Package 'parabar'

July 23, 2025

Title Progress Bar for Parallel Tasks

Version 1.4.2

Description A simple interface in the form of R6 classes for executing tasks in parallel, tracking their progress, and displaying accurate progress bars.

License MIT + file LICENSE

URL https://parabar.mihaiconstantin.com

BugReports https://github.com/mihaiconstantin/parabar/issues

Imports R6, progress, parallel, callr, filelock, utils

Encoding UTF-8

RoxygenNote 7.3.1

Collate 'TaskState.R' 'Options.R' 'Helper.R' 'Exception.R' 'Specification.R' 'BackendService.R' 'Backend.R' 'AsyncBackend.R' 'SyncBackend.R' 'BackendFactory.R' 'Bar.R' 'ModernBar.R' 'BasicBar.R' 'BarFactory.R' 'Context.R' 'ProgressTrackingContext.R' 'ContextFactory.R' 'SessionState.R' 'Warning.R' 'UserApiConsumer.R' 'exports.R' 'logo.R' 'parabar-package.R'

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation no

Author Mihai Constantin [aut, cre] (ORCID: <https://orcid.org/0000-0002-6460-0107>)

Maintainer Mihai Constantin <mihai@mihaiconstantin.com>

Repository CRAN

Date/Publication 2024-12-17 08:50:02 UTC

Contents

`AsyncBackend` *AsyncBackend*

Description

This is a concrete implementation of the abstract class `Backend` that implements the `BackendService` interface. This backend executes tasks in parallel asynchronously (i.e., without blocking the main `R` session) on a `parallel::makeCluster()` cluster created in a background `R` `session`.

Super classes

parabar::BackendService -> parabar::Backend -> AsyncBackend

Active bindings

task_state A list of logical values indicating the state of the task execution. See the TaskState class for more information on how the statues are determined. The following statuses are available:

- task_not_started: Indicates whether the backend is free. TRUE signifies that no task has been started and the backend is free to deploy.
- task_is_running: Indicates whether a task is currently running on the backend.
- task_is_completed: Indicates whether a task has finished executing. TRUE signifies that the output of the task has not been fetched. Calling the method get_option() will move the output from the background R session to the main R session. Once the output has been fetched, the backend is free to deploy another task.

session_state A list of logical values indicating the state of the background session managing the cluster. See the SessionState class for more information on the available statuses. The following statuses are available:

- session_is_starting: Indicates whether the session is starting.
- session_is_idle: Indicates whether the session is idle.
- session_is_busy: Indicates whether the session is busy. A session is busy when a task is running or when the output of a task has not been fetched into the main R session. See the task_state field.
- session_is_finished: Indicates whether the session was closed.

Methods

Public methods:

- AsyncBackend$new()
- AsyncBackend$start()
- AsyncBackend$stop()
- AsyncBackend$clear()
- AsyncBackend$peek()
- AsyncBackend$export()
- AsyncBackend$evaluate()
- AsyncBackend$sapply()
- AsyncBackend$lapply()
- AsyncBackend$apply()
- AsyncBackend$get_output()
- AsyncBackend$clone()

Method new(): Create a new AsyncBackend object.

Usage:

```
AsyncBackend$new()
```

Returns: An object of class `AsyncBackend`.

Method `start()`**:** Start the backend.

Usage:

```
AsyncBackend$start(specification)
```

Arguments:

`specification` An object of class `Specification` that contains the backend configuration.

Returns: This method returns void. The resulting backend must be stored in the `.cluster` private field on the `Backend` abstract class, and accessible to any concrete backend implementations via the active binding `cluster`.

Method `stop()`**:** Stop the backend.

Usage:

```
AsyncBackend$stop()
```

Returns: This method returns void.

Method `clear()`**:** Remove all objects from the backend. This function is equivalent to calling `rm(list = ls(all.names = TRUE))` on each node in the backend.

Usage:

```
AsyncBackend$clear()
```

Returns: This method returns void.

Method `peek()`**:** Inspect the backend for variables available in the `.GlobalEnv`.

Usage:

```
AsyncBackend$peek()
```

Returns: This method returns a list of character vectors, where each element corresponds to a node in the backend. The character vectors contain the names of the variables available in the `.GlobalEnv` on each node.

Method `export()`**:** Export variables from a given environment to the backend.

Usage:

```
AsyncBackend$export(variables, environment)
```

Arguments:

`variables` A character vector of variable names to export.

`environment` An environment object from which to export the variables.

Returns: This method returns void.

Method `evaluate()`**:** Evaluate an arbitrary expression on the backend.

Usage:

```
AsyncBackend$evaluate(expression)
```

Arguments:

`expression` An unquoted expression to evaluate on the backend.

Returns: This method returns the result of the expression evaluation.

Method `sapply()`**:** Run a task on the backend akin to `parallel::parSapply()`.

Usage:

```
AsyncBackend$sapply(x, fun, ...)
```

Arguments:

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `lapply()`**:** Run a task on the backend akin to `parallel::parLapply()`.

Usage:

```
AsyncBackend$lapply(x, fun, ...)
```

Arguments:

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `apply()`**:** Run a task on the backend akin to `parallel::parApply()`.

Usage:

```
AsyncBackend$apply(x, margin, fun, ...)
```

Arguments:

`x` An array to pass to the `fun` function.

`margin` A numeric vector indicating the dimensions of `x` the `fun` function should be applied over. For example, for a matrix, `margin = 1` indicates applying `fun` rows-wise, `margin = 2` indicates applying `fun` columns-wise, and `margin = c(1, 2)` indicates applying `fun` element-wise. Named dimensions are also possible depending on `x`. See `parallel::parApply()` and `base::apply()` for more details.

`fun` A function to apply to `x` according to the `margin`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `get_output()`**:** Get the output of the task execution.

Usage:

```
AsyncBackend$get_output(wait = FALSE)
```

Arguments:

`wait` A logical value indicating whether to wait for the task to finish executing before fetching the results. Defaults to `FALSE`. See the **Details** section for more information.

Details: This method fetches the output of the task execution after calling the `sapply()` method. It returns the output and immediately removes it from the backend. Subsequent calls to this method will throw an error if no additional tasks have been executed in the meantime. This method should be called after the execution of a task.
If `wait = TRUE`, the method will block the main process until the backend finishes executing the task and the results are available. If `wait = FALSE`, the method will immediately attempt to fetch the results from the background `R` session, and throw an error if the task is still running.

Returns: A vector, matrix, or list of the same length as `x`, containing the results of the `fun`. The output format differs based on the specific operation employed. Check out the documentation for the `apply` operations of `parallel::parallel` for more information.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
AsyncBackend$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`BackendService`, `Backend`, `SyncBackend`, `ProgressTrackingContext`, and `TaskState`.

Examples

```
# Create a specification object.
specification <- Specification$new()

# Set the number of cores.
specification$set_cores(cores = 2)

# Set the cluster type.
specification$set_type(type = "psock")

# Create an asynchronous backend object.
backend <- AsyncBackend$new()

# Start the cluster on the backend.
backend$start(specification)

# Check if there is anything on the backend.
backend$peek()

# Create a dummy variable.
name <- "parabar"

# Export the variable to the backend.
backend$export("name")
```

```
# Remove variable from current environment.
rm(name)

# Run an expression on the backend, using the exported variable `name`.
backend$evaluate({
    # Print the name.
    print(paste0("Hello, ", name, "!"))
})

# Run a task in parallel (i.e., approx. 2.5 seconds).
backend$sapply(
    x = 1:10,
    fun = function(x) {
        # Sleep a bit.
        Sys.sleep(0.5)

        # Compute something.
        output <- x + 1

        # Return the result.
        return(output)
    }
)

# Right know the main process is free and the task is executing on a `psock`
# cluster started in a background `R` session.

# Trying to get the output immediately will throw an error, indicating that the
# task is still running.
try(backend$get_output())

# However, we can block the main process and wait for the task to complete
# before fetching the results.
backend$get_output(wait = TRUE)

# Clear the backend.
backend$clear()

# Check that there is nothing on the cluster.
backend$peek()

# Stop the backend.
backend$stop()

# Check that the backend is not active.
backend$active
```

Description

This is an abstract class that serves as a base class for all concrete backend implementations. It defines the common properties that all concrete backends require.

Details

This class cannot be instantiated. It needs to be extended by concrete subclasses that implement the pure virtual methods. Instances of concrete backend implementations can be conveniently obtained using the `BackendFactory` class.

Super class

`parabar::BackendService` -> `Backend`

Active bindings

`cluster` The cluster object used by the backend. For `SyncBackend` objects, this is a cluster object created by `parallel::makeCluster()`. For `AsyncBackend` objects, this is a permanent `R` session created by `callr::r_session` that contains the `parallel::makeCluster()` cluster object.

`supports_progress` A boolean value indicating whether the backend implementation supports progress tracking.

`active` A boolean value indicating whether the backend implementation has an active cluster.

Methods

Public methods:

- `Backend$new()`
- `Backend$clone()`

Method `new()`**:** Create a new `Backend` object.

Usage:

```
Backend$new()
```

Returns: Instantiating this class will throw an error.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Backend$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`BackendService`, `SyncBackend`, `AsyncBackend`, `BackendFactory`, and `Context`.

BackendFactory *BackendFactory*

Description

This class is a factory that provides concrete implementations of the Backend abstract class.

Methods

Public methods:

- BackendFactory$get()
- BackendFactory$clone()

Method `get()`**:** Obtain a concrete implementation of the abstract Backend class of the specified type.

Usage:

```
BackendFactory$get(type)
```

Arguments:

`type` A character string specifying the type of the Backend to instantiate. Possible values are `"sync"` and `"async"`. See the **Details** section for more information.

