(\)](#), [outList\(\)](#)

reGet	<i>Download tarballs from Google Drive</i>
-------	--

Description

Inverse of [outUpload\(\)](#). Downloads one or more tar.gz archives from a Google Drive folder to a local directory, using [reproducible::preProcess\(\)](#) (so re-runs hit the local copy when present). Vectorised: typically called with the multi-row dribble returned by [outList\(\)](#) / [outScenarios\(\)](#).

Usage

```
reGet(gFiles, destDir, overwrite = FALSE, verbose = TRUE)
```

Arguments

gFiles	Either a Google Drive dribble (e.g. the output of <code>outList()</code> / <code>outScenarios()</code>) or a character vector of Drive file IDs or URLs.
destDir	Character scalar. Local directory to write tarballs into. Created if it does not exist.
overwrite	Logical. Force re-download even if the local file exists. Default FALSE.
verbose	Logical. Print elapsed time per download. Default TRUE.

Value

A data.table with columns name and local_path, one row per downloaded file.

See Also

`reUntar()`, `reload()`, `reGetUntarLoad()`, `outUpload()`

reGetUntarLoad

Download, untar, and load SpaDES sims from Google Drive

Description

Convenience wrapper around `reGet()`, `reUntar()`, and `reload()` – the inverse of `outSaveTarUpload()`. Operates on a batch: typically called with the multi-row dribble returned by `outList()` / `outScenarios()`.

Usage

```
reGetUntarLoad(
  gFiles,
  destDir,
  pathRemap = NULL,
  projectPath = getwd(),
  method = c("loadSimList", "readRDS"),
  parse = TRUE,
  remote = FALSE,
  overwrite = FALSE,
  verbose = TRUE,
  skipExisting = TRUE,
  useCache = FALSE,
  pattern = NULL
)
```

Arguments

<code>gFiles</code>	Either a Google Drive dribble (e.g. the output of <code>outList()</code> / <code>outScenarios()</code>) or a character vector of Drive file IDs or URLs.
<code>destDir</code>	Character scalar. Local directory to write tarballs into. Created if it does not exist.
<code>pathRemap</code>	Optional named character vector of length 2, <code>c(old = "/old/prefix", new = "/new/prefix")</code> , applied to all tarballs. If NULL (default), files are extracted to their original absolute paths (<code>tar --absolute-names</code>).
<code>projectPath</code>	Character scalar. Passed to <code>SpaDES.core::loadSimList()</code> for relative-path resolution. Default <code>getwd()</code> .
<code>method</code>	One of "loadSimList" (default) or "readRDS".
<code>parse</code>	Logical. If TRUE (default), module source code is re-parsed on load. FALSE forwards <code>parse = FALSE</code> to <code>SpaDES.core::loadSimList()</code> , which skips it – worthwhile because reparsing dominates the load time of a lazily saved <code>simList</code> (on a 19-module simulation, ~7 s of a ~9.5 s load). Objects are unaffected: user objects and each module's mod objects are still bound lazily. The result is inspect-only – it has no module code, so it cannot be passed to <code>SpaDES.core::spades()</code> . Ignored when <code>method = "readRDS"</code> .
<code>remote</code>	Logical. If FALSE (default), the archive is downloaded and untarred as usual, and its objects are lazily read from the local sidecar. If TRUE, the archive is never downloaded: the <code>simList</code> shell, its manifest and the output files are fetched from within the remote archive by HTTP range request, and each object is fetched only when something touches it. Output files are fetched either way, since downstream code reads them through <code>outputs(sim)\$file</code> and they are mostly needed. Requires an archive with an index beside it (see <code>reIndex()</code>) and <code>method = "loadSimList"</code> .
<code>overwrite</code>	Logical. Force re-download even if the local file exists. Default FALSE.
<code>verbose</code>	Logical. Print elapsed time per download. Default TRUE.
<code>skipExisting</code>	Logical, passed to <code>reUntar()</code> . If TRUE (default), archives already extracted on disk are not extracted again.
<code>useCache</code>	Logical. If TRUE, <code>reproducible::Cache()</code> the <i>metadata</i> step (<code>reGet()</code>), skipping the remote round-trip on repeat calls in a new session. Only the small <code>data.table</code> of names and paths is cached – never the <code>simLists</code> , whose objects are <code>delayedAssign()</code> promises that saving would force, and so fully materialise. A hit is discarded if any path it names has since gone away. Default FALSE, since a hit cannot notice an archive re-uploaded under the same id.
<code>pattern</code>	Optional regex. When supplied, <code>gFiles</code> is taken to be the <i>folder</i> holding the archives and is listed with <code>outList()</code> using this pattern, rather than being the archives themselves. Either way, the <code>_index.rds</code> sidecars written by <code>reIndex()</code> are excluded.

Value

A named list of `simList` objects, one per row of `gFiles`, named by the archive's name (sans `.tar.gz`).

See Also

[reGet\(\)](#), [reUntar\(\)](#), [reLoad\(\)](#), [outSaveTarUpload\(\)](#)

register_scenario_format

Register a project-specific path builder / parser.

Description

Pass a function (or, for withFieldLabel, a character vector) to register it; pass NULL explicitly to clear that slot; omit the argument to leave it untouched. Call with no arguments to inspect.

Usage

```
register_scenario_format(build, parse, withFieldLabel)
```

Arguments

build Function (custom path builder), or NULL to clear.

parse Function (custom path parser), or NULL to clear.

withFieldLabel Either:

- a *character vector* of field names (e.g. `c(".rep", ".SSP")`) – those fields' path segments get prefixed with the field name itself (`.rep5`, `.SSP370`); or
- a *named character vector* mapping field name to display label (e.g. `c(.rep = "rep", .SSP = "_ssp")`) – those fields' path segments get prefixed with the *mapped* label (`rep5`, `_ssp370`). Pass `character(0)` or NULL to clear.

Details

Lookup precedence (highest first): registered slot -> pathBuild/pathParse defined in the global environment -> the package defaults.

Override signature contract: `build(..., pre = "outputs")` — receives the scenario as named ... args (one per field) plus pre; returns a path string. `parse(path, pre = "outputs")` — returns a named list of fields.

Value

Invisibly, the current (build, parse, withFieldLabel) triple.

reIndex

Read one member out of a remote tar archive, without downloading it

Description

`outTar()` writes an *uncompressed* tar (R's `utils::tar()` defaults to `compression = "none"`), so every member sits at a known byte offset. Google Drive answers HTTP Range requests with 206 Partial Content, so a single object can be pulled out of a multi-GB archive by fetching only its bytes.

For a lazily saved `simList` this is the difference between downloading the whole tarball and reading the one object you want: measured on a 4.63 GB archive, one 6.2 MB object took **0.83 s** against ~304 s for the whole file.

`reIndex()` fetches the `<runName>_index.rds` written beside the tarball; `reGetMember()` uses it to fetch a single member.

Usage

```
reIndex(gFile, destDir = tempdir())
```

```
reGetMember(gFile, member, index = NULL, file = NULL, progress = NULL)
```

Arguments

<code>gFile</code>	A Google Drive dribble row, file id, or URL for the tarball .
<code>destDir</code>	Directory for the downloaded index. Default <code>tempdir()</code> .
<code>member</code>	Character. The archive member to fetch. Matched exactly, or as a regular expression when no exact match exists (it must then match one row).
<code>index</code>	Optional <code>data.frame</code> from <code>reIndex()</code> . Fetched if not supplied; pass it when reading several members so the index is downloaded once.
<code>file</code>	Optional path to write the raw member to. When <code>NULL</code> (default) the member is read back with <code>base::readRDS()</code> and returned.
<code>progress</code>	Logical or <code>NULL</code> . Show a transfer progress bar. <code>NULL</code> (default) shows one in an interactive session for members over ~5 MB, where the wait is long enough to look like a hang.

Value

`reIndex()`: a `data.frame` with name, offset and size, one row per archive member.

`reGetMember()`: the deserialized object, or the path written when `file` is supplied.

See Also

`reGetMember()`, `outTar()`, `reGetUntarLoad()`

reLoad

*Load saved SpaDES simLists***Description**

Inverse of `outSave()`. Loads one or more `simLists` from `.rds` files produced by `outSave()`. Defaults to `SpaDES.core::loadSimList()`; set `method = "readRDS"` to bypass `.unwrap` entirely.

Note that `SpaDES.core::saveSimList()` uses `.wrapResiliently` to NULL out file-backed objects with inaccessible backing files at save time. Load-time failures (e.g. backing files missing on this machine even though they were present at save time) are independent of that, and are handled by `loadSimList`'s pre-`.unwrap` resilient pass.

Usage

```
reLoad(
  simFileNames,
  projectPath = getwd(),
  method = c("loadSimList", "readRDS"),
  parse = TRUE,
  ...
)
```

Arguments

<code>simFileNames</code>	Character vector of paths to <code>.rds</code> files.
<code>projectPath</code>	Character scalar. Passed to <code>SpaDES.core::loadSimList()</code> for relative-path resolution. Default <code>getwd()</code> .
<code>method</code>	One of <code>"loadSimList"</code> (default) or <code>"readRDS"</code> .
<code>parse</code>	Logical. If <code>TRUE</code> (default), module source code is re-parsed on load. <code>FALSE</code> forwards <code>parse = FALSE</code> to <code>SpaDES.core::loadSimList()</code> , which skips it – worthwhile because reparsing dominates the load time of a lazily saved <code>simList</code> (on a 19-module simulation, ~7 s of a ~9.5 s load). Objects are unaffected: user objects and each module's <code>mod</code> objects are still bound lazily. The result is inspect-only – it has no module code, so it cannot be passed to <code>SpaDES.core::spades()</code> . Ignored when <code>method = "readRDS"</code> .
<code>...</code>	Additional args forwarded to <code>SpaDES.core::loadSimList()</code> (ignored when <code>method = "readRDS"</code>).

Value

A list of `simList` objects, named by `basename(simFileNames)`.

See Also

`reGet()`, `reUntar()`, `reGetUntarLoad()`, `outSave()`

reUntar

*Extract sim tarballs, optionally remapping a path prefix***Description**

Inverse of [outTar\(\)](#). Extracts one or more `.tar.gz` archives produced by [outTar\(\)](#) / [outSaveTarUpload\(\)](#), which contain absolute paths. If `pathRemap` is supplied, the leading path prefix is rewritten on extraction (handy when the archive was created on another user's machine, e.g. paths starting with `/home/emcintir/...`).

Path rewriting uses GNU tar's `--transform`. On systems without GNU tar, supply `pathRemap = NULL` and the archive's absolute paths are restored as-is.

Usage

```
reUntar(tarballs, pathRemap = NULL, verbose = FALSE, skipExisting = TRUE)
```

Arguments

<code>tarballs</code>	Character vector of paths to local tarballs.
<code>pathRemap</code>	Optional named character vector of length 2, <code>c(old = "/old/prefix", new = "/new/prefix")</code> , applied to all tarballs. If <code>NULL</code> (default), files are extracted to their original absolute paths (<code>tar --absolute-names</code>).
<code>verbose</code>	Logical. Pass <code>-v</code> to tar. Default <code>FALSE</code> .
<code>skipExisting</code>	Logical. If <code>TRUE</code> (default), an archive whose members are all already present on disk (after any <code>pathRemap</code>) is not extracted again. Set <code>FALSE</code> to force re-extraction, e.g. if the extracted files may have been modified in place.

Value

A character vector (same length as `tarballs`) of absolute paths to the `.rds simList` file inside each archive (after any remap), suitable for [reLoad\(\)](#).

See Also

[reGet\(\)](#), [reLoad\(\)](#), [reGetUntarLoad\(\)](#), [outTar\(\)](#)

runWorkerLoopFuture	<i>Worker loop for future/cluster-based remote execution</i>
---------------------	--

Description

A thin wrapper around `tmuxRunWorkerLoop` that optionally redirects console output to a log file before entering the job loop. Used internally by `experimentFuture` for remote (cluster) workers. Local workers use `callr::r_bg()` and do not need this wrapper.

Usage

```
runWorkerLoopFuture(
  queue_path,
  global_path,
  on_interrupt = c("requeue", "fail"),
  on_error = c("stop", "continue"),
  ss_id = NULL,
  email = NULL,
  cache_path = NULL,
  runNameLabel = quote(colnames(q)[1:2]),
  activeRunningPath = NULL,
  dots_path = NULL,
  stop_file = NULL,
  log_file = NULL,
  snapshot_library = TRUE,
  n_workers_on_host = 1L
)
```

Arguments

<code>queue_path</code>	Path to the local RDS queue file.
<code>global_path</code>	Path to the R script sourced for each job.
<code>on_interrupt</code>	"requeue" or "fail".
<code>on_error</code>	"stop" (default) leaves a worker stopped after a job errors, with the error visible in its log; "continue" goes on to the next claimable job. The errored row is marked ERROR either way (see <code>on_interrupt</code>).
<code>ss_id</code>	Google Sheets ID for the shared queue (or NULL for file-based queue).
<code>email</code>	Gargle OAuth e-mail.
<code>cache_path</code>	Gargle OAuth cache directory.
<code>runNameLabel</code>	Quoted expression for deriving a run name.
<code>activeRunningPath</code>	Directory for <code>Running_*.rds</code> marker files.
<code>dots_path</code>	Path to an RDS file whose contents are loaded into <code>.GlobalEnv</code> before sourcing <code>global_path</code> .

<code>stop_file</code>	Path to a sentinel file. When this file is created (e.g. by killExperimentFuture), the worker exits cleanly after its current job finishes.
<code>log_file</code>	Path to the log file for this worker. If NULL, output goes to the current connection.
<code>snapshot_library</code>	Logical (default TRUE). Each worker session takes a hardlinked snapshot of the project library before loading anything from it, so an install into the shared library cannot corrupt a running job. See librarySnapshot .
<code>n_workers_on_host</code>	Number of workers the launcher runs on this host; passed to tmuxRunWorkerLoop() .

Value

Invisibly returns the worker identifier string.

<code>scenario</code>	<i>Construct a scenario record.</i>
-----------------------	-------------------------------------

Description

Accepts any named arguments. Each name becomes a field label; each value the field's value. No specific field set is required by the package (fields are project-defined via the queue).

Usage

```
scenario(...)
```

Arguments

`...` Named field/value pairs.

Details

Light coercion: a single character of the form "a:b" is eval'd as an R expression (so queue cells like "1991:2020" become integer vectors).

Value

An S3 object of class "scenario".

Description

A "scenario" identifies a single simulation run. Field *names* and *values* are discovered from the driver queue (Google Sheet); they are not hardcoded in this package. The same run can be referred to in three interchangeable ways:

Details

1. Field values (one column per field in the queue), e.g. (.ELFind = "6.3.1", .samplingRange = 2071:2100, ...).
2. An output directory path under outputs/.
3. An upload tar filename (path with / -> _ and .tar.gz suffix).

This file defines:

- a canonical record (S3 class "scenario");
- the generic `as_scenario()` for coercing any representation into it;
- formatters `as_path()` / `as_tarname()` for going back;
- default builders `pathBuild()` / `pathParse()`: each non-empty field's *value* (no label) is one path segment, joined by /, in the order given by `scenarioFields()`. Integer-and-contiguous vectors render as start-end. Empty / NA fields are skipped entirely (one fewer segment); see `pathParse()` for the trailing-NA round-trip caveat.

Per-project format overrides: define your own `pathBuild` (and matching `pathParse`) in the global environment, or register them explicitly with `register_scenario_format()`. Lookup order, highest first: `register_scenario_format` slot -> a `pathBuild/pathParse` in the global environment -> the package default.

Field discovery: `queueRead()` caches the queue's non-meta column names as the active field set. Subsequent `pathParse()` calls use those labels for positional decoding. If you parse paths without first reading a queue, pass `fields = c(...)` explicitly (or call `scenarioFieldsSet()`).

Examples

```
## Not run:
## --- Default (generic) format -----
queue <- queueRead(folder = ss_id, name = "longRuns")
# -> data.table with columns .ELFind, .samplingRange, .GCM, .SSP, .rep
#   plus meta columns (status, started_at, ...). Non-meta columns are
#   auto-cached as scenarioFields().

scens <- as_scenario(queue)           # list of `scenario` objects
as_path(scens[[1]])
#> "outputs/6.3.1/2071-2100/CNRM-ESM2-1/370/5"
as_tarname(scens[[1]])
#> "6.3.1_2071-2100_CNRM-ESM2-1_370_5.tar.gz"
```

```

# Round-trip
s2 <- as_scenario("outputs/6.3.1/2071-2100/CNRM-ESM2-1/370/5")
identical(unclass(scens[[1]]), unclass(s2)) # TRUE

# Cross-reference queue against uploaded tarballs
uploads <- outScenarios(.uploadGSDir) # list of scenarios
missing <- queueUploadMissing(folder = ss_id, name = "longRuns",
                              uploadFolder = .uploadGSDir) # queue rows only

## --- Per-field labels in the path -----
# `withFieldLabel` accepts two forms.

# 1) Unnamed character vector: prefix with the field name itself.
as_path(scens[[1]], withFieldLabel = c(".rep", ".SSP"))
#> "outputs/6.3.1/2071-2100/CNRM-ESM2-1/.SSP370/.rep5"

# 2) Named character vector: prefix with the *mapped* label
# (e.g., emit `.rep` as `rep`, `.SSP` as `_ssp`).
as_path(scens[[1]], withFieldLabel = c(.rep = "rep", .SSP = "_ssp"))
#> "outputs/6.3.1/2071-2100/CNRM-ESM2-1/_ssp370/rep5"

# Set once for every subsequent as_path() / as_tarname():
register_scenario_format(withFieldLabel = c(.rep = "rep", .SSP = "_ssp"))
as_path(scens[[1]])
#> "outputs/6.3.1/2071-2100/CNRM-ESM2-1/_ssp370/rep5"
as_tarname(scens[[1]])
#> "6.3.1_2071-2100_CNRM-ESM2-1__ssp370_rep5.tar.gz"
# Round-trip parses back to canonical fields:
as_scenario("outputs/6.3.1/2071-2100/CNRM-ESM2-1/_ssp370/rep5")

## --- Project-specific format (FireSenseTesting layout) -----
# Layout: outputs/<.ELFind>/<range>/<GCM>_ssp<SSP>/rep<.rep>
# E.g.    outputs/6.3.1/2071-2100/CNRM-ESM2-1_ssp370/rep5

myBuild <- function(.ELFind, .samplingRange, .GCM, .SSP, .rep,
                    pre = "outputs") {
  sr <- if (is.numeric(.samplingRange)) .samplingRange
  else
    eval(parse(text = .samplingRange))
  file.path(pre, .ELFind,
            paste(range(sr), collapse = "-"),
            paste0(.GCM, ifelse(is.na(.SSP), "", paste0("_ssp", .SSP))),
            paste0("rep", .rep))
}

myParse <- function(path, fields = scenarioFields(), pre = "outputs") {
  clean <- sub("\\.tar\\.gz$", "", path)
  clean <- sub(paste0("^", pre, "[/_]"), "", clean)
  parts <- if (grepl("/", clean)) strsplit(clean, "/")[[1L]]
  else
    strsplit(clean, "_")[[1L]]
  repIdx <- which(grepl("^rep[0-9]+$", parts))
  rangeIdx <- which(grepl("[0-9]+--[0-9]+$", parts))
  gcmSsp <- paste(parts[(rangeIdx + 1L):(repIdx - 1L)], collapse = "_")

```

```

gs      <- if (grepl("_ssp", gcmSsp)) strsplit(gcmSsp, "_ssp")[[1L]]
      else                                     c(gcmSsp, NA_character_)
rng      <- as.integer(strsplit(parts[rangeIdx], "-")[[1L]])
list(.ELFind      = paste(parts[seq_len(rangeIdx - 1L)], collapse = "_"),
     .samplingRange = rng[1L]:rng[2L],
     .GCM          = gs[1L],
     .SSP          = gs[2L],
     .rep          = as.integer(sub("^rep", "", parts[repIdx])))
}

register_scenario_format(build = myBuild, parse = myParse)
as_path(scens[[1]])
#> "outputs/6.3.1/2071-2100/CNRM-ESM2-1_ssp370/rep5"
as_tarname(scens[[1]])
#> "6.3.1_2071-2100_CNRM-ESM2-1_ssp370_rep5.tar.gz"

# Equivalent: define pathBuild / pathParse in your global environment
# (e.g. in a project global.R) -- they will be auto-detected.
pathBuild <- myBuild
pathParse <- myParse

## End(Not run)

```

scenarioFields	<i>Active scenario field labels.</i>
----------------	--------------------------------------

Description

Returns the field labels currently used to (a) parse paths/tarnames back into scenario records and (b) determine which queue columns constitute the scenario (vs. queue meta-columns). Set automatically by [queueRead\(\)](#); can be set manually with [scenarioFieldsSet\(\)](#).

Usage

```

scenarioFields()

scenarioFieldsSet(fields)

```

Arguments

fields Character vector of field labels.

Value

Character vector of field names, or NULL if not yet known.

setProjPkgDir	<i>Set the package directory for a project</i>
---------------	--

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

<code>lib.loc</code>	The folder for installing packages inside of
<code>verbose</code>	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 <code>Require</code> function, when <code>verbose >= 2</code> , also returns details as if <code>returnDetails = TRUE</code> (for backwards compatibility).

setupFiles	<i>Parse a list of (possibly remote) R / config files</i>
------------	---

Description

Convenience helper, intended primarily for interactive use, that parses each file (local path or `github.com` URL with `@branch` notation) into a named list.

Usage

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

Arguments

<code>files</code>	A vector or list of files to parse. These can be remote <code>github.com</code> files.
<code>paths</code>	a list with named elements, specifically, <code>modulePath</code> , <code>projectPath</code> , <code>packagePath</code> and all others that are in <code>SpaDES.core::setPaths()</code> (i.e., <code>inputPath</code> , <code>outputPath</code> , <code>scratchPath</code> , <code>cachePath</code> , <code>rasterTmpDir</code>). Each of these has a sensible default, which will be overridden but any user supplied values. See setup .

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
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).

Details

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

setupFiles a named list with each element that was parsed.

See Also

[setupProject\(\)](#) for the high-level wrapper, [setup_family](#) for an overview.

setupFunctions	<i>Source user-supplied helper functions into the project environment</i>
----------------	---

Description

Source the functions supplied to [setupProject\(\)](#) so they are available to subsequent setup* steps and to the user's session.

Usage

```
setupFunctions(
  functions,
  name,
  sideEffects,
  paths,
  overwrite = FALSE,
  envir = parent.frame(),
  callingEnv = sys.frame(-2),
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  ...
)
```

Arguments

functions	A set of function definitions to be used within <code>setupProject</code> . These will be returned as a list element. If function definitions require non-base packages, prefix the function call with the package e.g., <code>terra::rast</code> . When using <code>setupProject</code> , the <code>functions</code> argument is evaluated after <code>paths</code> , so it cannot be used to define functions that help specify paths.
name	Optional. If supplied, the name of the project. If not supplied, an attempt will be made to extract the name from the <code>paths[["projectPath"]]</code> . If this is a GitHub project, then it should indicate the full Github repository and branch name, e.g., <code>"PredictiveEcology/WBI_forecasts@ChubatyPubNum12"</code>
sideEffects	Optional. This can be an expression or one or more file names or a code chunk surrounded by <code>{...}</code> . If a non-text file name is specified (e.g., <i>not</i> <code>.txt</code> or <code>.R</code> 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 (<code>*.txt</code> or <code>.R</code>), 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.
paths	a list with named elements, specifically, <code>modulePath</code> , <code>projectPath</code> , <code>packagePath</code> and all others that are in <code>SpaDES.core::setPaths()</code> (i.e., <code>inputPath</code> , <code>outputPath</code> , <code>scratchPath</code> , <code>cachePath</code> , <code>rasterTmpDir</code>). Each of these has a sensible default, which will be overridden but any user supplied values. See setup .
overwrite	Logical vector or character vector, however, only <code>getModule</code> 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 <code>setupProject</code> – including those downloaded via <code>sideEffects</code> . If a length > 1 logical or character vector, these will be passed to <code>getModule</code> : only the named modules will be overwritten or the logical vector of the modules. NOTE: if length > 1, no other file specified anywhere in <code>setupProject</code> will be overwritten except a module matching the vector <code>names()</code> (because only <code>setupModules</code> is currently responsive to a vector). To have fine grained control, a user can just manually delete a file, then rerun. Default is <code>getOption("SpaDES.project.overwrite", FALSE)</code> , i.e., FALSE unless that option is set.
envir	The environment where <code>setupProject</code> is called from. Defaults to <code>parent.frame()</code> which should be fine in most cases and user shouldn't need to set this
callingEnv	The environment from which the function was called. Defaults to <code>sys.frame(-2)</code> which represents the case where the inner <code>setup*</code> functions are called inside <code>setupProject</code> , which was called by a user.
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 <code>Require</code> function, when <code>verbose >= 2</code> , also returns details as if <code>returnDetails = TRUE</code> (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, written as a literal `list(...)` or as a variable holding one. 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. In a literal `list(...)` an entry may reference an entry written above it. 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.

Details

`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.

Value

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

See Also

[setupProject\(\)](#) for the high-level wrapper, [setup_family](#) for an overview.