Details: When `type = "sync"` a SyncBackend instance is created and returned. When `type = "async"` an AsyncBackend instance is provided instead.

Returns: A concrete implementation of the class Backend. It throws an error if the requested backend `type` is not supported.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BackendFactory$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

BackendService, Backend, SyncBackend, AsyncBackend, and ContextFactory.

Examples

```
# Create a backend factory.
backend_factory <- BackendFactory$new()

# Get a synchronous backend instance.
backend <- backend_factory$get("sync")

# Check the class of the backend instance.
class(backend)
```

```
# Get an asynchronous backend instance.
backend <- backend_factory$get("async")

# Check the class of the backend instance.
class(backend)
```

BackendService *BackendService*

Description

This is an interface that defines the operations available on a Backend implementation. Backend implementations and the Context class must implement this interface.

Methods

Public methods:

- BackendService$new()
- BackendService$start()
- BackendService$stop()
- BackendService$clear()
- BackendService$peek()
- BackendService$export()
- BackendService$evaluate()
- BackendService$sapply()
- BackendService$lapply()
- BackendService$apply()
- BackendService$get_output()
- BackendService$clone()

Method `new()`**:** Create a new BackendService object.

Usage:

```
BackendService$new()
```

Returns: Instantiating this class will throw an error.

Method `start()`**:** Start the backend.

Usage:

```
BackendService$start(specification)
```

Arguments:

`specification` An object of class Specification that contains the backend configuration.

Returns: This method returns void. The resulting backend must be stored in the `.cluster` private field on the `Backend` abstract class, and accessible to any concrete backend implementations via the active binding `cluster`.

Method `stop()`**:** Stop the backend.

Usage:

```
BackendService$stop()
```

Returns: This method returns void.

Method `clear()`**:** Remove all objects from the backend. This function is equivalent to calling `rm(list = ls(all.names = TRUE))` on each node in the backend.

Usage:

```
BackendService$clear()
```

Details: This method is ran by default when the backend is started.

Returns: This method returns void.

Method `peek()`**:** Inspect the backend for variables available in the `.GlobalEnv`.

Usage:

```
BackendService$peek()
```

Returns: This method returns a list of character vectors, where each element corresponds to a node in the backend. The character vectors contain the names of the variables available in the `.GlobalEnv` on each node.

Method `export()`**:** Export variables from a given environment to the backend.

Usage:

```
BackendService$export(variables, environment)
```

Arguments:

`variables` A character vector of variable names to export.

`environment` An environment object from which to export the variables.

Returns: This method returns void.

Method `evaluate()`**:** Evaluate an arbitrary expression on the backend.

Usage:

```
BackendService$evaluate(expression)
```

Arguments:

`expression` An unquoted expression to evaluate on the backend.

Returns: This method returns the result of the expression evaluation.

Method `sapply()`**:** Run a task on the backend akin to `parallel::parSapply()`.

Usage:

```
BackendService$sapply(x, fun, ...)
```

Arguments:

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `lapply()`**:** Run a task on the backend akin to `parallel::parLapply()`.

Usage:

```
BackendService$lapply(x, fun, ...)
```

Arguments:

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `apply()`**:** Run a task on the backend akin to `parallel::parApply()`.

Usage:

```
BackendService$apply(x, margin, fun, ...)
```

Arguments:

`x` An array to pass to the `fun` function.

`margin` A numeric vector indicating the dimensions of `x` the `fun` function should be applied over. For example, for a matrix, `margin = 1` indicates applying `fun` rows-wise, `margin = 2` indicates applying `fun` columns-wise, and `margin = c(1, 2)` indicates applying `fun` element-wise. Named dimensions are also possible depending on `x`. See `parallel::parApply()` and `base::apply()` for more details.

`fun` A function to apply to `x` according to the `margin`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `get_output()`**:** Get the output of the task execution.

Usage:

```
BackendService$get_output(...)
```

Arguments:

`...` Additional optional arguments that may be used by concrete implementations.

Details: This method fetches the output of the task execution after calling the `sapply()` method. It returns the output and immediately removes it from the backend. Therefore, subsequent calls to this method are not advised. This method should be called after the execution of a task.

Returns: A vector, matrix, or list of the same length as `x`, containing the results of the `fun`. The output format differs based on the specific operation employed. Check out the documentation for the `apply` operations of `parallel::parallel` for more information.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BackendService$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`Backend`, `SyncBackend`, `AsyncBackend`, and `Context`.

`Bar` *Bar*

Description

This is an abstract class that defines the pure virtual methods a concrete bar must implement.

Details

This class cannot be instantiated. It needs to be extended by concrete subclasses that implement the pure virtual methods. Instances of concrete backend implementations can be conveniently obtained using the `BarFactory` class.

Active bindings

`engine` The bar engine.

Methods

Public methods:

- `Bar$new()`
- `Bar$create()`
- `Bar$update()`
- `Bar$terminate()`
- `Bar$clone()`

Method `new()`**:** Create a new `Bar` object.

Usage:

```
Bar$new()
```

Returns: Instantiating this class will throw an error.

Method `create()`**:** Create a progress bar.

Usage:

```
Bar$create(total, initial, ...)
```

Arguments:

`total` The total number of times the progress bar should tick.

`initial` The starting point of the progress bar.

`...` Additional arguments for the bar creation. See the **Details** section for more information.

Details: The optional `...` named arguments depend on the specific concrete implementation (i.e., `BasicBar` or `ModernBar`).

Returns: This method returns void. The resulting bar is stored in the private field `.bar`, accessible via the active binding `engine`.

Method `update()`**:** Update the progress bar.

Usage:

```
Bar$update(current)
```

Arguments:

`current` The position the progress bar should be at (e.g., 30 out of 100), usually the index in a loop.

Method `terminate()`**:** Terminate the progress bar.

Usage:

```
Bar$terminate()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Bar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`BasicBar`, `ModernBar`, and `BarFactory`.

`BarFactory` *BackendFactory*

Description

This class is a factory that provides concrete implementations of the `Bar` abstract class.

Methods

Public methods:

- `BarFactory$get()`
- `BarFactory$clone()`

Method `get()`**:** Obtain a concrete implementation of the abstract `Bar` class of the specified type.

Usage:

```
BarFactory$get(type)
```

Arguments:

`type` A character string specifying the type of the `Bar` to instantiate. Possible values are `"modern"` and `"basic"`. See the **Details** section for more information.

Details: When `type = "modern"` a `ModernBar` instance is created and returned. When `type = "basic"` a `BasicBar` instance is provided instead.

Returns: A concrete implementation of the class `Bar`. It throws an error if the requested bar `type` is not supported.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BarFactory$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`Bar`, `BasicBar`, and `ModernBar`.

Examples

```
# Create a bar factory.
bar_factory <- BarFactory$new()

# Get a modern bar instance.
bar <- bar_factory$get("modern")

# Check the class of the bar instance.
class(bar)

# Get a basic bar instance.
bar <- bar_factory$get("basic")

# Check the class of the bar instance.
class(bar)
```

BasicBar *BasicBar*

Description

This is a concrete implementation of the abstract class Bar using the utils::txtProgressBar() as engine for the progress bar.

Super class

parabar::Bar -> BasicBar

Methods

Public methods:

- BasicBar$new()
- BasicBar$create()
- BasicBar$update()
- BasicBar$terminate()
- BasicBar$clone()

Method new(): Create a new BasicBar object.

Usage:

```
BasicBar$new()
```

Returns: An object of class BasicBar.

Method create(): Create a progress bar.

Usage:

```
BasicBar$create(total, initial, ...)
```

Arguments:

total The total number of times the progress bar should tick.

initial The starting point of the progress bar.

... Additional arguments for the bar creation passed to utils::txtProgressBar().

Returns: This method returns void. The resulting bar is stored in the private field .bar, accessible via the active binding engine. Both the private field and the active binding are defined in the super class Bar.

Method update(): Update the progress bar by calling utils::setTxtProgressBar().

Usage:

```
BasicBar$update(current)
```

Arguments:

current The position the progress bar should be at (e.g., 30 out of 100), usually the index in a loop.

Method `terminate()`**:** Terminate the progress bar by calling `base::close()` on the private field `.bar`.

Usage:

```
BasicBar$terminate()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BasicBar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`Bar`, `ModernBar`, and `BarFactory`.

Examples

```
# Create a basic bar instance.
bar <- BasicBar$new()

# Specify the number of ticks to be performed.
total <- 100

# Create the progress bar.
bar$create(total = total, initial = 0)

# Use the progress bar.
for (i in 1:total) {
    # Sleep a bit.
    Sys.sleep(0.02)

    # Update the progress bar.
    bar$update(i)
}

# Terminate the progress bar.
bar$terminate()
```

---

clear *Clear a Backend*

Description

This function can be used to clear a `backend` created by `start_backend()`.

Usage

```
clear(backend)
```

Arguments

backend	An object of class Backend as returned by the start_backend() function.

Details

This function is a convenience wrapper around the lower-lever API of parabar aimed at developers. More specifically, this function calls the clear method on the provided backend instance.

Value

The function returns void. It throws an error if the value provided for the backend argument is not an instance of class Backend.

See Also

start_backend(), peek(), export(), evaluate(), configure_bar(), par_sapply(), par_lapply(), par_apply(), stop_backend(), and BackendService.