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)))

```

setupGitIgnore	<i>Add packagePath and/or modulePath to the project's .gitignore</i>
----------------	--

Description

Helper that keeps the .gitignore of a project under git control in sync with the project's resolved paths.

Usage

```

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

```

Arguments

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 .
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 = ...).
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).

Details

setupGitIgnore will add the relevant paths to .gitignore.

Value

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.

See Also

[setupProject\(\)](#) for the high-level wrapper, [setup_family](#) for an overview.

setupModules	<i>Download (or git clone) SpaDES modules into the project's modulePath</i>
--------------	---

Description

Materialise the modules requested in [setupProject\(\)](#) beneath `paths[["modulePath"]]`, optionally as git submodules.

Usage

```
setupModules(
  name,
  paths,
  modules,
  inProject,
  useGit = getOption("SpaDES.project.useGit", FALSE),
  overwrite = FALSE,
  envir = parent.frame(),
  callingEnv = sys.frame(-2),
  gitUserName,
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  updateRprofile = getOption("SpaDES.project.updateRprofile", TRUE),
  ...
)
```

Arguments

name	Optional. If supplied, the name of the project. If not supplied, an attempt will be made to extract the name from the <code>paths[["projectPath"]]</code> . 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 <code>SpaDES.core::setPaths()</code> (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 <code>getModule</code>. These should be one of: simple name (e.g., <code>fireSense</code>) which will be searched for locally in the <code>paths[["modulePath"]]</code>; or a GitHub repo with branch (<code>GitHubAccount/Repo@branch</code> e.g., <code>"PredictiveEcology/Biomass_core@development"</code>); or a character vector that identifies one or more module folders (local or GitHub) (not the module <code>.R</code> 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. For convenience, these can also be 2 other url formats: 1. the <code>raw.githubusercontent.com</code> url that points to the main module file or the folder e.g., <code>"https://raw.githubusercontent.com/PredictiveEcology/Biomass_core/refs/heads/main/modulePath.R"</code> 2. The <code>github.com</code> url used for cloning a git repository, with optional <code>"@branch"</code> specified: <code>"https://github.com/PredictiveEcology/Biomass_speciesParameters.git@development"</code>. See setup.
inProject	A logical. If TRUE, then the current directory is inside the <code>paths[["projectPath"]]</code> .
useGit	(if not FALSE, then experimental still). There are two levels at which a project can use GitHub, either the <code>projectPath</code> and/or the <code>modules</code>. 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 <code>modules</code>. A logical or "sub" for <i>submodule</i>. If "sub", then this function will attempt to clone the identified modules <i>as git submodules</i>. This will only work if the <code>projectPath</code> 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 <code>modules</code> as git submodules. If it is not already, it will use <code>git clone</code> for each module. After <code>git clone</code> or <code>git submodule add</code> are run, it will run <code>git checkout</code> for the named branch and then <code>git pull</code> to get and change branch for each module, according to its specification in <code>modules</code>. If FALSE, this function will download modules with <code>getModules</code>. NOTE: <i>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.</i> This can be set with the option (<code>SpaDES.project.useGit = xxx</code>). Default is <code>getOption("SpaDES.project.useGit", FALSE)</code>, i.e., FALSE unless that option is set.
overwrite	Logical vector or character vector, however, only <code>getModule</code> 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 <code>setupProject</code> – including those downloaded via <code>sideEffects</code>. If a length > 1 logical or character vector, these will be passed to <code>getModule</code>: only the named modules will be overwritten or the logical vector of the modules. NOTE: if length > 1, no other file specified anywhere in <code>setupProject</code> will be overwritten except a module matching the vector <code>names()</code> (because only <code>setupModules</code> is currently responsive to a vector). To have fine grained control, a user can just manually delete a

	file, then rerun. Default is <code>getOption("SpaDES.project.overwrite", FALSE)</code> , i.e., FALSE unless that option is set.
<code>envir</code>	The environment where <code>setupProject</code> is called from. Defaults to <code>parent.frame()</code> which should be fine in most cases and user shouldn't need to set this
<code>callingEnv</code>	The environment from which the function was called. Defaults to <code>sys.frame(-2)</code> which represents the case where the inner <code>setup*</code> functions are called inside <code>setupProject</code> , which was called by a user.
<code>gitUserName</code>	The GitHub account name. Used with <code>git clone git@github.com:gitHuserName/name</code>
<code>verbose</code>	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 <code>Require</code> function, when <code>verbose >= 2</code> , also returns details as if <code>returnDetails = TRUE</code> (for backwards compatibility).
<code>dots</code>	Any other named objects passed as a list a user might want for other elements.
<code>defaultDots</code>	A named list of any arbitrary R objects, written as a literal <code>list(...)</code> or as a variable holding one. These can be supplied to give default values to objects that are otherwise passed in with the <code>...</code> , i.e., not specifically named for these <code>setup*</code> functions. If named objects are supplied as top-level arguments, then the <code>defaultDots</code> will be overridden. This can be particularly useful if the arguments passed to <code>...</code> do not always exist, but rely on external e.g., batch processing to optionally fill them. In a literal <code>list(...)</code> an entry may reference an entry written above it. See examples.
<code>updateRprofile</code>	Logical. Should the <code>paths\$packagePath</code> be set in the <code>.Rprofile</code> file for this project. Note: if <code>paths\$packagePath</code> is within the <code>tempdir()</code> , then there will be a warning, indicating this won't persist. If the user is using Rstudio and the <code>paths\$projectPath</code> is not the root of the current Rstudio project, then a warning will be given, indicating the <code>.Rprofile</code> may not be read upon restart. Default is <code>getOption("SpaDES.project.updateRprofile", TRUE)</code> , i.e., TRUE unless that option is set.
<code>...</code>	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 <code>SpaDES.core::simInit</code> , or <code>SpaDES.core::simInitAndSpades</code> , (e.g. <code>studyAreaName</code> or objects) or additional objects to be passed to the simulation (in older versions of <code>SpaDES.core</code> , these were passed as a named list to the <code>objects</code> argument). Order matters. These are sequentially evaluated, and also any arguments that are specified before the named arguments e.g., <code>name</code> , <code>paths</code> , will be evaluated prior to any of the named arguments, i.e., "at the start" of the <code>setupProject</code> . If placed after the first named argument, then they will be evaluated at the end of the <code>setupProject</code> , so can access all the packages, objects, etc.

Details

`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.

Value

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 elements are the R packages that the module depends on (reqsPkgs)

See Also

[setupProject\(\)](#) for the high-level wrapper, [setup_family](#) for an overview.

setupOptions	<i>Apply (and stage) project options</i>
--------------	--

Description

Set the options() supplied to [setupProject\(\)](#) and record the prior values so they can be restored.

Usage

```
setupOptions(
  name,
  options,
  paths,
  times,
  overwrite = FALSE,
  envir = parent.frame(),
  callingEnv = sys.frame(-2),
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  useGit = getOption("SpaDES.project.useGit", FALSE),
  updateRprofile = getOption("SpaDES.project.updateRprofile", TRUE),
  ...
)
```

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"
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 .

paths	a list with named elements, specifically, modulePath, projectPath, packagePath and all others that are in <code>SpaDES.core::setPaths()</code> (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 .
times	Optional. This will be returned if supplied; if supplied, the values can be used in e.g., <code>params</code> , e.g., <code>params = list(mod = list(startTime = times\$start))</code> . Default (when not supplied) is <code>list(start = 0, end = 1)</code> . See help for <code>SpaDES.core::simInit</code> .
overwrite	Logical vector or character vector, however, only <code>getModule</code> 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 <code>setupProject</code> – including those downloaded via <code>sideEffects</code> . If a length > 1 logical or character vector, these will be passed to <code>getModule</code> : only the named modules will be overwritten or the logical vector of the modules. NOTE: if length > 1, no other file specified anywhere in <code>setupProject</code> will be overwritten except a module matching the vector <code>names()</code> (because only <code>setupModules</code> is currently responsive to a vector). To have fine grained control, a user can just manually delete a file, then rerun. Default is <code>getOption("SpaDES.project.overwrite", FALSE)</code> , i.e., FALSE unless that option is set.
envir	The environment where <code>setupProject</code> is called from. Defaults to <code>parent.frame()</code> which should be fine in most cases and user shouldn't need to set this
callingEnv	The environment from which the function was called. Defaults to <code>sys.frame(-2)</code> which represents the case where the inner <code>setup*</code> functions are called inside <code>setupProject</code> , which was called by a user.
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 <code>Require</code> function, when <code>verbose >= 2</code> , also returns details as if <code>returnDetails = TRUE</code> (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, written as a literal <code>list(...)</code> or as a variable holding one. These can be supplied to give default values to objects that are otherwise passed in with the <code>...</code> , i.e., not specifically named for these <code>setup*</code> functions. If named objects are supplied as top-level arguments, then the <code>defaultDots</code> will be overridden. This can be particularly useful if the arguments passed to <code>...</code> do not always exist, but rely on external e.g., batch processing to optionally fill them. In a literal <code>list(...)</code> an entry may reference an entry written above it. See examples.
useGit	(if not FALSE, then experimental still). There are two levels at which a project can use GitHub, either the <code>projectPath</code> 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 <i>submodule</i> . If "sub", then this function will attempt to clone the identified modules <i>as git submodules</i> . This will only work if the <code>projectPath</code> 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). Default is getOption("SpaDES.project.useGit", FALSE), i.e., FALSE unless that option is set.

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. Default is getOption("SpaDES.project.updateRprofile", TRUE), i.e., TRUE unless that option is set.

... 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.

Details

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.

Value

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.

See Also

[setupProject\(\)](#) for the high-level wrapper, [setup_family](#) for an overview.

 setupPackages

Install module + user-supplied R packages into the project library

Description

Combine the modules' reqdPkgs with the user-supplied packages and install all of them into paths[["packagePath"]] via `Require::Install()`.

Usage

```
setupPackages(
  packages,
  modulePackages = list(),
  require = list(),
  paths,
  libPaths,
  setLinuxBinaryRepo = TRUE,
  standAlone,
  envir = parent.frame(),
  callingEnv = sys.frame(-2),
  verbose = getOption("Require.verbose"),
  dots,
  defaultDots,
  ...
)
```

Arguments

- | | |
|----------------|--|
| packages | Optional. A vector of packages that must exist in the libPaths. This will be passed to <code>Require::Install</code> , i.e., these will be installed, but not attached to the search path. See also the <code>require</code> argument. To force skip of package installation (without assessing modules), set <code>packages = NULL</code> .

To install everything from repos (e.g., prebuilt binaries on <code>predictiveecology.r-universe.dev</code>) rather than from GitHub source – thereby avoiding git authentication and a source-build toolchain such as <code>Rtools</code> – set <code>options = list(Require.noRemotes = TRUE)</code> . Any GitHub-style specs (<code>account/repo@branch</code>), including those declared in module metadata (<code>reqdPkgs</code>), are then rewritten to their bare package name (version constraints preserved) and resolved from repos. Ensure the relevant repository is in <code>getOption("repos")</code> and carries a version satisfying any constraint. See <code>Require::RequireOptions</code> . |
| modulePackages | A named list, where names are the module names, and the elements of the list are packages in a form that <code>Require::Require</code> accepts. |
| require | Optional. A character vector of packages to install <i>and</i> attach (with <code>Require::Require</code>). These will be installed and attached at the start of <code>setupProject</code> so that a user can use these during <code>setupProject</code> . Default is <code>NULL</code> , i.e., nothing additional installed/attached. See setup |

paths	a list with named elements, specifically, modulePath, projectPath, packagePath and all others that are in <code>SpaDES.core::setPaths()</code> (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 .
libPaths	Deprecated. Use <code>paths = list(packagePath = ...)</code> .
setLinuxBinaryRepo	Logical. Should the binary RStudio Package Manager be used on Linux (ignored if Windows). Default is <code>getOption("SpaDES.project.setLinuxBinaryRepo", TRUE)</code> , i.e., TRUE unless that option is set.
standAlone	A logical. Passed to <code>Require::standAlone</code> . This keeps all packages installed in a project-level library, if TRUE. Default is <code>getOption("SpaDES.project.standAlone", TRUE)</code> , i.e., TRUE unless that option is set.
envir	The environment where <code>setupProject</code> is called from. Defaults to <code>parent.frame()</code> which should be fine in most cases and user shouldn't need to set this
callingEnv	The environment from which the function was called. Defaults to <code>sys.frame(-2)</code> which represents the case where the inner <code>setup*</code> functions are called inside <code>setupProject</code> , which was called by a user.
verbose	Numeric or logical indicating how verbose the function should be. At <code>verbose >= 2</code> , the combined <code>reqdPkgs</code> are printed grouped by module. At <code>verbose >= 3</code> , additionally the <code>dput()</code> of the exact package vector passed to <code>Require::Require</code> is printed, which can be copy-pasted to reproduce the install call. If not supplied, defaults to <code>getOption("Require.verbose")</code> .
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, written as a literal <code>list(...)</code> or as a variable holding one. These can be supplied to give default values to objects that are otherwise passed in with the <code>...</code> , i.e., not specifically named for these <code>setup*</code> functions. If named objects are supplied as top-level arguments, then the <code>defaultDots</code> will be overridden. This can be particularly useful if the arguments passed to <code>...</code> do not always exist, but rely on external e.g., batch processing to optionally fill them. In a literal <code>list(...)</code> an entry may reference an entry written above it. 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 <code>SpaDES.core::simInit</code> , or <code>SpaDES.core::simInitAndSpades</code> , (e.g. <code>studyAreaName</code> or objects) or additional objects to be passed to the simulation (in older versions of <code>SpaDES.core</code> , these were passed as a named list to the <code>objects</code> argument). Order matters. These are sequentially evaluated, and also any arguments that are specified before the named arguments e.g., <code>name</code> , <code>paths</code> , will be evaluated prior to any of the named arguments, i.e., "at the start" of the <code>setupProject</code> . If placed after the first named argument, then they will be evaluated at the end of the <code>setupProject</code> , so can access all the packages, objects, etc.

Details

`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(...)`.

Value

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

Skipping Require

When a call asks for the same package specs as an earlier successful call, and those packages and all their Depends/Imports/LinkingTo dependencies are installed at the same versions (and, for GitHub packages, the same RemoteSha) in the same libPaths, Require::Require is skipped; (HEAD) packages are still passed to it. This needs reproducible and qs2. Turn it off with options(SpaDES.project.skipIdenticalRequire = FALSE); the default is TRUE.

See Also

[setupProject\(\)](#) for the high-level wrapper, [setup_family](#) for an overview.

 setupParams

Prepare module parameter lists for simInit()

Description

Build the nested params list that `SpaDES.core::simInit()` consumes from the user-supplied params argument to `setupProject()`.

Usage

```
setupParams(
  name,
  params,
  paths,
  modules,
  times,
  options,
  overwrite = FALSE,
  envir = parent.frame(),
  callingEnv = sys.frame(-2),
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  repDeclared = FALSE,
  ...
)
```

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"
------	--

params	Optional. Similar to options, however, this named list will be returned, i.e., there are no side effects. If the project declares a <code>.rep</code> dot (the replicate column of an experiment table), as an argument or in <code>defaultDots</code> , <code>params\$globals\$.rep</code> is set to its value (as integer), so modules with a <code>.rep</code> parameter receive it. The value is the one a dot resolves to: an object <code>.rep</code> the caller can see (e.g. the queue's column in a worker) wins over the <code>defaultDots</code> entry. A <code>.rep</code> the project does not declare, e.g. one left over in an interactive session, is ignored. A <code>.rep</code> set in <code>params\$globals</code> is kept, and a module's own <code>params\$<module>\$.rep</code> overrides <code>.globals</code> , as for any global parameter. No other dot is passed on in this way. See setup .
paths	a list with named elements, specifically, <code>modulePath</code> , <code>projectPath</code> , <code>packagePath</code> and all others that are in <code>SpaDES.core::setPaths()</code> (i.e., <code>inputPath</code> , <code>outputPath</code> , <code>scratchPath</code> , <code>cachePath</code> , <code>rasterTmpDir</code>). 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 <code>getModule</code> . These should be one of: simple name (e.g., <code>fireSense</code>) which will be searched for locally in the <code>paths[["modulePath"]]</code> ; or a GitHub repo with branch (<code>GitHubAccount/Repo@branch</code> e.g., <code>"PredictiveEcology/Biomass_core@development"</code>); or a character vector that identifies one or more module folders (local or GitHub) (not the module <code>.R</code> 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. For convenience, these can also be 2 other url formats: 1. the <code>raw.githubusercontent.com</code> url that points to the main module file or the folder e.g., <code>"https://raw.githubusercontent.com/PredictiveEcology/Biomass_core/refs/heads/main/module.R"</code> 2. The <code>github.com</code> url used for cloning a git repository, with optional <code>"@branch"</code> specified: <code>"https://github.com/PredictiveEcology/Biomass_speciesParameters.git@development"</code> See setup .
times	Optional. This will be returned if supplied; if supplied, the values can be used in e.g., <code>params</code> , e.g., <code>params = list(mod = list(startTime = times\$start))</code> . Default (when not supplied) is <code>list(start = 0, end = 1)</code> . See help for <code>SpaDES.core::simInit</code> .
options	Optional. Either a named list to be passed to <code>options</code> or a character vector indicating one or more file(s) to source, in the order provided. These will be parsed locally (not the <code>.GlobalEnv</code>), so they will not create globally accessible objects. NOTE: <code>options</code> is run 2x within <code>setupProject</code> , once before <code>setupPaths</code> and once after <code>setupPackages</code> . This occurs because many packages use <code>options</code> for their behaviour (need them set before e.g., <code>Require::require</code> is run; but many packages also change <code>options</code> at startup. See details. See setup .
overwrite	Logical vector or character vector, however, only <code>getModule</code> will respond to a vector of values. If length-one <code>TRUE</code> , then all files that were previously downloaded will be overwritten throughout the sequence of <code>setupProject</code> – including those downloaded via <code>sideEffects</code> . If a length > 1 logical or character vector, these will be passed to <code>getModule</code> : only the named modules will be overwritten or the logical vector of the modules. NOTE: if length > 1, no other file specified anywhere in <code>setupProject</code> will be overwritten except a module matching the vector <code>names()</code> (because only <code>setupModules</code> is currently responsive to a vector). To have fine grained control, a user can just manually delete a

	file, then rerun. Default is <code>getOption("SpaDES.project.overwrite", FALSE)</code> , i.e., FALSE unless that option is set.
<code>envir</code>	The environment where <code>setupProject</code> is called from. Defaults to <code>parent.frame()</code> which should be fine in most cases and user shouldn't need to set this
<code>callingEnv</code>	The environment from which the function was called. Defaults to <code>sys.frame(-2)</code> which represents the case where the inner <code>setup*</code> functions are called inside <code>setupProject</code> , which was called by a user.
<code>verbose</code>	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 <code>Require</code> function, when <code>verbose >= 2</code> , also returns details as if <code>returnDetails = TRUE</code> (for backwards compatibility).
<code>dots</code>	Any other named objects passed as a list a user might want for other elements.
<code>defaultDots</code>	A named list of any arbitrary R objects, written as a literal <code>list(...)</code> or as a variable holding one. These can be supplied to give default values to objects that are otherwise passed in with the <code>...</code> , i.e., not specifically named for these <code>setup*</code> functions. If named objects are supplied as top-level arguments, then the <code>defaultDots</code> will be overridden. This can be particularly useful if the arguments passed to <code>...</code> do not always exist, but rely on external e.g., batch processing to optionally fill them. In a literal <code>list(...)</code> an entry may reference an entry written above it. See examples.
<code>repDeclared</code>	Logical. TRUE when the project declares a <code>.rep</code> dot (an argument or a <code>defaultDots</code> entry of <code>setupProject()</code> , which sets it); then a <code>.rep</code> the caller can see becomes <code>params\$.globals\$.rep</code> even when the default was not bound. FALSE (default, for direct calls) only uses a <code>.rep</code> in <code>callingEnv</code> .
<code>...</code>	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 <code>SpaDES.core::simInit</code> , or <code>SpaDES.core::simInitAndSpades</code> , (e.g. <code>studyAreaName</code> or objects) or additional objects to be passed to the simulation (in older versions of <code>SpaDES.core</code> , these were passed as a named list to the <code>objects</code> argument). Order matters. These are sequentially evaluated, and also any arguments that are specified before the named arguments e.g., <code>name</code> , <code>paths</code> , will be evaluated prior to any of the named arguments, i.e., "at the start" of the <code>setupProject</code> . If placed after the first named argument, then they will be evaluated at the end of the <code>setupProject</code> , so can access all the packages, objects, etc.

Value

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

See Also

[setupProject\(\)](#) for the high-level wrapper, [setup_family](#) for an overview.

 setupPaths

 Set up project, module, and scratch paths

Description

Resolve, default-fill, and apply the path list used by `setupProject()`.

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(),
  callingEnv = sys.frame(-2),
  useGit = getOption("SpaDES.project.useGit", FALSE),
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  ...
)
```

Arguments

name	Optional. If supplied, the name of the project. If not supplied, an attempt will be made to extract the name from the <code>paths[["projectPath"]]</code> . 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, <code>modulePath</code> , <code>projectPath</code> , <code>packagePath</code> and all others that are in <code>SpaDES.core::setPaths()</code> (i.e., <code>inputPath</code> , <code>outputPath</code> , <code>scratchPath</code> , <code>cachePath</code> , <code>rasterTmpDir</code>). 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 <code>paths[["projectPath"]]</code> .
standAlone	A logical. Passed to <code>Require::standAlone</code> . This keeps all packages installed in a project-level library, if TRUE. Default is <code>getOption("SpaDES.project.standAlone", TRUE)</code> , i.e., TRUE unless that option is set.
libPaths	Deprecated. Use <code>paths = list(packagePath = ...)</code> .
updateRprofile	Logical. Should the <code>paths\$packagePath</code> be set in the <code>.Rprofile</code> file for this project. Note: if <code>paths\$packagePath</code> is within the <code>tempdir()</code> , 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. Default is `getOption("SpaDES.project.updateRprofile", TRUE)`, i.e., TRUE unless that option is set.

Restart	Logical or character. If either TRUE or a character, and if the projectPath is not the current path, and the session is interactive and in RStudio or Positron, it will try to restart the IDE in the projectPath. If character, it should represent the filename of the script that contains the <code>setupProject</code> 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 there. Default is FALSE, and no restart is attempted. The two IDEs differ in what "the project" means, and <code>setupProject</code> follows each one's own convention: RStudio a project is a folder containing an .Rproj file. On a successful restart this creates that file (and an .Rproj.user folder) and opens a new RStudio session rooted at projectPath. Positron a project is simply an open workspace folder — Positron neither creates nor reads .Rproj files, so none is written. A new Positron window is opened on projectPath, which becomes the working directory, and the project .Rprofile is read as usual. Two limitations apply to Positron only. Re-opening the script after the restart uses the <code>positron.session_init</code> hook, which requires Positron 2026.04 or newer; on older versions everything else works and <code>setupProject</code> prints the path of the file to open by hand. And because Positron has no equivalent of RStudio's <code>~/ .active-rstudio-document</code>, an <i>unsaved</i> editor buffer cannot be recovered — save the script first, or pass its path as <code>Restart = "myScript.R"</code>.
overwrite	Logical vector or character vector, however, only <code>getModule</code> 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 <code>setupProject</code> – including those downloaded via <code>sideEffects</code>. If a length > 1 logical or character vector, these will be passed to <code>getModule</code>: only the named modules will be overwritten or the logical vector of the modules. NOTE: if length > 1, no other file specified anywhere in <code>setupProject</code> will be overwritten except a module matching the vector <code>names()</code> (because only <code>setupModules</code> is currently responsive to a vector). To have fine grained control, a user can just manually delete a file, then rerun. Default is <code>getOption("SpaDES.project.overwrite", FALSE)</code>, i.e., FALSE unless that option is set.
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., <code>setupProject</code>, then the <code>envir</code> should be the internal transient environment of that function.
callingEnv	The environment from which the function was called. Defaults to <code>sys.frame(-2)</code> which represents the case where the inner <code>setup*</code> functions are called inside <code>setupProject</code>, which was called by a user.
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 `git 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`). Default is `getOption("SpaDES.project.useGit", FALSE)`, i.e., FALSE unless that option is set.

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 <code>Require</code> function, when <code>verbose >= 2</code> , also returns details as if <code>returnDetails = TRUE</code> (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, written as a literal <code>list(...)</code> or as a variable holding one. These can be supplied to give default values to objects that are otherwise passed in with the <code>...</code> , i.e., not specifically named for these <code>setup*</code> functions. If named objects are supplied as top-level arguments, then the <code>defaultDots</code> will be overridden. This can be particularly useful if the arguments passed to <code>...</code> do not always exist, but rely on external e.g., batch processing to optionally fill them. In a literal <code>list(...)</code> an entry may reference an entry written above it. 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 <code>SpaDES.core::simInit</code> , or <code>SpaDES.core::simInitAndSpades</code> , (e.g. <code>studyAreaName</code> or objects) or additional objects to be passed to the simulation (in older versions of <code>SpaDES.core</code> , these were passed as a named list to the <code>objects</code> argument). Order matters. These are sequentially evaluated, and also any arguments that are specified before the named arguments e.g., <code>name</code> , <code>paths</code> , will be evaluated prior to any of the named arguments, i.e., "at the start" of the <code>setupProject</code> . If placed after the first named argument, then they will be evaluated at the end of the <code>setupProject</code> , so can access all the packages, objects, etc.

Details

`setupPaths` 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.

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.