Examples

```
# Create an asynchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "async")

# Check that the backend is active.
backend$active

# Check if there is anything on the backend.
peek(backend)

# Create a dummy variable.
name <- "parabar"

# Export the `name` variable in the current environment to the backend.
export(backend, "name", environment())

# Remove the dummy variable from the current environment.
rm(name)

# Check the backend to see that the variable has been exported.
peek(backend)

# Run an expression on the backend.
# Note that the symbols in the expression are resolved on the backend.
evaluate(backend, {
    # Print the name.
    print(paste0("Hello, ", name, "!"))
})

# Clear the backend.
clear(backend)
```

```
# Check that there is nothing on the backend.
peek(backend)

# Use a basic progress bar (i.e., see `parabar::Bar`).
configure_bar(type = "basic", style = 3)

# Run a task in parallel (i.e., approx. 1.25 seconds).
output <- par_sapply(backend, x = 1:10, fun = function(x) {
    # Sleep a bit.
    Sys.sleep(0.25)

    # Compute and return.
    return(x + 1)
})

# Print the output.
print(output)

# Stop the backend.
stop_backend(backend)

# Check that the backend is not active.
backend$active
```

`configure_bar` *Configure The Progress Bar*

Description

This function can be used to conveniently configure the progress bar by adjusting the `progress_bar_config` field of the `Options` instance in the `base::.Options` list.

Usage

```
configure_bar(type = "modern", ...)
```

Arguments

`type`	A character string specifying the type of progress bar to be used with compatible `backends`. Possible values are `"modern"` and `"basic"`. The default value is `"modern"`.
`...`	A list of named arguments used to configure the progress bar. See the **Details** section for more information.

Details

The optional `...` named arguments depend on the `type` of progress bar being configured. When `type = "modern"`, the `...` take the named arguments of the `progress::progress_bar` class. When `type = "basic"`, the `...` take the named arguments of the `utils::txtProgressBar()` built-in function. See the **Examples** section for a demonstration.

Value

The function returns void. It throws an error if the requested bar `type` is not supported.

See Also

`progress::progress_bar`, `utils::txtProgressBar()`, `set_default_options()`, `get_option()`, `set_option()`

Examples

```
# Set the default package options.
set_default_options()

# Get the progress bar type from options.
get_option("progress_bar_type")

# Get the progress bar configuration from options.
get_option("progress_bar_config")

# Adjust the format of the `modern` progress bar.
configure_bar(type = "modern", format = "[:bar] :percent")

# Check that the configuration has been updated in the options.
get_option("progress_bar_config")

# Change to and adjust the style of the `basic` progress bar.
configure_bar(type = "basic", style = 3)

# Check that the configuration has been updated in the options.
get_option("progress_bar_type")
get_option("progress_bar_config")
```

`Context` *Context*

Description

This class represents the base context for interacting with `Backend` implementations via the `BackendService` interface.

Details

This class is a vanilla wrapper around a `Backend` implementation. It registers a backend instance and forwards all `BackendService` methods calls to the backend instance. Subclasses can override any of the `BackendService` methods to decorate the backend instance with additional functionality (e.g., see the `ProgressTrackingContext` class for an example).

Super class

parabar::BackendService -> Context

Active bindings

backend The Backend object registered with the context.

Methods

Public methods:

- Context$new()
- Context$set_backend()
- Context$start()
- Context$stop()
- Context$clear()
- Context$peek()
- Context$export()
- Context$evaluate()
- Context$sapply()
- Context$lapply()
- Context$apply()
- Context$get_output()
- Context$clone()

Method new(): Create a new Context object.

Usage:

```
Context$new()
```

Returns: An object of class Context.

Method set_backend(): Set the backend instance to be used by the context.

Usage:

```
Context$set_backend(backend)
```

Arguments:

backend An object of class Backend that implements the BackendService interface.

Method start(): Start the backend.

Usage:

```
Context$start(specification)
```

Arguments:

specification An object of class Specification that contains the backend configuration.

Returns: This method returns void. The resulting backend must be stored in the .cluster private field on the Backend abstract class, and accessible to any concrete backend implementations via the active binding cluster.

Method `stop()`**:** Stop the backend.

Usage:

```
Context$stop()
```

Returns: This method returns void.

Method `clear()`**:** Remove all objects from the backend. This function is equivalent to calling `rm(list = ls(all.names = TRUE))` on each node in the backend.

Usage:

```
Context$clear()
```

Returns: This method returns void.

Method `peek()`**:** Inspect the backend for variables available in the `.GlobalEnv`.

Usage:

```
Context$peek()
```

Returns: This method returns a list of character vectors, where each element corresponds to a node in the backend. The character vectors contain the names of the variables available in the `.GlobalEnv` on each node.

Method `export()`**:** Export variables from a given environment to the backend.

Usage:

```
Context$export(variables, environment)
```

Arguments:

`variables` A character vector of variable names to export.

`environment` An environment object from which to export the variables. Defaults to the parent frame.

Returns: This method returns void.

Method `evaluate()`**:** Evaluate an arbitrary expression on the backend.

Usage:

```
Context$evaluate(expression)
```

Arguments:

`expression` An unquoted expression to evaluate on the backend.

Returns: This method returns the result of the expression evaluation.

Method `sapply()`**:** Run a task on the backend akin to `parallel::parSapply()`.

Usage:

```
Context$sapply(x, fun, ...)
```

Arguments:

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `lapply()`**:** Run a task on the backend akin to `parallel::parLapply()`.

Usage:

```
Context$lapply(x, fun, ...)
```

Arguments:

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `apply()`**:** Run a task on the backend akin to `parallel::parApply()`.

Usage:

```
Context$apply(x, margin, fun, ...)
```

Arguments:

`x` An array to pass to the `fun` function.

`margin` A numeric vector indicating the dimensions of `x` the `fun` function should be applied over. For example, for a matrix, `margin = 1` indicates applying `fun` rows-wise, `margin = 2` indicates applying `fun` columns-wise, and `margin = c(1, 2)` indicates applying `fun` element-wise. Named dimensions are also possible depending on `x`. See `parallel::parApply()` and `base::apply()` for more details.

`fun` A function to apply to `x` according to the `margin`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `get_output()`**:** Get the output of the task execution.

Usage:

```
Context$get_output(...)
```

Arguments:

`...` Additional arguments to pass to the backend registered with the context. This is useful for backends that require additional arguments to fetch the output (e.g., `AsyncBackend$get_output(wait = TRUE)`).

Details: This method fetches the output of the task execution after calling the `sapply()` method. It returns the output and immediately removes it from the backend. Therefore, subsequent calls to this method are not advised. This method should be called after the execution of a task.

Returns: A vector, matrix, or list of the same length as `x`, containing the results of the `fun`. The output format differs based on the specific operation employed. Check out the documentation for the `apply` operations of `parallel::parallel` for more information.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Context$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

ProgressTrackingContext, BackendService, Backend, and SyncBackend.

Examples

```
# Define a task to run in parallel.
task <- function(x, y) {
    # Sleep a bit.
    Sys.sleep(0.25)

    # Return the result of a computation.
    return(x + y)
}

# Create a specification object.
specification <- Specification$new()

# Set the number of cores.
specification$set_cores(cores = 2)

# Set the cluster type.
specification$set_type(type = "psock")

# Create a backend factory.
backend_factory <- BackendFactory$new()

# Get a synchronous backend instance.
backend <- backend_factory$get("sync")

# Create a base context object.
context <- Context$new()

# Register the backend with the context.
context$set_backend(backend)

# From now all, all backend operations are intercepted by the context.

# Start the backend.
context$start(specification)

# Run a task in parallel (i.e., approx. 1.25 seconds).
context$sapply(x = 1:10, fun = task, y = 10)

# Get the task output.
```

```
context$get_output()

# Close the backend.
context$stop()

# Get an asynchronous backend instance.
backend <- backend_factory$get("async")

# Register the backend with the same context object.
context$set_backend(backend)

# Start the backend reusing the specification object.
context$start(specification)

# Run a task in parallel (i.e., approx. 1.25 seconds).
context$sapply(x = 1:10, fun = task, y = 10)

# Get the task output.
backend$get_output(wait = TRUE)

# Close the backend.
context$stop()
```

`ContextFactory` *ContextFactory*

Description

This class is a factory that provides instances of the `Context` class.

Methods

Public methods:

- `ContextFactory$get()`
- `ContextFactory$clone()`

Method `get()`**:** Obtain instances of the `Context` class.

Usage:

```
ContextFactory$get(type)
```

Arguments:

`type` A character string specifying the type of the `Context` to instantiate. Possible values are `"regular"` and `"progress"`. See the **Details** section for more information.

Details: When `type = "regular"` a `Context` instance is created and returned. When `type = "progress"` a `ProgressTrackingContext` instance is provided instead.

Returns: An object of type `Context`. It throws an error if the requested context `type` is not supported.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ContextFactory$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Context, ProgressTrackingContext, BackendService, and Backend

Examples

```
# Create a context factory.
context_factory <- ContextFactory$new()

# Get a regular context instance.
context <- context_factory$get("regular")

# Check the class of the context instance.
class(context)

# Get a progress context instance.
context <- context_factory$get("progress")
class(context)
```

`evaluate` *Evaluate An Expression On The Backend*

Description

This function can be used to evaluate an arbitrary `base::expression()` a backend created by `start_backend()`.

Usage

```
evaluate(backend, expression)
```

Arguments

backend	An object of class Backend as returned by the `start_backend()` function.
expression	An unquoted expression to evaluate on the backend.

Details

This function is a convenience wrapper around the lower-lever API of parabar aimed at developers. More specifically, this function calls the `evaluate` method on the provided backend instance.

Value

This method returns the result of the expression evaluation. It throws an error if the value provided for the backend argument is not an instance of class Backend.

See Also

start_backend(), peek(), export(), clear(), configure_bar(), par_sapply(), par_lapply(), par_apply(), stop_backend(), and BackendService.