Paths

Path	Default if not supplied by user
<i>Project Level Paths</i>	
projectPath	getOption("spades.projectPath") if set; otherwise, if getwd() is name, then just getwd; if not file.pat
packagePath	getOption("spades.packagePath", file.path(tools::R_user_dir("data"), name, "packages", ver
<i>Module Level Paths</i>	
cachePath	file.path(projectPath, "cache")
inputPath	file.path(projectPath, "inputs")
modulePath	file.path(projectPath, "modules")
outputPath	file.path(projectPath, "outputs")
<i>Temporary Paths</i>	
scratchPath	file.path(tempdir(), name)
rasterPath	file.path(scratchPath, "raster")
terraPath	file.path(scratchPath, "terra")
<i>Other Paths</i>	
logPath	file.path(outputPath(sim), "log")
tilePath	Not implemented yet

See Also

[setupProject\(\)](#) for the high-level wrapper, [setup_family](#) for an overview.

setupProject	<i>Sets up a new or existing SpaDES project</i>
--------------	---

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", FALSE),
  useGit = getOption("SpaDES.project.useGit", FALSE),
  setLinuxBinaryRepo = getOption("SpaDES.project.setLinuxBinaryRepo", TRUE),
  standAlone = getOption("SpaDES.project.standAlone", TRUE),
  libPaths = NULL,
  updateRprofile = getOption("SpaDES.project.updateRprofile", TRUE),
  overwrite = getOption("SpaDES.project.overwrite", FALSE),
  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 <code>paths[["projectPath"]]</code> . 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, <code>modulePath</code> , <code>projectPath</code> , <code>packagePath</code> and all others that are in <code>SpaDES.core::setPaths()</code> (i.e., <code>inputPath</code> , <code>outputPath</code> , <code>scratchPath</code> , <code>cachePath</code> , <code>rasterTmpDir</code>). 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 <code>getModule</code> . These should be one of: simple name (e.g., <code>fireSense</code>) which will be searched for locally in the <code>paths[["modulePath"]]</code> ; or a GitHub repo with branch (<code>GitHubAccount/Repo@branch</code> 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. For convenience, these can also be 2 other url formats: 1. the <code>raw.githubusercontent.com</code> url that points to the main module file or the folder e.g., "https://raw.githubusercontent.com/PredictiveEcology/Biomass_core/refs/" 2. The <code>github.com</code> url used for cloning a git repository, with optional "@branch" specified: "https://github.com/PredictiveEcology/Biomass_speciesParameters.git@deve" See setup.
packages	Optional. A vector of packages that must exist in the <code>libPaths</code> . This will be passed to <code>Require::Install</code> , i.e., these will be installed, but not attached to the search path. See also the <code>require</code> argument. To force skip of package installation (without assessing modules), set <code>packages = NULL</code> . To install everything from repos (e.g., prebuilt binaries on <code>predictiveecology.r-universe.dev</code>) rather than from GitHub source – thereby avoiding git authentication and a source-build toolchain such as <code>Rtools</code> – set <code>options = list(Require.noRemotes = TRUE)</code> . Any GitHub-style specs (<code>account/repo@branch</code>), including those declared in module metadata (<code>reqdPkgs</code>), are then rewritten to their bare package name (version constraints preserved) and resolved from repos. Ensure the relevant repository is in <code>getOption("repos")</code> and carries a version satisfying any constraint. See <code>Require::RequireOptions</code> .
times	Optional. This will be returned if supplied; if supplied, the values can be used in e.g., <code>params</code> , e.g., <code>params = list(mod = list(startTime = times\$start))</code> . Default (when not supplied) is <code>list(start = 0, end = 1)</code> . See help for <code>SpaDES.core::simInit</code> .
options	Optional. Either a named list to be passed to <code>options</code> or a character vector indicating one or more file(s) to source, in the order provided. These will be parsed locally (not the <code>.GlobalEnv</code>), so they will not create globally accessible objects. NOTE: <code>options</code> is run 2x within <code>setupProject</code> , once before <code>setupPaths</code> and once after <code>setupPackages</code> . This occurs because many packages use options for

	their behaviour (need them set before e.g., <code>Require::require</code> 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. If the project declares a <code>.rep</code> dot (the replicate column of an experiment table), as an argument or in <code>defaultDots</code> , <code>params\$.globals\$.rep</code> is set to its value (as integer), so modules with a <code>.rep</code> parameter receive it. The value is the one a dot resolves to: an object <code>.rep</code> the caller can see (e.g. the queue's column in a worker) wins over the <code>defaultDots</code> entry. A <code>.rep</code> the project does not declare, e.g. one left over in an interactive session, is ignored. A <code>.rep</code> set in <code>params\$.globals</code> is kept, and a module's own <code>params\$<module>\$.rep</code> overrides <code>.globals</code> , as for any global parameter. No other dot is passed on in this way. See setup .
sideEffects	Optional. This can be an expression or one or more file names or a code chunk surrounded by <code>{...}</code> . If a non-text file name is specified (e.g., <i>not .txt or .R</i> 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 (<code>*.txt</code> or <code>.R</code>), 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 <code>setupProject</code> . These will be returned as a list element. If function definitions require non-base packages, prefix the function call with the package e.g., <code>terra::rast</code> . When using <code>setupProject</code> , the <code>functions</code> argument is evaluated after paths, so it cannot be used to define functions that help specify paths.
config	Reserved for future use. Currently unimplemented; supplying a value triggers an error.
require	Optional. A character vector of packages to install <i>and</i> attach (with <code>Require::Require</code>). These will be installed and attached at the start of <code>setupProject</code> so that a user can use these during <code>setupProject</code> . Default is <code>NULL</code> , i.e., nothing additional installed/attached. See setup
studyArea	Optional. If a list, it will be passed to <code>geodata::gadm</code> . 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 <code>gadm</code> . 2 additional named elements can be passed for convenience, <code>subregion = "..."</code> , which will be grepped with the column <code>NAME_1</code> , and <code>epsg = "..."</code> , so a user can pass an <code>epsg.io</code> code to reproject the <code>studyArea</code> . Default is <code>NULL</code> , i.e., no study area is built. See examples.
Restart	Logical or character. If either <code>TRUE</code> or a character, and if the <code>projectPath</code> is not the current path, and the session is interactive and in RStudio or Positron, it will try to restart the IDE in the <code>projectPath</code> . If character, it should represent the filename of the script that contains the <code>setupProject</code> call that should be copied to the new folder and opened. If <code>TRUE</code> , it will use the active file as the one that should be copied to the new <code>projectPath</code> and opened there. Default is <code>FALSE</code> , and no restart is attempted. The two IDEs differ in what "the project" means, and <code>setupProject</code> follows each one's own convention:

RStudio a project is a folder containing an `.Rproj` file. On a successful restart this creates that file (and an `.Rproj.user` folder) and opens a new RStudio session rooted at `projectPath`.

Positron a project is simply an open workspace folder — Positron neither creates nor reads `.Rproj` files, so none is written. A new Positron window is opened on `projectPath`, which becomes the working directory, and the project `.Rprofile` is read as usual.

Two limitations apply to Positron only. Re-opening the script after the restart uses the `positron.session_init` hook, which requires **Positron 2026.04 or newer**; on older versions everything else works and `setupProject` prints the path of the file to open by hand. And because Positron has no equivalent of RStudio's `~/ .active-rstudio-document`, an *unsaved* editor buffer cannot be recovered — save the script first, or pass its path as `Restart = "myScript.R"`.

`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 `git 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`). Default is `getOption("SpaDES.project.useGit", FALSE)`, i.e., `FALSE` unless that option is set.

`setLinuxBinaryRepo`

Logical. Should the binary RStudio Package Manager be used on Linux (ignored if Windows). Default is `getOption("SpaDES.project.setLinuxBinaryRepo", TRUE)`, i.e., `TRUE` unless that option is set.

`standAlone`

A logical. Passed to `Require::standAlone`. This keeps all packages installed in a project-level library, if `TRUE`. Default is `getOption("SpaDES.project.standAlone", TRUE)`, i.e., `TRUE` unless that option is set.

`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. Default is `getOption("SpaDES.project.updateRprofile", TRUE)`, i.e., `TRUE` unless that option is set.

overwrite	Logical vector or character vector, however, only <code>getModule</code> 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 <code>setupProject</code> – including those downloaded via <code>sideEffects</code> . If a length > 1 logical or character vector, these will be passed to <code>getModule</code> : only the named modules will be overwritten or the logical vector of the modules. NOTE: if length > 1, no other file specified anywhere in <code>setupProject</code> will be overwritten except a module matching the vector names() (because only <code>setupModules</code> is currently responsive to a vector). To have fine grained control, a user can just manually delete a file, then rerun. Default is <code>getOption("SpaDES.project.overwrite", FALSE)</code> , i.e., FALSE unless that option is set.
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 <code>Require</code> function, when <code>verbose >= 2</code> , also returns details as if <code>returnDetails = TRUE</code> (for backwards compatibility).
defaultDots	A named list of any arbitrary R objects, written as a literal <code>list(...)</code> or as a variable holding one. These can be supplied to give default values to objects that are otherwise passed in with the <code>...</code> , i.e., not specifically named for these <code>setup*</code> functions. If named objects are supplied as top-level arguments, then the <code>defaultDots</code> will be overridden. This can be particularly useful if the arguments passed to <code>...</code> do not always exist, but rely on external e.g., batch processing to optionally fill them. In a literal <code>list(...)</code> an entry may reference an entry written above it. See examples.
envir	The environment where <code>setupProject</code> is called from. Defaults to <code>parent.frame()</code> 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 <code>SpaDES.core::simInit</code> , or <code>SpaDES.core::simInitAndSpades</code> , (e.g. <code>studyAreaName</code> or objects) or additional objects to be passed to the simulation (in older versions of <code>SpaDES.core</code> , these were passed as a named list to the <code>objects</code> argument). Order matters. These are sequentially evaluated, and also any arguments that are specified before the named arguments e.g., <code>name</code> , <code>paths</code> , will be evaluated prior to any of the named arguments, i.e., "at the start" of the <code>setupProject</code> . If placed after the first named argument, then they will be evaluated at the end of the <code>setupProject</code> , 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 remaining 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)
  }
)
```

Argument order (evaluation sequence): The evaluation rules are:

- **Each ... argument is evaluated once, in the order it is written.** The formal/named arguments (`paths`, `modules`, `options`, `params`, `times`, ...) are each evaluated *in their entirety as a block*. So if a user needs an object to exist before, e.g., `paths` is evaluated, they should declare that argument *before* `paths`.
- **Because of this sequential evaluation, any argument written *above* another is available to it**, exactly like a normal series of statements. A later argument can refer to the resolved value of an earlier one by name.
- **Every `defaultDots` entry the caller did not supply is bound, as a value, before any argument is evaluated.** A value the caller *does* supply always takes precedence over the `defaultDots` fallback; a name that only resolves to a function in a package namespace does not count as caller-supplied. This is what makes a pattern such as `.samplingRange = unlist(.samplingRange)` (with `defaultDots = list(.samplingRange = 1990:2020)`) resolve to the caller's value when one is supplied (e.g. a batch run that sets `.samplingRange` before calling), and to the default otherwise. Because the defaults are bound up front, any argument can reference them – a ... argument under a different name (`cores = .cores`) or a formal (`times = as.list(unlist(.times))`).
- **A dot's own name is not visible to its own expression** unless the caller defined it or `defaultDots` supplies it, so `.x = if (exists(".x")) .x else <fallback>` behaves as written. All of this happens in one scope environment whose parent is the calling environment (`envir`); only values are ever bound there, never unevaluated expressions.

If "dot" arguments are declared before the first formal argument, they are evaluated before the formals; otherwise they are evaluated after the formals. For this purpose `params`, `studyArea`, `times`

and defaultDots do not count as formals: params is evaluated in order like a ... argument, and defaultDots is the fallback table the dots are resolved against, not a configuration block – so a dot written after defaultDots (the natural place, since that is where its fallback lives) is still resolved before paths, modules and the other formals, and can be used inside them.

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@development/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., `prefixRequire::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 do 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 build the same argument's (i.e. paths, params, options) final list. Hence, in an file to be 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"
                    )
```

Verbosity

`verbose` is passed through to the inner `setup*` helpers. Notably, `verbose >= 2` prints the modules' `reqdPkgs` grouped by module, and `verbose >= 3` additionally prints the `dput()` of the exact package vector passed to `Require::Require` (see [setupPackages\(\)](#)).

See Also

Inner `setup*` helpers (each has its own help page; see [setup_family](#) for a one-page overview): [setupPaths\(\)](#), [setupFunctions\(\)](#), [setupSideEffects\(\)](#), [setupOptions\(\)](#), [setupModules\(\)](#), [setupPackages\(\)](#), [setupParams\(\)](#), [setupGitIgnore\(\)](#), [setupStudyArea\(\)](#), [setupFiles\(\)](#). [teardownProject\(\)](#) reverses `setupProject()` and restores the prior `.libPaths()` (kept on the output as `out$paths$.previousLibPaths`). 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 = ".") #
)
```

setupSideEffects	<i>Run side-effect scripts (e.g., authentication, custom package options)</i>
------------------	---

Description

Source the side-effect scripts or expressions supplied to `setupProject()`; nothing is returned to the user.

Usage

```
setupSideEffects(
  name,
  sideEffects,
  paths,
  times,
  overwrite = FALSE,
  envir = parent.frame(),
  callingEnv = sys.frame(-2),
  verbose = getOption("Require.verbose", 1L),
  dots,
  defaultDots,
  ...
)
```

Arguments

- | | |
|-------------|--|
| name | Optional. If supplied, the name of the project. If not supplied, an attempt will be made to extract the name from the <code>paths[["projectPath"]]</code> . If this is a GitHub project, then it should indicate the full Github repository and branch name, e.g., "PredictiveEcology/WBI_forecasts@ChubatyPubNum12" |
| sideEffects | Optional. This can be an expression or one or more file names or a code chunk surrounded by <code>{...}</code> . If a non-text file name is specified (e.g., <i>not .txt or .R</i> 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.

paths	a list with named elements, specifically, modulePath, projectPath, packagePath and all others that are in <code>SpaDES.core::setPaths()</code> (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 .
times	Optional. This will be returned if supplied; if supplied, the values can be used in e.g., <code>params</code> , e.g., <code>params = list(mod = list(startTime = times\$start))</code> . Default (when not supplied) is <code>list(start = 0, end = 1)</code> . See help for <code>SpaDES.core::simInit</code> .
overwrite	Logical vector or character vector, however, only <code>getModule</code> 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 <code>setupProject</code> – including those downloaded via <code>sideEffects</code> . If a length > 1 logical or character vector, these will be passed to <code>getModule</code> : only the named modules will be overwritten or the logical vector of the modules. NOTE: if length > 1, no other file specified anywhere in <code>setupProject</code> will be overwritten except a module matching the vector <code>names()</code> (because only <code>setupModules</code> is currently responsive to a vector). To have fine grained control, a user can just manually delete a file, then rerun. Default is <code>getOption("SpaDES.project.overwrite", FALSE)</code> , i.e., FALSE unless that option is set.
envir	The environment where <code>setupProject</code> is called from. Defaults to <code>parent.frame()</code> which should be fine in most cases and user shouldn't need to set this
callingEnv	The environment from which the function was called. Defaults to <code>sys.frame(-2)</code> which represents the case where the inner <code>setup*</code> functions are called inside <code>setupProject</code> , which was called by a user.
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 <code>Require</code> function, when <code>verbose >= 2</code> , also returns details as if <code>returnDetails = TRUE</code> (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, written as a literal <code>list(...)</code> or as a variable holding one. These can be supplied to give default values to objects that are otherwise passed in with the <code>...</code> , i.e., not specifically named for these <code>setup*</code> functions. If named objects are supplied as top-level arguments, then the <code>defaultDots</code> will be overridden. This can be particularly useful if the arguments passed to <code>...</code> do not always exist, but rely on external e.g., batch processing to optionally fill them. In a literal <code>list(...)</code> an entry may reference an entry written above it. 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 <code>SpaDES.core::simInit</code> , or <code>SpaDES.core::simInitAndSpades</code> ,

(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.

Details

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*.

Value

`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.

See Also

[setupProject\(\)](#) for the high-level wrapper, [setup_family](#) for an overview.

<code>setupStudyArea</code>	<i>Resolve a study area from a <code>studyArea spec</code> via <code>geodata::gadm()</code></i>
-----------------------------	---

Description

Convenience wrapper that returns an `sf` polygon for the requested country / subregion using `geodata::gadm()`.

Usage

```
setupStudyArea(
  studyArea,
  paths,
  envir = parent.frame(),
  callingEnv = sys.frame(-2),
  verbose = getOption("Require.verbose", 1L)
)
```

Arguments

studyArea	Optional. If a list, it will be passed to <code>geodata::gadm</code> . 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 <code>gadm</code> . 2 additional named elements can be passed for convenience, <code>subregion = "..."</code> , which will be grepped with the column <code>NAME_1</code> , and <code>epsg = "..."</code> , so a user can pass an <code>epsg.io</code> code to reproject the <code>studyArea</code> . Default is <code>NULL</code> , i.e., no study area is built. See examples.
paths	a list with named elements, specifically, <code>modulePath</code> , <code>projectPath</code> , <code>packagePath</code> and all others that are in <code>SpaDES.core::setPaths()</code> (i.e., <code>inputPath</code> , <code>outputPath</code> , <code>scratchPath</code> , <code>cachePath</code> , <code>rasterTmpDir</code>). Each of these has a sensible default, which will be overridden but any user supplied values. See setup .
envir	The environment where <code>setupProject</code> is called from. Defaults to <code>parent.frame()</code> which should be fine in most cases and user shouldn't need to set this
callingEnv	The environment from which the function was called. Defaults to <code>sys.frame(-2)</code> which represents the case where the inner <code>setup*</code> functions are called inside <code>setupProject</code> , which was called by a user.
verbose	Numeric or logical indicating how verbose should the function be. If -1 or -2, then as little verbosity as possible. If 0 or <code>FALSE</code> , then minimal outputs; if 1 or <code>TRUE</code> , more outputs; 2 even more. NOTE: in <code>Require</code> function, when <code>verbose >= 2</code> , also returns details as if <code>returnDetails = TRUE</code> (for backwards compatibility).

Details

`setupStudyArea` calls `[geodata::gadm()]` to get an `sf` polygon or set of polygons of a country or a subdivision of a country. The user can pass a named list of character elements that match entries in the columns "NAME_1" and "NAME_2" of the `sf` object. If passing `NAME_2`, the user must pass `level = 2`. If passing `NAME_1` and `level = 2` all subdivision polygons under `NAME_1` will be returned, which can be useful to explore subdivision names. `setupStudyArea` only uses `inputPath` within its `paths` argument, which will be passed to `path` argument of `gadm`.

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

Value

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

See Also

[setupProject\(\)](#) for the high-level wrapper, [setup_family](#) for an overview.

show,simLists-method *Show method for simLists*

Description

Show method for simLists

Usage

```
## S4 method for signature 'simLists'  
show(object)
```

Arguments

object simLists

Author(s)

Eliot McIntire

simInitAndExperiment *Run simInit and experiment in one step*

Description

Run simInit and experiment in one step

Usage

```
simInitAndExperiment(  
  times,  
  params,  
  modules,  
  objects,  
  paths,  
  inputs,  
  outputs,  
  loadOrder,  
  notOlderThan,  
  replicates,  
  dirPrefix,  
  substrLength,  
  saveExperiment,  
  experimentFile,  
  clearSimEnv,  
  cl,  
  ...  
)
```

Arguments

times, paths, outputs, loadOrder	Passed to <code>SpaDES.core::simInit()</code> ; see there.
params	Like for <code>SpaDES.core::simInit()</code> , but for each parameter, provide a list of alternative values.
modules	Like for <code>SpaDES.core::simInit()</code> , but a list of module names (as strings).
objects	Like for <code>SpaDES.core::simInit()</code> , but a list of named lists of named objects.
inputs	Like for <code>SpaDES.core::simInit()</code> , but a list of inputs data.frames.
notOlderThan	Currently unused (kept for back-compatibility).
replicates	The number of replicates to run of the same simList.
dirPrefix	String vector. This will be concatenated as a prefix on the directory names.
substrLength	Numeric. While making outputPath for each spades call, this is the number of characters kept from each factor level.
saveExperiment	Logical. Should the resulting experimental design be saved to a file. Default TRUE.
experimentFile	String. Filename if saveExperiment is TRUE; saved to outputPath(sim) in .RData format.
clearSimEnv	Logical. If TRUE, then the <code>envir(sim)</code> of each simList in the return is emptied, to reduce RAM load. Default FALSE.
cl	Deprecated and ignored; control parallelism with <code>future::plan()</code> .
...	Passed to <code>experiment2()</code> and onward to <code>SpaDES.core::spades()</code> (e.g. debug, <code>.plotInitialTime</code> , cache, and events – see Controlling events in <code>experiment2()</code>).

Details

`simInitAndExperiment` cannot pass modules or params to `experiment` because these are also in `simInit`. If the experiment is being used to vary these arguments, it must be done separately (i.e., `simInit` then `experiment`).

Moved here from the now-unmaintained `SpaDES.experiment` package.

simLists-class

The simLists class

Description

This is a grouping of `simList` objects. Normally this class will be made using `experiment2()`, but can be made manually if there are existing `simList` objects.

Details

This class (and the `experiment()` / `experiment2()` functions that produce it) was moved here from the now-unmaintained `SpaDES.experiment` package.

Slots

`paths` Named list of `modulePath`, `inputPath`, and `outputPath` paths. Partial matching is performed. These will be prepended to the relative paths of each `simList`

`.xData` Environment holding the `simLists`.

Author(s)

Eliot McIntire

`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, and can mostly be specified in the various `setup*` functions also. The *DEFAULT VALUE* column lists the value returned by `spadesProjectOptions()`, which is also the fallback used by `setupProject()` (and the inner `setup*` functions) when the option is not otherwise set.

<i>OPTION</i>	<i>DEFAULT VALUE</i>	<i>DESCRIPTION</i>
<code>reproducible.cachePath</code>	<code><projectPath>/cache</code>	NOTE: uses <code>reproducible</code> . Within
<code>spades.inputPath</code>	<code><projectPath>/inputs</code>	Within <code>projectPath</code> , with subfolder
<code>spades.modulePath</code>	<code><projectPath>/modules</code>	Within <code>projectPath</code> , with subfolder
<code>spades.outputPath</code>	<code><projectPath>/outputs</code>	Within <code>projectPath</code> , with subfolder
<code>spades.packagePath</code>	<code>.libPathDefault(<projectPath>)</code>	The project-level package library
<code>spades.projectPath</code>	<code>"."</code>	The project root (current directory by
<code>spades.scratchPath</code>	<code><tempdir()>/<projectPath></code>	Within <code>tempdir()</code> , with subfolder b
<code>SpaDES.project.Restart</code>	<code>FALSE</code>	Passed to the <code>Restart</code> argument in s
<code>SpaDES.project.useGit</code>	<code>FALSE</code>	Passed to the <code>useGit</code> argument in se
<code>SpaDES.project.setLinuxBinaryRepo</code>	<code>TRUE</code>	Passed to the <code>setLinuxBinaryRepo</code>
<code>SpaDES.project.standAlone</code>	<code>TRUE</code>	Passed to the <code>standAlone</code> argument
<code>SpaDES.project.updateRprofile</code>	<code>TRUE</code>	Passed to the <code>updateRprofile</code> argum
<code>SpaDES.project.overwrite</code>	<code>FALSE</code>	Passed to the <code>overwrite</code> argument i
<code>SpaDES.project.ask</code>	<code>TRUE</code>	If <code>FALSE</code> , do not prompt before cloni

SpaDES.project.gitignore	TRUE	Whether setupProject manages a .gitignore
SpaDES.project.fast	FALSE	If TRUE, use fastOptions() and skip setupProject()
SpaDES.project.checkProject	FALSE	If TRUE, setupProject() runs checkProject()
SpaDES.project.checkProject.minFreeGB	50	Free disk space (GB) below which checkProject() fails

Value

named list of the *default* options currently available.

statusCalculate_family
<i>Pre-built statusCalculate expressions for experimentTmux / experimentFuture</i>

Description

A family of ready-made quoted expressions for the statusCalculate argument of `experimentTmux()` and `experimentFuture()`. Pass one of these objects directly instead of writing a custom `quote({ ... })` block:

```
experimentTmux(..., statusCalculate = statusCalculate_LandR)
```

Each expression is evaluated once per queue row inside `tmuxRefreshQueueStatus()`. Before evaluation the row's non-meta columns are unpacked into the local environment by name, as are any objects forwarded through `...`. The expression may assign to any subset of the recognised meta-column names (`started_at`, `finished_at`, `heartbeat_at`, `heartbeat_iter`, `iterationsTotal`, ...) and should set `done <- TRUE` to signal that the job has completed.

Usage