Examples

```
# Create an asynchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "async")

# Check that the backend is active.
backend$active

# Check if there is anything on the backend.
peek(backend)

# Create a dummy variable.
name <- "parabar"

# Export the `name` variable in the current environment to the backend.
export(backend, "name", environment())

# Remove the dummy variable from the current environment.
rm(name)

# Check the backend to see that the variable has been exported.
peek(backend)

# Run an expression on the backend.
# Note that the symbols in the expression are resolved on the backend.
evaluate(backend, {
    # Print the name.
    print(paste0("Hello, ", name, "!"))
})

# Clear the backend.
clear(backend)

# Check that there is nothing on the backend.
peek(backend)

# Use a basic progress bar (i.e., see `parabar::Bar`).
configure_bar(type = "basic", style = 3)

# Run a task in parallel (i.e., approx. 1.25 seconds).
output <- par_sapply(backend, x = 1:10, fun = function(x) {
    # Sleep a bit.
    Sys.sleep(0.25)
```

```
    # Compute and return.
    return(x + 1)
})

# Print the output.
print(output)

# Stop the backend.
stop_backend(backend)

# Check that the backend is not active.
backend$active
```

`Exception` *Package Exceptions*

Description

This class contains static methods for throwing exceptions with informative messages.

Format

`Exception$abstract_class_not_instantiable(object)` Exception for instantiating abstract classes or interfaces.

`Exception$method_not_implemented()` Exception for calling methods without an implementation.

`Exception$feature_not_developed()` Exception for running into things not yet developed.

`Exception$not_enough_cores()` Exception for requesting more cores than available on the machine.

`Exception$cluster_active()` Exception for attempting to start a cluster while another one is active.

`Exception$cluster_not_active()` Exception for attempting to stop a cluster while not active.

`Exception$async_task_not_started()` Exception for reading results while an asynchronous task has not yet started.

`Exception$async_task_running()` Exception for reading results while an asynchronous task is running.

`Exception$async_task_completed()` Exception for reading results while a completed asynchronous task has unread results.

`Exception$async_task_error(error)` Exception for errors while running an asynchronous task.

`Exception$stop_busy_backend_not_allowed()` Exception for stopping a busy backend without intent.

`Exception$temporary_file_creation_failed()` Exception for reading results while an asynchronous task is running.

`Exception$type_not_assignable(actual, expected)` Exception for when providing incorrect object types.

`Exception$unknown_package_option(option)` Exception for when requesting unknown package options.

`Exception$primitive_as_task_not_allowed()` Exception for when decorating primitive functions with progress tracking.

`Exception$array_margins_not_compatible(actual, allowed)` Exception for using improper margins in the `BackendService$apply` operation.

`export` *Export Objects To a Backend*

Description

This function can be used to export objects to a `backend` created by `start_backend()`.

Usage

```
export(backend, variables, environment)
```

Arguments

`backend`	An object of class `Backend` as returned by the `start_backend()` function.
`variables`	A character vector of variable names to export to the backend.
`environment`	An environment from which to export the variables. If no environment is provided, the `.GlobalEnv` environment is used.

Details

This function is a convenience wrapper around the lower-lever API of `parabar` aimed at developers. More specifically, this function calls the `export` method on the provided `backend` instance.

Value

The function returns void. It throws an error if the value provided for the `backend` argument is not an instance of class `Backend`.

See Also

`start_backend()`, `peek()`, `evaluate()`, `clear()`, `configure_bar()`, `par_sapply()`, `par_lapply()`, `par_apply()`, `stop_backend()`, and `BackendService`.

Examples

```
# Create an asynchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "async")

# Check that the backend is active.
backend$active

# Check if there is anything on the backend.
peek(backend)

# Create a dummy variable.
name <- "parabar"

# Export the `name` variable in the current environment to the backend.
export(backend, "name", environment())

# Remove the dummy variable from the current environment.
rm(name)

# Check the backend to see that the variable has been exported.
peek(backend)

# Run an expression on the backend.
# Note that the symbols in the expression are resolved on the backend.
evaluate(backend, {
    # Print the name.
    print(paste0("Hello, ", name, "!"))
})

# Clear the backend.
clear(backend)

# Check that there is nothing on the backend.
peek(backend)

# Use a basic progress bar (i.e., see `parabar::Bar`).
configure_bar(type = "basic", style = 3)

# Run a task in parallel (i.e., approx. 1.25 seconds).
output <- par_sapply(backend, x = 1:10, fun = function(x) {
    # Sleep a bit.
    Sys.sleep(0.25)

    # Compute and return.
    return(x + 1)
})

# Print the output.
print(output)

# Stop the backend.
stop_backend(backend)
```

```
# Check that the backend is not active.
backend$active
```

`get_option` *Get or Set Package Option*

Description

The `get_option()` function is a helper for retrieving the value of `parabar` `options`. If the `option` requested is not available in the session `base::.Options` list, the corresponding default value set by the `Options` `R6::R6` class is returned instead.

The `set_option()` function is a helper for setting `parabar` `options`. The function adjusts the fields of the `Options` instance stored in the `base::.Options` list. If no `Options` instance is present in the `base::.Options` list, a new one is created.

The `set_default_options()` function is used to set the default `options` values for the `parabar` package. The function is automatically called at package load and the entry created can be retrieved via `getOption("parabar")`. Specific package `options` can be retrieved using the helper function `get_option()`.

Usage

```
get_option(option)

set_option(option, value)

set_default_options()
```

Arguments

`option`	A character string representing the name of the option to retrieve or adjust. See the public fields of `R6::R6` class `Options` for the list of available `parabar` `options`.
`value`	The value to set the `option` to.

Value

The `get_option()` function returns the value of the requested `option` present in the `base::.Options` list, or its corresponding default value (i.e., see `Options`). If the requested `option` is not known, an error is thrown.

The `set_option()` function returns void. It throws an error if the requested `option` to be adjusted is not known.

The `set_default_options()` function returns void. The `options` set can be consulted via the `base::.Options` list. See the `Options` `R6::R6` class for more information on the default values set by this function.

See Also

Options, set_default_options(), base::options(), and base::getOption().

Examples

```
# Get the status of progress tracking.
get_option("progress_track")

# Set the status of progress tracking to `FALSE`.
set_option("progress_track", FALSE)

# Get the status of progress tracking again.
get_option("progress_track")

# Restore default options.
set_default_options()

# Get the status of progress tracking yet again.
get_option("progress_track")
```

Helper *Package Helpers*

Description

This class contains static helper methods.

Format

`Helper$get_class_name(object)` Helper for getting the class of a given object.

`Helper$is_of_class(object, class)` Check if an object is of a certain class.

`Helper$get_option(option)` Get package option, or corresponding default value.

`Helper$set_option(option, value)` Set package option.

`Helper$check_object_type(object, expected_type)` Check the type of a given object.

`Helper$check_array_margins(margins, dimensions)` Helper to check array margins for the `BackendService$apply` operation.

LOGO *The Package Logo*

Description

The logo is generated by `make_logo()` and displayed on package attach for interactive `R` sessions.

Usage

```
LOGO
```

Format

An object of class `character` containing the `ASCII` logo.

See Also

`make_logo()`

Examples

```
print(LOGO)
```

make_logo *Generate Package Logo*

Description

This function is meant for generating or updating the logo. After running this procedure we end up with what is stored in the `LOGO` constant.

Usage

```
make_logo(
  template = "./inst/assets/logo/parabar-logo.txt",
  version = c(1, 0, 0)
)
```

Arguments

`template`	A character string representing the path to the logo template.
`version`	A numerical vector of three positive integers representing the version of the package to append to the logo.

Value

The ASCII logo.

See Also

LOGO

Examples

```
## Not run:

# Generate the logo.
logo <- make_logo()

# Print the logo.
cat(logo)

## End(Not run)
```

ModernBar *ModernBar*

Description

This is a concrete implementation of the abstract class Bar using the progress::progress_bar as engine for the progress bar.

Super class

parabar::Bar -> ModernBar

Methods

Public methods:

- ModernBar$new()
- ModernBar$create()
- ModernBar$update()
- ModernBar$terminate()
- ModernBar$clone()

Method new(): Create a new ModernBar object.

Usage:

```
ModernBar$new()
```

Returns: An object of class ModernBar.

Method create(): Create a progress bar.

Usage:

```
ModernBar$create(total, initial, ...)
```

Arguments:

`total` The total number of times the progress bar should tick.

`initial` The starting point of the progress bar.

`...` Additional arguments for the bar creation passed to `progress::progress_bar$new()`.

Returns: This method returns void. The resulting bar is stored in the private field `.bar`, accessible via the active binding `engine`. Both the private field and the active binding are defined in the super class `Bar`.

Method `update()`**:** Update the progress bar by calling `progress::progress_bar$update()`.

Usage:

```
ModernBar$update(current)
```

Arguments:

`current` The position the progress bar should be at (e.g., 30 out of 100), usually the index in a loop.

Method `terminate()`**:** Terminate the progress bar by calling `progress::progress_bar$terminate()`.

Usage:

```
ModernBar$terminate()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ModernBar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`Bar`, `BasicBar`, and `BarFactory`.

Examples

```
# Create a modern bar instance.
bar <- ModernBar$new()

# Specify the number of ticks to be performed.
total <- 100

# Create the progress bar.
bar$create(total = total, initial = 0)

# Use the progress bar.
for (i in 1:total) {
    # Sleep a bit.
    Sys.sleep(0.02)
```

```
    # Update the progress bar.
    bar$update(i)
}

# Terminate the progress bar.
bar$terminate()
```

`Options` *Class for Package Options*

Description

This class holds public fields that represent the package `options` used to configure the default behavior of the functionality `parabar` provides.

Details

An instance of this class is automatically created and stored in the session `base::.Options` at load time. This instance can be accessed and changed via `getOption("parabar")`. Specific package `options` can be retrieved using the helper function `get_option()`.

Public fields

`progress_track` A logical value indicating whether progress tracking should be enabled (i.e., `TRUE`) or disabled (i.e., `FALSE`) globally for compatible backends. The default value is `TRUE`.

`progress_timeout` A numeric value indicating the timeout (i.e., in seconds) between subsequent checks of the log file for new progress records. The default value is `0.001`.

`progress_wait` A numeric value indicating the approximate duration (i.e., in seconds) to wait between progress bar updates before checking if the task has finished (i.e., possibly with an error). The default value is `0.1`.

`progress_bar_type` A character string indicating the default bar type to use with compatible backends. Possible values are `"modern"` (the default) or `"basic"`.

`progress_bar_config` A list of lists containing the default bar configuration for each supported bar engine. Elements of these lists represent arguments for the corresponding bar engines. Currently, the supported bar engines are:

- `modern`: The `progress::progress_bar` engine, with the following default configuration:
  - `show_after = 0`
  - `format = "> completed :current out of :total tasks [:percent] [:elapsed]"`
- `basic`: The `utils::txtProgressBar` engine, with no default configuration.

`stop_forceful` A logical value indicating whether to allow stopping an asynchronous backend forcefully (i.e., `TRUE`), or not (i.e., `FALSE`). When stopping forcefully, the backend is terminated without waiting for a running tasks to finish or for the results to be read into the main `R` session. The default value is `FALSE`.

Active bindings

`progress_log_path` A character string indicating the path to the log file where to track the execution progress of a running task. The default value is a temporary file generated by `base::tempfile()`. Calling this active binding repeatedly will yield different temporary file paths. Fixing the path to a specific value is possible by setting this active binding to a character string representing the desired path. Setting this active binding to `NULL` will reset it to the default value (i.e., yielding different temporary file paths).

See Also

`get_option()`, `set_option()`, and `set_default_options()`.

Examples

```
# Set the default package options (i.e., automatically set at load time).
set_default_options()

# First, get the options instance from the session options.
parabar <- getOption("parabar")

# Then, disable progress tracking.
parabar$progress_track <- FALSE

# Check that the change was applied (i.e., `progress_track: FALSE`).
getOption("parabar")

# To restore defaults, set the default options again.
set_default_options()

# Check that the change was applied (i.e., `progress_track: TRUE`).
getOption("parabar")

# We can also use the built-in helpers to get and set options more conveniently.

# Get the progress tracking option.
get_option("progress_track")

# Set the progress tracking option to `FALSE`.
set_option("progress_track", FALSE)

# Check that the change was applied (i.e., `progress_track: FALSE`).
get_option("progress_track")

# Get a temporary file for logging the progress.
get_option("progress_log_path")

# Fix the logging file path.
set_option("progress_log_path", "./progress.log")

# Check that the logging path change was applied.
get_option("progress_log_path")
```

```
# Restore the logging path to the default behavior.
set_option("progress_log_path", NULL)

# Check that the logging path change was applied.
get_option("progress_log_path")

# Restore the defaults.
set_default_options()
```

`par_apply` *Run a Task in Parallel*

Description

This function can be used to run a task in parallel. The task is executed in parallel on the specified backend, similar to `parallel::parApply()`. If `backend = NULL`, the task is executed sequentially using `base::apply()`. See the **Details** section for more information on how this function works.

Usage

```
par_apply(backend = NULL, x, margin, fun, ...)
```

Arguments

`backend`	An object of class `Backend` as returned by the `start_backend()` function. It can also be `NULL` to run the task sequentially via `base::apply()`. The default value is `NULL`.
`x`	An array to pass to the `fun` function.
`margin`	A numeric vector indicating the dimensions of `x` the `fun` function should be applied over. For example, for a matrix, `margin = 1` indicates applying `fun` rows-wise, `margin = 2` indicates applying `fun` columns-wise, and `margin = c(1, 2)` indicates applying `fun` element-wise. Named dimensions are also possible depending on `x`. See `parallel::parApply()` and `base::apply()` for more details.
`fun`	A function to apply to `x` according to the `margin`.
`...`	Additional arguments to pass to the `fun` function.

Details

This function uses the `UserApiConsumer` class that acts like an interface for the developer API of the `parabar` package.

Value

The dimensions of the output vary according to the `margin` argument. Consult the documentation of `base::apply()` for a detailed explanation on how the output is structured.

See Also

start_backend(), peek(), export(), evaluate(), clear(), configure_bar(), par_sapply(), par_lapply(), stop_backend(), set_option(), get_option(), Options, UserApiConsumer, and BackendService.

Examples

```
# Define a simple task.
task <- function(x) {
    # Perform computations.
    Sys.sleep(0.01)

    # Return the result.
    mean(x)
}

# Define a matrix for the task.
x <- matrix(rnorm(100^2, mean = 10, sd = 0.5), nrow = 100, ncol = 100)

# Start an asynchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "async")

# Run a task in parallel over the rows of `x`.
results <- par_apply(backend, x = x, margin = 1, fun = task)

# Run a task in parallel over the columns of `x`.
results <- par_apply(backend, x = x, margin = 2, fun = task)

# The task can also be run over all elements of `x` using `margin = c(1, 2)`.
# Improper dimensions will throw an error.
try(par_apply(backend, x = x, margin = c(1, 2, 3), fun = task))

# Disable progress tracking.
set_option("progress_track", FALSE)

# Run a task in parallel.
results <- par_apply(backend, x = x, margin = 1, fun = task)

# Enable progress tracking.
set_option("progress_track", TRUE)

# Change the progress bar options.
configure_bar(type = "modern", format = "[:bar] :percent")

# Run a task in parallel.
results <- par_apply(backend, x = x, margin = 1, fun = task)

# Stop the backend.
stop_backend(backend)

# Start a synchronous backend.
```

```
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "sync")

# Run a task in parallel.
results <- par_apply(backend, x = x, margin = 1, fun = task)

# Disable progress tracking to remove the warning that progress is not supported.
set_option("progress_track", FALSE)

# Run a task in parallel.
results <- par_apply(backend, x = x, margin = 1, fun = task)

# Stop the backend.
stop_backend(backend)

# Run the task using the `base::lapply` (i.e., non-parallel).
results <- par_apply(NULL, x = x, margin = 1, fun = task)
```

`par_lapply` *Run a Task in Parallel*

Description

This function can be used to run a task in parallel. The task is executed in parallel on the specified backend, similar to `parallel::parLapply()`. If `backend = NULL`, the task is executed sequentially using `base::lapply()`. See the **Details** section for more information on how this function works.

Usage

```
par_lapply(backend = NULL, x, fun, ...)
```

Arguments

`backend`	An object of class `Backend` as returned by the `start_backend()` function. It can also be `NULL` to run the task sequentially via `base::lapply()`. The default value is `NULL`.
`x`	An atomic vector or list to pass to the `fun` function.
`fun`	A function to apply to each element of `x`.
`...`	Additional arguments to pass to the `fun` function.

Details

This function uses the `UserApiConsumer` class that acts like an interface for the developer API of the `parabar` package.

Value

A list of the same length as x containing the results of the fun. The output format resembles that of base::lapply().

See Also

start_backend(), peek(), export(), evaluate(), clear(), configure_bar(), par_sapply(), par_apply(), stop_backend(), set_option(), get_option(), Options, UserApiConsumer, and BackendService.

Examples

```
# Define a simple task.
task <- function(x) {
    # Perform computations.
    Sys.sleep(0.01)

    # Return the result.
    return(x + 1)
}

# Start an asynchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "async")

# Run a task in parallel.
results <- par_lapply(backend, x = 1:300, fun = task)

# Disable progress tracking.
set_option("progress_track", FALSE)

# Run a task in parallel.
results <- par_lapply(backend, x = 1:300, fun = task)

# Enable progress tracking.
set_option("progress_track", TRUE)

# Change the progress bar options.
configure_bar(type = "modern", format = "[:bar] :percent")

# Run a task in parallel.
results <- par_lapply(backend, x = 1:300, fun = task)

# Stop the backend.
stop_backend(backend)

# Start a synchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "sync")

# Run a task in parallel.
results <- par_lapply(backend, x = 1:300, fun = task)
```

```
# Disable progress tracking to remove the warning that progress is not supported.
set_option("progress_track", FALSE)

# Run a task in parallel.
results <- par_lapply(backend, x = 1:300, fun = task)

# Stop the backend.
stop_backend(backend)

# Run the task using the `base::lapply` (i.e., non-parallel).
results <- par_lapply(NULL, x = 1:300, fun = task)
```

`par_sapply` *Run a Task in Parallel*

Description

This function can be used to run a task in parallel. The task is executed in parallel on the specified backend, similar to `parallel::parSapply()`. If `backend = NULL`, the task is executed sequentially using `base::sapply()`. See the **Details** section for more information on how this function works.

Usage

```
par_sapply(backend = NULL, x, fun, ...)
```

Arguments

`backend`	An object of class `Backend` as returned by the `start_backend()` function. It can also be `NULL` to run the task sequentially via `base::sapply()`. The default value is `NULL`.
`x`	An atomic vector or list to pass to the `fun` function.
`fun`	A function to apply to each element of `x`.
`...`	Additional arguments to pass to the `fun` function.

Details

This function uses the `UserApiConsumer` class that acts like an interface for the developer API of the `parabar` package.

Value

A vector of the same length as `x` containing the results of the `fun`. The output format resembles that of `base::sapply()`.

See Also

start_backend(), peek(), export(), evaluate(), clear(), configure_bar(), par_lapply(), par_apply(), stop_backend(), set_option(), get_option(), Options, UserApiConsumer, and BackendService.

Examples