```
statusCalculate_FireSenseFit
statusCalculate_LandR
```

Format

A `base::quote()`d block expression (`is.call(statusCalculate_FireSenseFit)` is TRUE).
A `base::quote()`d block expression (`is.call(statusCalculate_LandR)` is TRUE).

Functions

- `statusCalculate_FireSenseFit`: Heartbeat calculator for **fireSense** fire-spread simulations. Scans the job's output directory for `burnMap_year<XXXX>.tif` files and "Annual Fire Maps" output files to populate the queue meta-columns:
`heartbeat_iter` Most recent fire-map year found after the worker claimed the job, or `times$start` if none yet.

heartbeat_at Modification timestamp of that file (NA until the first fire-map appears).
 started_at Modification timestamp of the running-flag file (i.e. when the worker claimed the job).
 done TRUE when a burnMap file containing year<times\$end> is found.
 finished_at Timestamp of the final-year burnMap (set only when done).
 iterationsTotal The end year extracted from the burnMap filename (set only when done).

- statusCalculate_LandR: Heartbeat calculator for **LandR** vegetation simulations.
 Scans the job's output directory for cohortData_year<XXXX>.rds checkpoint files (written at each SpaDES save event) and maps them to the standard queue meta-columns:

heartbeat_iter Current simulation year reached (character).
 heartbeat_at Timestamp of the latest checkpoint file.
 started_at Timestamp of the earliest checkpoint file (may be refined later by the running-flag-file logic in [tmuxRefreshQueueStatus\(\)](#)).
 done Set to TRUE when heartbeat_iter >= outs\$times\$end, triggering a status transition to DONE.
 finished_at Timestamp of the final checkpoint (set only when done).
 iterationsTotal The end year as a character string (set only when done).

Variables required in scope

The expressions below expect the following to be available, either as queue-data-frame columns or as named objects in the ... passed to [tmuxRefreshQueueStatus\(\)](#):

pathBuild A function whose arguments match the queue columns used to construct the output-directory path. For statusCalculate_LandR the call is pathBuild(.ELFind, .samplingRange, .GCM, .SSP, .rep).

outs A list (typically stored in dots_path and loaded into the worker's environment before global.R is sourced) whose element outs\$times\$end gives the simulation end year.

Variables required in scope (statusCalculate_FireSenseFit only)

times A list with elements \$start and \$end giving the simulation start and end years (integers). Typically a queue column.

See Also

[experimentTmux\(\)](#), [experimentFuture\(\)](#), [tmuxRefreshQueueStatus\(\)](#), [get_sim_year_heartbeat\(\)](#)

teardownProject	<i>Tear down a project created by setupProject()</i>
-----------------	--

Description

Reverse the side-effects of [setupProject\(\)](#):

Usage

```
teardownProject(x, origLibPaths)
```

Arguments

<code>x</code>	Either the list returned by setupProject() , or a character vector of paths to remove (back-compat with the previous <code>.teardownProject(prjPaths, origLibPaths)</code> signature).
<code>origLibPaths</code>	Optional. The <code>.libPaths()</code> to restore. Defaults to <code>x\$paths\$.previousLibPaths</code> when <code>x</code> is a <code>setupProject()</code> output, so most callers will not need to supply this.

Details

1. remove the project library directory created by `setupProject()`,
2. unlink the project paths returned by `setupProject()`,
3. restore the `.libPaths()` value that was in effect before `setupProject()` was called.

The previous `.libPaths()` is stored on the `setupProject()` output as `out$paths$.previousLibPaths` (and on `attr(out$paths, "extraPaths")`), so `teardownProject(out)` is enough – no need to remember `origLibPaths` separately.

Value

NULL, invisibly. Called for its side effects.

See Also

[setupProject\(\)](#) for what is being torn down.

tmuxActiveRunningPath *Log path for default tmux status*

Description

Just a default path.

Usage

```
tmuxActiveRunningPath(
    activeRunningPath = NULL,
    queue_path,
    prefix = "logs",
    suffix = queue_path
)
```

Arguments

activeRunningPath	Optional character path. If NULL (default), derived from prefix and queue_path.
queue_path	Character. Path to the queue .rds file, used to derive the default path.
prefix	Character. Directory name placed beside the queue file. Default "logs".
suffix	Character. Suffix used in the path. Defaults to queue_path.

Value

The default path.

tmuxFindDuplicates *Find duplicate worker panes running the same job*

Description

Strips the leading "<host?>-<node>-<pid>-" prefix from each pane title and groups panes whose remainders are identical. Intended to surface cases where the same queue row has been claimed by two workers (e.g. a stale RUNNING reclaim that was actually live).

Usage

```
tmuxFindDuplicates(panes = NULL, runPattern = "outputs-")
```

Arguments

panes	Optional data.frame as returned by <code>tmuxListPanes()</code> . If NULL (the default) one is fetched internally.
runPattern	Optional regex; only panes whose stripped title matches it are considered. Default "outputs-" matches this codebase's usual runName prefix; pass NULL to disable the filter.

Details

The prefix strip matches 1 or 2 non-dash chunks followed by a 6+-digit PID followed by a dash – covering both <host>-<node>-<pid>-<runName> and <node>-<pid>-<runName> title formats. Old-style titles lacking this prefix are kept verbatim; a title is considered a duplicate only if its stripped form appears on 2+ panes, so two differently-formatted titles with the same tail still collapse correctly.

Value

data.frame with the same columns as `tmuxListPanes()` plus `run_id` (the stripped runName used for grouping) and `group` (integer identifying each duplicate set). Rows are ordered by group then `pane_ref`. Empty data.frame (with these columns) when no duplicates.

tmuxKillPanes	<i>Kill a set of tmux panes (e.g., those spawned by <code>experimentTmux()</code>)</i>
---------------	--

Description

Development utility: kills all panes identified by their tmux pane IDs. Uses `kill-pane -t <pane-id>`; panes already gone are ignored. See tmux manual. **1**

Usage

```
tmuxKillPanes(panes)
```

Arguments

panes	Character vector of tmux pane IDs (e.g., <code>c("%2", "%3")</code>) returned by <code>experimentTmux()</code> .
-------	---

Value

Invisibly returns the subset of panes successfully targeted.

tmuxListPanes	<i>List all tmux panes on this machine across every tmux server</i>
---------------	---

Description

Thin alias for `experimentMonitor()` in tmux-scan mode (no ef / queue_paths). Preserved for backwards compatibility; new code should call `experimentMonitor()` directly so the same call works for `experimentFuture()` / `experimentSBATCH()` runs by passing ef.

Usage

```
tmuxListPanes(stats = FALSE)
```

Arguments

stats	Logical. When TRUE, queries ps per worker (locally or via batched SSH) to append state, cpuAvg (percent CPU averaged over the process's lifetime – not the instantaneous rate htop shows), RAM (GB) (resident memory), availableCores (total CPUs on the node, from nproc), and total RAM (GB) (total RAM on the node, from /proc/meminfo). Default FALSE.
-------	--

Value

Same as `experimentMonitor(stats = stats)` in tmux mode – see that function's docs.

tmuxMirrorQueueToSheets	<i>Mirror local queue to Google Sheets</i>
-------------------------	--

Description

Mirror local queue to Google Sheets

Usage

```
tmuxMirrorQueueToSheets(queue_path, ss_id, sheet_name = "Status")
```

Arguments

queue_path	Path to the local tmux_queue.rds
ss_id	The Google Sheet ID (from the URL)
sheet_name	The name of the tab to write to

tmuxPrepareQueueFromDF

Initialize a file-backed queue from a data.frame

Description

Mirrors df into a queue RDS and adds status columns: status, claimed_by, started_at, finished_at.

Adds metadata columns used by workers:

- status: PENDING | RUNNING | DONE | FAILED
- claimed_by: tmux pane id that claimed the row
- started_at: "YYYY-MM-DD HH:MM:SS"
- finished_at: "YYYY-MM-DD HH:MM:SS"
- DEoptimElapsedTime: numeric seconds (sum(diff(allIterations[allIterations < 20 minutes])))
- machine_name: Sys.info()[["nodename"]]
- process_id: Sys.getpid()
- heartbeat_at: latest timestamp (as character) detected by heartbeat
- heartbeat_iter: latest iteration number (integer) detected by heartbeat

Usage

```
tmuxPrepareQueueFromDF(df, queue_path)
```

```
tmuxPrepareQueueFromDF(df, queue_path)
```

Arguments

df	data.frame; experiment rows (columns become objects in workers)
queue_path	character; path to the queue .rds (absolute recommended)

Value

Invisibly returns queue_path.

Invisibly returns queue_path.

tmuxReconcileQueueWithDF

Reconcile an existing file-backed queue with a freshly built df

Description

`experimentTmux()` materialises the queue from `df` only when `queue_path` does not yet exist; an existing queue is authoritative, so a resumed run keeps its DONE/RUNNING state instead of restarting finished work. That rule is right, but it used to be applied *silently*: a caller who had carefully rebuilt `df` – new scenarios added, finished ones dropped – would watch the previous queue run instead, with nothing to distinguish that from success. The only way to act on the new `df` was to notice, and delete or rename the file.

This keeps the rule and makes it visible, and adds the middle option that was missing: take the existing queue *and* the rows `df` adds.

Rows are compared on the non-meta_cols, non-list columns – the scenario fields. List columns (`.modules`, `.times`) are payload, not identity.

Usage

```
tmuxReconcileQueueWithDF(
  df,
  queue_path,
  onExistingQueue = c("resume", "append", "rebuild")
)
```

Arguments

`df` data.frame of runs, as passed to `experimentTmux()`.

`queue_path` Character path to the queue .rds.

`onExistingQueue`

What to do when `queue_path` already exists:

- "resume" (default) – keep the existing queue; warn if `df` holds rows it does not, naming them and saying how to act on them;
- "append" – add those rows as PENDING, leaving existing rows and their status untouched;
- "rebuild" – discard the existing queue and start again from `df`.

Value

Invisibly, `queue_path`.

tmuxRefreshQueueStatus

Refresh and Assess Queue Status from Simulation Outputs

Description

Scans the simulation output directories (defined by `runNameLabel`) to assess current status based on file timestamps and visual content of PNGs. If a PNG has not been updated for a specified timeout, the task is marked as "FINISHED" (if red pixels are detected) or "INTERRUPTED" (if no red is detected).

Usage

```
tmuxRefreshQueueStatus(
  queue_path,
  timeout_min = 20,
  runNameLabel = quote(colnames(q)[1:2]),
  statusCalculate = getOption("spades.statusCalculate"),
  folderWithIterInFilename = getOption("spades.folderWithIterInFilename"),
  recheckDone = FALSE,
  activeRunningPath = getOption("spades.activeRunningPath"),
  ...
)
```

Arguments

<code>queue_path</code>	Character. Absolute path to the <code>experiment_queue.rds</code> file.
<code>timeout_min</code>	Numeric. Minutes of inactivity before a task is considered stale. Defaults to 20.
<code>runNameLabel</code>	A quoted expression to derive a run label from the queue. Default uses first two columns.
<code>statusCalculate</code>	A quoted expression to compute job status from output files. Defaults to <code>getOption("spades.statusCalculate", NULL)</code> .
<code>folderWithIterInFilename</code>	A quoted expression for a folder with iteration info in filenames. Defaults to <code>getOption("spades.folderWithIterInFilename", NULL)</code> .
<code>recheckDone</code>	Logical. If TRUE, re-evaluate DONE status. Default FALSE.
<code>activeRunningPath</code>	Directory for "running" flag files. See <code>tmuxActiveRunningPath</code> .
<code>...</code>	Additional arguments (currently unused).

Value

A `data.frame` (the updated queue), invisibly. As a side effect, updates the RDS file on disk.

Examples

```
## Not run:
# Assessment of all simulations in the current project
tmuxRefreshQueueStatus("experiment_queue.rds", timeout_min = 30)

## End(Not run)
```

tmuxRunNextWorker	<i>Run one queued job (claim-next semantics) in the current R session.</i>
-------------------	--

Description

Run one queued job (claim-next semantics) in the current R session.

Usage

```
tmuxRunNextWorker(
  queue_path,
  global_path,
  on_interrupt = c("requeue", "fail"),
  heartbeat_interval_s = 60,
  runNameLabel = quote(colnames(q)[1:2]),
  statusCalculate = getOption("spades.statusCalculate"),
  folderWithIterInFilename = getOption("spades.folderWithIterInFilename"),
  activeRunningPath = getOption("spades.activeRunningPath"),
  ss_id = NULL
)
```

Arguments

queue_path	character; path to the queue .rds
global_path	character; script to source for the job
on_interrupt	"requeue" or "fail". If the sourced script is interrupted, either requeue or mark as INTERRUPTED. A script that errors leaves the row "ERROR", which is not claimed again until set back to "PENDING".
heartbeat_interval_s	numeric; seconds between heartbeats while the job runs
runNameLabel	A quoted expression (possibly of q, which is the result of <code>q <- readRDS(queue_path)</code>). Default is the first 2 column names of q. These will be concatenated and used as labels for various things including the activeRunningPath file(s).
statusCalculate	A quoted expression to compute job status from output files. Defaults to <code>getOption("spades.statusCalculate")</code> .
folderWithIterInFilename	A quoted expression for a folder containing iteration info in filenames. Defaults to <code>getOption("spades.folderWithIterInFilename", NULL)</code> .

activeRunningPath	Directory for "running" flag files. See tmuxActiveRunningPath.
ss_id	Optional Google Sheets/Drive ID for the shared queue. When supplied workers use the GS backend instead of the local RDS file.

Value

"ok" | "interrupt" | "empty" (if no pending work found); used by tmuxRunWorkerLoop()

tmuxRunWorkerLoop	<i>Run queued jobs repeatedly (pane-local loop).</i>
-------------------	--

Description

Run queued jobs repeatedly (pane-local loop).

Usage

```
tmuxRunWorkerLoop(
  queue_path,
  global_path,
  on_interrupt = c("requeue", "fail"),
  on_error = c("stop", "continue"),
  heartbeat_interval_s = 60,
  stop_file = NULL,
  activeRunningPath = getOption("spades.activeRunningPath"),
  runNameLabel = quote(colnames(q)[1:2]),
  ss_id = NULL,
  pane_mode = c("reuse", "killAndNewPane"),
  email = getOption("gargle_oauth_email"),
  cache_path = getOption("gargle_oauth_cache"),
  dots_path = NULL,
  snapshot_library = TRUE,
  n_workers_on_host = 1L
)
```

Arguments

queue_path	character; path to the queue .rds
global_path	character; script to source for the job
on_interrupt	"requeue" or "fail". If the sourced script is interrupted, either requeue or mark as INTERRUPTED. A script that errors leaves the row "ERROR", which is not claimed again until set back to "PENDING".
on_error	"stop" (default) or "continue". After a job errors, "stop" leaves this worker at the prompt with the error and call stack available; "continue" goes on to the next claimable job. The errored row is marked "ERROR" either way, and a worker with nothing claimable left (only ERROR, DONE or HOLD rows) ends.

heartbeat_interval_s	numeric; seconds between heartbeats while the job runs
stop_file	optional path; if present, stop after current iteration
activeRunningPath	Directory for "running" flag files. See <code>tmuxActiveRunningPath</code> .
runNameLabel	A quoted expression (possibly of <code>q</code> , which is the result of <code>q <- readRDS(queue_path)</code>). Default is the first 2 column names of <code>q</code> . These will be concatenated and used as labels for various things including the <code>activeRunningPath</code> file(s).
ss_id	Optional Google Sheets/Drive ID for the shared queue. When supplied workers use the GS backend instead of the local RDS file.
pane_mode	Character. "reuse" (default) loops inside the same R session. "killAndNewPane" runs one job, spawns a fresh replacement pane, retiles the tmux window, then kills the current pane – freeing all R memory between jobs.
email	gargle OAuth email; forwarded to replacement panes in <code>killAndNewPane</code> mode.
cache_path	gargle OAuth cache path; forwarded to replacement panes.
dots_path	Path to <code>.tmux_dots.rds</code> holding extra ... args; forwarded to replacement panes so they can reload complex objects before sourcing.
snapshot_library	Logical. Forwarded to replacement panes: each new worker session takes a hardlinked snapshot of the project library before loading anything from it (see librarySnapshot). This session's own snapshot, if the startup script made one, is released when R exits.
n_workers_on_host	Number of workers the launcher runs on this host. This session's GDAL block cache is capped at its share of the host's RAM, and terra results go to disk, unless the user set these; see <code>.workerResourceDefaults()</code> .

Value

invisibly TRUE

tmuxSetMouse	<i>Enable or disable tmux mouse interaction</i>
--------------	---

Description

Sets tmux mouse mode via `set-option -g mouse on/off`, enabling pane selection, resizing, and scrolling with the mouse. See tmux manual for details. [1](#)

Usage

```
tmuxSetMouse(on = TRUE)
```

Arguments

`on` Logical; TRUE to enable, FALSE to disable. Default TRUE.

Value

Invisibly returns on.

tmuxSetPaneTitle	<i>Set a tmux pane's title by matching its current title</i>
------------------	--

Description

Scans every tmux server on this machine (sockets under \$TMUX_TMPDIR/tmux-<uid>/) for panes whose current title exactly matches oldTitle, then rewrites each to newTitle. Useful for upgrading old-style worker-pane titles (without <node>-<pid> prefix) to the new convention so that .gs_reclaim_dead_jobs() can recognise them.

Usage

```
tmuxSetPaneTitle(oldTitle, newTitle)
```

Arguments

oldTitle	Character(1). Exact current title to match.
newTitle	Character(1). Replacement title.

Value

Invisibly, a character vector of the pane IDs that were updated (e.g. c("%12", "%33")). Prints a message per update and a warning when no match is found.

user	<i>Helpers to develop easier to understand code.</i>
------	--

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., <code>"^machine1"</code> will match <code>"machine15"</code> and <code>"machine12"</code> , but not <code>"thisIs_machine1"</code> .

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

.libPathDefault, 5
.teardownProject (teardownProject), 124

as.data.table.simLists, 5
as.data.table.simLists(), 16, 23, 24
as_path, 6
as_path(), 83
as_scenario, 7
as_scenario(), 7, 8, 62, 73, 83
as_tarname, 7
as_tarname(), 74, 83
assessDoneInFigure, 8
awaitExperimentFuture, 8, 26, 28, 50
awaitExperimentsSBATCH, 9, 34, 36, 51
awaitExperimentsSBATCH(), 17

base::quote(), 122
base::readRDS(), 78
base::traceback(), 52

checkProject, 9

dataInventory, 12
dataInventory(), 9–11

experiment, 14
experiment(), 16, 22–24, 45, 120
experiment2, 23
experiment2(), 14–16, 22, 120
experiment_family, 16, 16, 24, 45
experimentFuture, 9, 26, 30, 32, 34, 36, 50, 81
experimentFuture(), 16, 17, 22, 24, 33, 73, 122, 123, 127
experimentFutureList, 20, 30
experimentFutureList(), 18, 22, 34
experimentMonitor, 32
experimentMonitor(), 21, 22, 127
experimentsSBATCH, 9, 34, 51
experimentsSBATCH(), 16, 17, 22, 33, 73, 127
experimentTmux, 26, 28, 34, 36, 38
experimentTmux(), 16, 17, 22, 73, 122, 123, 129

factorialDesign, 44
factorialDesign(), 14, 16
fdSelectLimit, 45
fdSelectLimit(), 58, 59
findProjectName (findProjectPath), 46
findProjectPath, 46

get_sim_year_heartbeat, 46
get_sim_year_heartbeat(), 123
getGithubFile, 47, 49
getModule, 48, 48
googledrive::as_id(), 61, 64
googledrive::drive_upload(), 61, 62, 64
grDevices::hcl.colors(), 68, 71

initialize, simLists-method, 49

killExperimentFuture, 32, 50, 82
killExperimentFuture(), 17
killExperimentSBATCH, 34, 36, 51
killExperimentSBATCH(), 17

lastTraceback, 51
libraryInUse (librarySnapshot), 52
librarySnapshot, 28, 36, 42, 52, 82, 133
listModules, 54
localhostLabel, 56
localhostLabel(), 33, 43

machine (user), 134
machine(), 114
makeDESCRIPTION
 (makeDESCRIPTIONproject), 56
makeDESCRIPTIONproject, 56
metadataInModules (packagesInModules), 64
metadataInModules(), 14, 55
moduleDependencies (listModules), 54

- moduleDependenciesToGraph
 (listModules), 54
- node (user), 134
- node(), 114
- openFds, 58
- openFds(), 58
- openFdsReport, 58
- openFdsReport(), 58
- outList, 59
- outList(), 22, 62, 73–76
- outSave, 60
- outSave(), 61–64, 79
- outSaveTarUpload, 61
- outSaveTarUpload(), 60, 63, 64, 75, 77, 80
- outScenarios, 62
- outScenarios(), 18, 22, 59, 73–76
- outTar, 63
- outTar(), 60–62, 64, 78, 80
- outUpload, 64
- outUpload(), 60–63, 74, 75
- packagesInModules, 64
- parallelly::makeClusterPSOCK(), 58
- pathBuild, 65
- pathBuild(), 66, 83
- pathParse, 66
- pathParse(), 65, 83
- pkgload2, 67
- plotChangeOverTime, 67
- PlotModuleGraph (listModules), 54
- plotSAs, 68
- plotSAsLeaflet (plotSAs), 68
- plotSAsLeaflet(), 71
- plotTimeSeriesLeaflet, 70
- plotTimeSeriesLeaflet(), 67, 68
- preRunSetupProject, 71
- queueRead, 73
- queueRead(), 18, 22, 66, 74, 83, 85
- queueUploadMissing, 74
- queueUploadMissing(), 18, 22, 59, 73
- reGet, 74
- reGet(), 75–77, 79, 80
- reGetMember (reIndex), 78
- reGetMember(), 78
- reGetUntarLoad, 75
- reGetUntarLoad(), 75, 78–80
- register_scenario_format, 77
- register_scenario_format(), 7, 8, 65, 83
- reIndex, 78
- reIndex(), 76, 78
- releaseLibrarySnapshot
 (librarySnapshot), 52
- reLoad, 79
- reLoad(), 75, 77, 80
- reproducible::Cache(), 76
- reproducible::preProcess(), 74
- reUntar, 80
- reUntar(), 75–77, 79
- runWorkerLoopFuture, 81
- scenario, 82
- scenario_family, 22, 83
- scenarioFields, 85
- scenarioFields(), 65, 66, 73, 83
- scenarioFieldsSet (scenarioFields), 85
- scenarioFieldsSet(), 83, 85
- setProjPkgDir, 86
- setup, 86, 88, 90, 92, 94, 95, 97, 98, 100, 102,
 107, 108, 116, 118
- setup_family, 87, 89, 91, 94, 96, 99, 101,
 105, 114, 117, 118
- setupFiles, 86
- setupFiles(), 114
- setupFunctions, 87
- setupFunctions(), 114
- setupGitIgnore, 90
- setupGitIgnore(), 114
- setupModules, 91
- setupModules(), 114
- setupOptions, 94
- setupOptions(), 114
- setupPackages, 97
- setupPackages(), 114
- setupParams, 99
- setupParams(), 114
- setupPaths, 102
- setupPaths(), 114
- setupProject, 72, 106
- setupProject(), 9–12, 16, 87, 89, 91, 94, 96,
 99, 101, 102, 105, 115, 117, 118, 124
- setupSideEffects, 115
- setupSideEffects(), 114
- setupStudyArea, 117
- setupStudyArea(), 114

- show, simLists-method, 119
- simInitAndExperiment, 119
- simLists, 16
- simLists (simLists-class), 120
- simLists-class, 120
- snapshotLibrary (librarySnapshot), 52
- SpaDES.core::loadSimList(), 76, 79
- SpaDES.core::moduleMetadata(), 14
- SpaDES.core::saveSimList(), 60–62, 79
- SpaDES.core::simInit(), 15, 16, 44, 45, 99, 120
- SpaDES.core::spades(), 14–16, 22–24, 76, 79, 120
- SpaDES.project
 - (SpaDES.project-package), 4
- SpaDES.project-package, 4
- spadesProjectOptions, 121
- statusCalculate_family, 122
- statusCalculate_FireSenseFit, 17
- statusCalculate_FireSenseFit
 - (statusCalculate_family), 122
- statusCalculate_LandR, 17
- statusCalculate_LandR
 - (statusCalculate_family), 122
- sweepLibrarySnapshots
 - (librarySnapshot), 52
- syncProjectLibrary (librarySnapshot), 52
- syncProjectLibrary(), 28, 36, 42

- teardownProject, 124
- teardownProject(), 114
- tempdir(), 78
- terra::map.pal(), 68, 71
- tmuxActiveRunningPath, 125
- tmuxActiveRunningPath(), 43
- tmuxFindDuplicates, 125
- tmuxFindDuplicates(), 22, 43
- tmuxKillPanels, 126
- tmuxKillPanels(), 17, 22, 43
- tmuxListPanels, 127
- tmuxListPanels(), 21, 22, 34, 43, 126
- tmuxMirrorQueueToSheets, 127
- tmuxMirrorQueueToSheets(), 43
- tmuxPrepareQueueFromDF, 128
- tmuxPrepareQueueFromDF(), 43
- tmuxReconcileQueueWithDF, 129
- tmuxReconcileQueueWithDF(), 42
- tmuxRefreshQueueStatus, 31, 32, 130
- tmuxRefreshQueueStatus(), 18–20, 22, 43, 122, 123
- tmuxRunNextWorker, 131
- tmuxRunNextWorker(), 43
- tmuxRunWorkerLoop, 28, 34, 81, 132
- tmuxRunWorkerLoop(), 43, 82
- tmuxSetMouse, 133
- tmuxSetMouse(), 43
- tmuxSetPaneTitle, 134
- tmuxSetPaneTitle(), 43

- user, 134
- user(), 114
- utils::getParseData(), 13
- utils::tar(), 78