```
# Define a simple task.
task <- function(x) {
    # Perform computations.
    Sys.sleep(0.01)

    # Return the result.
    return(x + 1)
}

# Start an asynchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "async")

# Run a task in parallel.
results <- par_sapply(backend, x = 1:300, fun = task)

# Disable progress tracking.
set_option("progress_track", FALSE)

# Run a task in parallel.
results <- par_sapply(backend, x = 1:300, fun = task)

# Enable progress tracking.
set_option("progress_track", TRUE)

# Change the progress bar options.
configure_bar(type = "modern", format = "[:bar] :percent")

# Run a task in parallel.
results <- par_sapply(backend, x = 1:300, fun = task)

# Stop the backend.
stop_backend(backend)

# Start a synchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "sync")

# Run a task in parallel.
results <- par_sapply(backend, x = 1:300, fun = task)

# Disable progress tracking to remove the warning that progress is not supported.
set_option("progress_track", FALSE)

# Run a task in parallel.
results <- par_sapply(backend, x = 1:300, fun = task)
```

```
# Stop the backend.
stop_backend(backend)

# Run the task using the `base::sapply` (i.e., non-parallel).
results <- par_sapply(NULL, x = 1:300, fun = task)
```

---

peek *Inspect a Backend*

Description

This function can be used to check the names of the variables present on a backend created by start_backend().

Usage

```
peek(backend)
```

Arguments

backend	An object of class Backend as returned by the start_backend() function.

Details

This function is a convenience wrapper around the lower-lever API of parabar aimed at developers. More specifically, this function calls the peek method on the provided backend instance.

Value

The function returns a list of character vectors, where each list element corresponds to a node, and each element of the character vector is the name of a variable present on that node. It throws an error if the value provided for the backend argument is not an instance of class Backend.

See Also

start_backend(), export(), evaluate(), clear(), configure_bar(), par_sapply(), par_lapply(), par_apply(), stop_backend(), and BackendService.

Examples

```
# Create an asynchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "async")

# Check that the backend is active.
backend$active

# Check if there is anything on the backend.
peek(backend)

# Create a dummy variable.
name <- "parabar"

# Export the `name` variable in the current environment to the backend.
export(backend, "name", environment())

# Remove the dummy variable from the current environment.
rm(name)

# Check the backend to see that the variable has been exported.
peek(backend)

# Run an expression on the backend.
# Note that the symbols in the expression are resolved on the backend.
evaluate(backend, {
    # Print the name.
    print(paste0("Hello, ", name, "!"))
})

# Clear the backend.
clear(backend)

# Check that there is nothing on the backend.
peek(backend)

# Use a basic progress bar (i.e., see `parabar::Bar`).
configure_bar(type = "basic", style = 3)

# Run a task in parallel (i.e., approx. 1.25 seconds).
output <- par_sapply(backend, x = 1:10, fun = function(x) {
    # Sleep a bit.
    Sys.sleep(0.25)

    # Compute and return.
    return(x + 1)
})

# Print the output.
print(output)

# Stop the backend.
stop_backend(backend)
```

```
# Check that the backend is not active.
backend$active
```

ProgressTrackingContext

ProgressTrackingContext

Description

This class represents a progress tracking context for interacting with Backend implementations via the BackendService interface.

Details

This class extends the base Context class and overrides the sapply parent method to decorate the backend instance with additional functionality. Specifically, this class creates a temporary file to log the progress of backend tasks, and then creates a progress bar to display the progress of the backend tasks.

The progress bar is updated after each backend task execution. The timeout between subsequent checks of the temporary log file is controlled by the Options class and defaults to `0.001`. This value can be adjusted via the Options instance present in the session base::.Options list (i.e., see set_option()). For example, to set the timeout to `0.1` we can run `set_option("progress_timeout", 0.1)`.

This class is a good example of how to extend the base Context class to decorate the backend instance with additional functionality.

Super classes

parabar::BackendService -> parabar::Context -> ProgressTrackingContext

Active bindings

`bar` The Bar instance registered with the context.

Methods

Public methods:

- ProgressTrackingContext$set_backend()
- ProgressTrackingContext$set_bar()
- ProgressTrackingContext$configure_bar()
- ProgressTrackingContext$sapply()
- ProgressTrackingContext$lapply()
- ProgressTrackingContext$apply()
- ProgressTrackingContext$clone()

Method `set_backend()`**:** Set the backend instance to be used by the context.

Usage:

```
ProgressTrackingContext$set_backend(backend)
```

Arguments:

`backend` An object of class `Backend` that supports progress tracking implements the `BackendService` interface.

Details: This method overrides the parent method to validate the backend provided and guarantee it is an instance of the `AsyncBackend` class.

Method `set_bar()`**:** Set the `Bar` instance to be used by the context.

Usage:

```
ProgressTrackingContext$set_bar(bar)
```

Arguments:

`bar` An object of class `Bar`.

Method `configure_bar()`**:** Configure the `Bar` instance registered with the context.

Usage:

```
ProgressTrackingContext$configure_bar(...)
```

Arguments:

`...` A list of named arguments passed to the `create()` method of the `Bar` instance. See the documentation of the specific concrete bar for details (e.g., `ModernBar`).

Method `sapply()`**:** Run a task on the backend akin to `parallel::parSapply()`, but with a progress bar.

Usage:

```
ProgressTrackingContext$sapply(x, fun, ...)
```

Arguments:

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `lapply()`**:** Run a task on the backend akin to `parallel::parLapply()`, but with a progress bar.

Usage:

```
ProgressTrackingContext$lapply(x, fun, ...)
```

Arguments:

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `apply()`**:** Run a task on the backend akin to `parallel::parApply()`.

Usage:

```
ProgressTrackingContext$apply(x, margin, fun, ...)
```

Arguments:

`x` An array to pass to the `fun` function.

`margin` A numeric vector indicating the dimensions of `x` the `fun` function should be applied over. For example, for a matrix, `margin = 1` indicates applying `fun` rows-wise, `margin = 2` indicates applying `fun` columns-wise, and `margin = c(1, 2)` indicates applying `fun` element-wise. Named dimensions are also possible depending on `x`. See `parallel::parApply()` and `base::apply()` for more details.

`fun` A function to apply to `x` according to the `margin`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ProgressTrackingContext$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`Context`, `BackendService`, `Backend`, and `AsyncBackend`.

Examples

```
# Define a task to run in parallel.
task <- function(x, y) {
    # Sleep a bit.
    Sys.sleep(0.15)

    # Return the result of a computation.
    return(x + y)
}

# Create a specification object.
specification <- Specification$new()

# Set the number of cores.
specification$set_cores(cores = 2)

# Set the cluster type.
```

```
specification$set_type(type = "psock")

# Create a backend factory.
backend_factory <- BackendFactory$new()

# Get a backend instance that does not support progress tracking.
backend <- backend_factory$get("sync")

# Create a progress tracking context object.
context <- ProgressTrackingContext$new()

# Attempt to set the incompatible backend instance.
try(context$set_backend(backend))

# Get a backend instance that does support progress tracking.
backend <- backend_factory$get("async")

# Register the backend with the context.
context$set_backend(backend)

# From now all, all backend operations are intercepted by the context.

# Start the backend.
context$start(specification)

# Create a bar factory.
bar_factory <- BarFactory$new()

# Get a modern bar instance.
bar <- bar_factory$get("modern")

# Register the bar with the context.
context$set_bar(bar)

# Configure the bar.
context$configure_bar(
    show_after = 0,
    format = " > completed :current out of :total tasks [:percent] [:elapsed]"
)

# Run a task in parallel (i.e., approx. 1.9 seconds).
context$sapply(x = 1:25, fun = task, y = 10)

# Get the task output.
backend$get_output(wait = TRUE)

# Change the bar type.
bar <- bar_factory$get("basic")

# Register the bar with the context.
context$set_bar(bar)

# Remove the previous bar configuration.
```

```
context$configure_bar()

# Run a task in parallel (i.e., approx. 1.9 seconds).
context$sapply(x = 1:25, fun = task, y = 10)

# Get the task output.
backend$get_output(wait = TRUE)

# Close the backend.
context$stop()
```

`SessionState` *SessionState*

Description

This class holds the state of a background `session` used by an asynchronous backend (i.e., `AsyncBackend`). See the **Details** section for more information.

Details

The session state is useful to check if an asynchronous backend is ready for certain operations. A session can only be in one of the following four states at a time:

- `session_is_starting`: When `TRUE`, it indicates that the session is starting.
- `session_is_idle`: When `TRUE`, it indicates that the session is idle and ready to execute operations.
- `session_is_busy`: When `TRUE`, it indicates that the session is busy (i.e., see the `TaskState` class for more information about a task's state).
- `session_is_finished`: When `TRUE`, it indicates that the session is closed and no longer available for operations.

Active bindings

`session_is_starting` A logical value indicating whether the session is starting.

`session_is_idle` A logical value indicating whether the session is idle and ready to execute operations.

`session_is_busy` A logical value indicating whether the session is busy.

`session_is_finished` A logical value indicating whether the session is closed and no longer available for operations.

Methods

Public methods:

- SessionState$new()
- SessionState$clone()

Method `new()`**:** Create a new SessionState object and determine the state of a given background session.

Usage:

```
SessionState$new(session)
```

Arguments:

`session` A callr::r_session object.

Returns: An object of class SessionState.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SessionState$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

TaskState, AsyncBackend and ProgressTrackingContext.

Examples

```
# Handy function to print the session states all at once.
check_state <- function(session) {
    # Create a session object and determine its state.
    session_state <- SessionState$new(session)

    # Print the state.
    cat(
        "Session is starting: ", session_state$session_is_starting, "\n",
        "Session is idle: ", session_state$session_is_idle, "\n",
        "Session is busy: ", session_state$session_is_busy, "\n",
        "Session is finished: ", session_state$session_is_finished, "\n",
        sep = ""
    )
}

# Create a specification object.
specification <- Specification$new()

# Set the number of cores.
specification$set_cores(cores = 2)

# Set the cluster type.
specification$set_type(type = "psock")
```

```
# Create an asynchronous backend object.
backend <- AsyncBackend$new()

# Start the cluster on the backend.
backend$start(specification)

# Check that the session is idle.
check_state(backend$cluster)

{
    # Run a task in parallel (i.e., approx. 0.25 seconds).
    backend$sapply(
        x = 1:10,
        fun = function(x) {
            # Sleep a bit.
            Sys.sleep(0.05)

            # Compute something.
            output <- x + 1

            # Return the result.
            return(output)
        }
    )

    # And immediately check that the session is busy.
    check_state(backend$cluster)
}

# Get the output and wait for the task to complete.
output <- backend$get_output(wait = TRUE)

# Check that the session is idle again.
check_state(backend$cluster)

# Manually close the session.
backend$cluster$close()

# Check that the session is finished.
check_state(backend$cluster)

# Stop the backend.
backend$stop()
```

Specification *Specification*

Description

This class contains the information required to start a backend. An instance of this class is used by the `start` method of the `BackendService` interface.

Active bindings

`cores` The number of nodes to use in the cluster creation.

`type` The type of cluster to create.

`types` The supported cluster types.

Methods

Public methods:

- `Specification$set_cores()`
- `Specification$set_type()`
- `Specification$clone()`

Method `set_cores()`**:** Set the number of nodes to use in the cluster.

Usage:

```
Specification$set_cores(cores)
```

Arguments:

`cores` The number of nodes to use in the cluster.

Details: This method also performs a validation of the requested number of cores, ensuring that the the value lies between 1 and `parallel::detectCores() - 1`.

Method `set_type()`**:** Set the type of cluster to create.

Usage:

```
Specification$set_type(type)
```

Arguments:

`type` The type of cluster to create. Possible values are `"fork"` and `"psock"`. Defaults to `"psock"`.

Details: If no type is explicitly requested (i.e., `type = NULL`), the type is determined based on the operating system. On Unix-like systems, the type is set to `"fork"`, while on Windows systems, the type is set to `"psock"`. If an unknown type is requested, a warning is issued and the type is set to `"psock"`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Specification$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`BackendService`, `Backend`, `SyncBackend`, and `AsyncBackend`.

Examples

```
# Create a specification object.
specification <- Specification$new()

# Set the number of cores.
specification$set_cores(cores = 4)

# Set the cluster type.
specification$set_type(type = "psock")

# Get the number of cores.
specification$cores

# Get the cluster type.
specification$type

# Attempt to set too many cores.
specification$set_cores(cores = 100)

# Check that the cores were reasonably set.
specification$cores

# Allow the object to determine the adequate cluster type.
specification$set_type(type = NULL)

# Check the type determined.
specification$type

# Attempt to set an invalid cluster type.
specification$set_type(type = "invalid")

# Check that the type was set to `psock`.
specification$type
```

`start_backend` *Start a Backend*

Description

This function can be used to start a backend. Check out the **Details** section for more information.

Usage

```
start_backend(cores, cluster_type = "psock", backend_type = "async")
```

Arguments

`cores`	A positive integer representing the number of cores to use (i.e., the number of processes to start). This value must be between 2 and `parallel::detectCores() - 1`.
`cluster_type`	A character string representing the type of cluster to create. Possible values are `"fork"` and `"psock"`. Defaults to `"psock"`. See the section **Cluster Type** for more information.
`backend_type`	A character string representing the type of backend to create. Possible values are `"sync"` and `"async"`. Defaults to `"async"`. See the section **Backend Type** for more information.

Details

This function is a convenience wrapper around the lower-lever API of `parabar` aimed at developers. More specifically, this function uses the `Specification` class to create a specification object, and the `BackendFactory` class to create a `Backend` instance based on the specification object.

Value

A `Backend` instance that can be used to parallelize computations. The methods available on the `Backend` instance are defined by the `BackendService` interface.

Cluster Type

The cluster type determines the type of cluster to create. The requested value is validated and passed to the `type` argument of the `parallel::makeCluster()` function. The following table lists the possible values and their corresponding description.

Cluster	Description
`"fork"`	For Unix-based systems.
`"psock"`	For Windows-based systems.

Backend Type

The backend type determines the type of backend to create. The requested value is passed to the `BackendFactory` class, which returns a `Backend` instance of the desired type. The following table lists the possible backend types and their corresponding description.

Backend	Description	Implementation	Progress
`"sync"`	A synchronous backend.	`SyncBackend`	no
`"async"`	An asynchronous backend.	`AsyncBackend`	yes

In a nutshell, the difference between the two backend types is that for the synchronous backend the cluster is created in the main process, while for the asynchronous backend the cluster is created in a backend `R` process using `callr::r_session`. Therefore, the synchronous backend is blocking the main process during task execution, while the asynchronous backend is non-blocking. Check out the implementations listed in the table above for more information. All concrete implementations extend the `Backend` abstract class and implement the `BackendService` interface.

See Also

peek(), export(), evaluate(), clear(), configure_bar(), par_sapply(), par_lapply(), par_apply(), stop_backend(), and BackendService.

Examples

```
# Create an asynchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "async")

# Check that the backend is active.
backend$active

# Check if there is anything on the backend.
peek(backend)

# Create a dummy variable.
name <- "parabar"

# Export the `name` variable in the current environment to the backend.
export(backend, "name", environment())

# Remove the dummy variable from the current environment.
rm(name)

# Check the backend to see that the variable has been exported.
peek(backend)

# Run an expression on the backend.
# Note that the symbols in the expression are resolved on the backend.
evaluate(backend, {
    # Print the name.
    print(paste0("Hello, ", name, "!"))
})

# Clear the backend.
clear(backend)

# Check that there is nothing on the backend.
peek(backend)

# Use a basic progress bar (i.e., see `parabar::Bar`).
configure_bar(type = "basic", style = 3)

# Run a task in parallel (i.e., approx. 1.25 seconds).
output <- par_sapply(backend, x = 1:10, fun = function(x) {
    # Sleep a bit.
    Sys.sleep(0.25)

    # Compute and return.
    return(x + 1)
})
```

```
# Print the output.
print(output)

# Stop the backend.
stop_backend(backend)

# Check that the backend is not active.
backend$active
```

---

stop_backend *Stop a Backend*

Description

This function can be used to stop a backend created by start_backend().

Usage

```
stop_backend(backend)
```

Arguments

backend	An object of class Backend as returned by the start_backend() function.

Details

This function is a convenience wrapper around the lower-lever API of parabar aimed at developers. More specifically, this function calls the stop method on the provided backend instance.

Value

The function returns void. It throws an error if:

- the value provided for the backend argument is not an instance of class Backend.
- the backend object provided is already stopped (i.e., is not active).

See Also

start_backend(), peek(), export(), evaluate(), clear(), configure_bar(), par_sapply(), par_apply(), par_lapply(), and BackendService.

Examples

```
# Create an asynchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "async")

# Check that the backend is active.
backend$active

# Check if there is anything on the backend.
peek(backend)

# Create a dummy variable.
name <- "parabar"

# Export the `name` variable in the current environment to the backend.
export(backend, "name", environment())

# Remove the dummy variable from the current environment.
rm(name)

# Check the backend to see that the variable has been exported.
peek(backend)

# Run an expression on the backend.
# Note that the symbols in the expression are resolved on the backend.
evaluate(backend, {
    # Print the name.
    print(paste0("Hello, ", name, "!"))
})

# Clear the backend.
clear(backend)

# Check that there is nothing on the backend.
peek(backend)

# Use a basic progress bar (i.e., see `parabar::Bar`).
configure_bar(type = "basic", style = 3)

# Run a task in parallel (i.e., approx. 1.25 seconds).
output <- par_sapply(backend, x = 1:10, fun = function(x) {
    # Sleep a bit.
    Sys.sleep(0.25)

    # Compute and return.
    return(x + 1)
})

# Print the output.
print(output)

# Stop the backend.
stop_backend(backend)
```

```
# Check that the backend is not active.
backend$active
```

SyncBackend *SyncBackend*

Description

This is a concrete implementation of the abstract class Backend that implements the BackendService interface. This backend executes tasks in parallel on a parallel::makeCluster() cluster synchronously (i.e., blocking the main R session).

Super classes

parabar::BackendService -> parabar::Backend -> SyncBackend

Methods

Public methods:

- SyncBackend$new()
- SyncBackend$start()
- SyncBackend$stop()
- SyncBackend$clear()
- SyncBackend$peek()
- SyncBackend$export()
- SyncBackend$evaluate()
- SyncBackend$sapply()
- SyncBackend$lapply()
- SyncBackend$apply()
- SyncBackend$get_output()
- SyncBackend$clone()

Method `new()`**:** Create a new SyncBackend object.

Usage:

```
SyncBackend$new()
```

Returns: An object of class SyncBackend.

Method `start()`**:** Start the backend.

Usage:

```
SyncBackend$start(specification)
```

Arguments:

`specification` An object of class Specification that contains the backend configuration.

Returns: This method returns void. The resulting backend must be stored in the `.cluster` private field on the `Backend` abstract class, and accessible to any concrete backend implementations via the active binding `cluster`.

Method `stop()`**:** Stop the backend.

Usage:

```
SyncBackend$stop()
```

Returns: This method returns void.

Method `clear()`**:** Remove all objects from the backend. This function is equivalent to calling `rm(list = ls(all.names = TRUE))` on each node in the backend.

Usage:

```
SyncBackend$clear()
```

Returns: This method returns void.

Method `peek()`**:** Inspect the backend for variables available in the `.GlobalEnv`.

Usage:

```
SyncBackend$peek()
```

Returns: This method returns a list of character vectors, where each element corresponds to a node in the backend. The character vectors contain the names of the variables available in the `.GlobalEnv` on each node.

Method `export()`**:** Export variables from a given environment to the backend.

Usage:

```
SyncBackend$export(variables, environment)
```

Arguments:

`variables` A character vector of variable names to export.

`environment` An environment object from which to export the variables. Defaults to the parent frame.

Returns: This method returns void.

Method `evaluate()`**:** Evaluate an arbitrary expression on the backend.

Usage:

```
SyncBackend$evaluate(expression)
```

Arguments:

`expression` An unquoted expression to evaluate on the backend.

Returns: This method returns the result of the expression evaluation.

Method `sapply()`**:** Run a task on the backend akin to `parallel::parSapply()`.

Usage:

```
SyncBackend$sapply(x, fun, ...)
```

Arguments:

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `lapply()`**:** Run a task on the backend akin to `parallel::parLapply()`.

Usage:

```
SyncBackend$lapply(x, fun, ...)
```

Arguments:

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `apply()`**:** Run a task on the backend akin to `parallel::parApply()`.

Usage:

```
SyncBackend$apply(x, margin, fun, ...)
```

Arguments:

`x` An array to pass to the `fun` function.

`margin` A numeric vector indicating the dimensions of `x` the `fun` function should be applied over. For example, for a matrix, `margin = 1` indicates applying `fun` rows-wise, `margin = 2` indicates applying `fun` columns-wise, and `margin = c(1, 2)` indicates applying `fun` element-wise. Named dimensions are also possible depending on `x`. See `parallel::parApply()` and `base::apply()` for more details.

`fun` A function to apply to `x` according to the `margin`.

`...` Additional arguments to pass to the `fun` function.

Returns: This method returns void. The output of the task execution must be stored in the private field `.output` on the `Backend` abstract class, and is accessible via the `get_output()` method.

Method `get_output()`**:** Get the output of the task execution.

Usage:

```
SyncBackend$get_output(...)
```

Arguments:

`...` Additional arguments currently not in use.

Details: This method fetches the output of the task execution after calling the `sapply()` method. It returns the output and immediately removes it from the backend. Therefore, subsequent calls to this method will return `NULL`. This method should be called after the execution of a task.

Returns: A vector, matrix, or list of the same length as `x`, containing the results of the `fun`. The output format differs based on the specific operation employed. Check out the documentation for the `apply` operations of `parallel::parallel` for more information.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SyncBackend$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`BackendService`, `Backend`, `AsyncBackend`, and `Context`.

Examples

```
# Create a specification object.
specification <- Specification$new()

# Set the number of cores.
specification$set_cores(cores = 2)

# Set the cluster type.
specification$set_type(type = "psock")

# Create a synchronous backend object.
backend <- SyncBackend$new()

# Start the cluster on the backend.
backend$start(specification)

# Check if there is anything on the backend.
backend$peek()

# Create a dummy variable.
name <- "parabar"

# Export the variable from the current environment to the backend.
backend$export("name", environment())

# Remove variable from current environment.
rm(name)

# Run an expression on the backend, using the exported variable `name`.
backend$evaluate({
    # Print the name.
    print(paste0("Hello, ", name, "!"))
})

# Run a task in parallel (i.e., approx. 1.25 seconds).
backend$sapply(
```

```
    x = 1:10,
    fun = function(x) {
        # Sleep a bit.
        Sys.sleep(0.25)

        # Compute something.
        output <- x + 1

        # Return the result.
        return(output)
    }
)

# Get the task output.
backend$get_output()

# Clear the backend.
backend$clear()

# Check that there is nothing on the cluster.
backend$peek()

# Stop the backend.
backend$stop()

# Check that the backend is not active.
backend$active
```

`TaskState` *TaskState*

Description

This class holds the state of a task deployed to an asynchronous backend (i.e., `AsyncBackend`). See the **Details** section for more information.

Details

The task state is useful to check if an asynchronous backend is free to execute other operations. A task can only be in one of the following three states at a time:

- `task_not_started`: When `TRUE`, it indicates whether the backend is free to execute another operation.
- `task_is_running`: When `TRUE`, it indicates that there is a task running on the backend.
- `task_is_completed`: When `TRUE`, it indicates that the task has been completed, but the backend is still busy because the task output has not been retrieved.

The task state is determined based on the state of the background `session` (i.e., see the `get_state` method for `callr::r_session`) and the state of the task execution inferred from polling the process (i.e., see the `poll_process` method for `callr::r_session`) as follows:

Session State	Execution State	Not Started	Is Running	Is Completed
idle	timeout	TRUE	FALSE	FALSE
busy	timeout	FALSE	TRUE	FALSE
busy	ready	FALSE	FALSE	TRUE

Active bindings

`task_not_started` A logical value indicating whether the task has been started. It is used to determine if the backend is free to execute another operation.

`task_is_running` A logical value indicating whether the task is running.

`task_is_completed` A logical value indicating whether the task has been completed and the output needs to be retrieved.

Methods

Public methods:

- `TaskState$new()`
- `TaskState$clone()`

Method `new()`**:** Create a new `TaskState` object and determine the state of a task on a given background `session`.

Usage:

```
TaskState$new(session)
```

Arguments:

`session` A `callr::r_session` object.

Returns: An object of class `TaskState`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TaskState$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`SessionState`, `AsyncBackend` and `ProgressTrackingContext`.

Examples

```
# Handy function to print the task states all at once.
check_state <- function(session) {
    # Create a task state object and determine the state.
    task_state <- TaskState$new(session)

    # Print the state.
    cat(
        "Task not started: ", task_state$task_not_started, "\n",
        "Task is running: ", task_state$task_is_running, "\n",
        "Task is completed: ", task_state$task_is_completed, "\n",
        sep = ""
    )
}

# Create a specification object.
specification <- Specification$new()

# Set the number of cores.
specification$set_cores(cores = 2)

# Set the cluster type.
specification$set_type(type = "psock")

# Create an asynchronous backend object.
backend <- AsyncBackend$new()

# Start the cluster on the backend.
backend$start(specification)

# Check that the task has not been started (i.e., the backend is free).
check_state(backend$cluster)

{
    # Run a task in parallel (i.e., approx. 0.25 seconds).
    backend$sapply(
        x = 1:10,
        fun = function(x) {
            # Sleep a bit.
            Sys.sleep(0.05)

            # Compute something.
            output <- x + 1

            # Return the result.
            return(output)
        }
    )

    # And immediately check the state to see that the task is running.
    check_state(backend$cluster)
}
```

```
# Sleep for a bit to wait for the task to complete.
Sys.sleep(1)

# Check that the task is completed (i.e., the output needs to be retrieved).
check_state(backend$cluster)

# Get the output.
output <- backend$get_output(wait = TRUE)

# Check that the task has not been started (i.e., the backend is free again).
check_state(backend$cluster)

# Stop the backend.
backend$stop()
```

`UserApiConsumer` *UserApiConsumer*

Description

This class is an opinionated interface around the developer API of the `parabar` package. See the **Details** section for more information on how this class works.

Details

This class acts as a wrapper around the `R6::R6` developer API of the `parabar` package. In a nutshell, it provides an opinionated interface by wrapping the developer API in simple functional calls. More specifically, for executing a task in parallel, this class performs the following steps:

- Validates the backend provided.
- Instantiates an appropriate `parabar` context based on the backend. If the backend supports progress tracking (i.e., the backend is an instance of `AsyncBackend`), a progress tracking context (i.e., `ProgressTrackingContext`) is instantiated and used. Otherwise, a regular context (i.e., `Context`) is instantiated. A regular context is also used if the progress tracking is disabled via the `Options` instance.
- Registers the `backend` with the context.
- Instantiates and configures the progress bar based on the `Options` instance in the session `base::.Options` list.
- Executes the task in parallel, and displays a progress bar if appropriate.
- Fetches the results from the backend and returns them.

Methods

Public methods:

- UserApiConsumer$sapply()
- UserApiConsumer$lapply()
- UserApiConsumer$apply()
- UserApiConsumer$clone()

Method `sapply()`**:** Execute a task in parallel akin to `parallel::parSapply()`.

Usage:

```
UserApiConsumer$sapply(backend, x, fun, ...)
```

Arguments:

`backend` An object of class `Backend` as returned by the `start_backend()` function. It can also be `NULL` to run the task sequentially via `base::sapply()`.

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: A vector of the same length as `x` containing the results of the `fun`. The output format resembles that of `base::sapply()`.

Method `lapply()`**:** Execute a task in parallel akin to `parallel::parLapply()`.

Usage:

```
UserApiConsumer$lapply(backend, x, fun, ...)
```

Arguments:

`backend` An object of class `Backend` as returned by the `start_backend()` function. It can also be `NULL` to run the task sequentially via `base::lapply()`.

`x` An atomic vector or list to pass to the `fun` function.

`fun` A function to apply to each element of `x`.

`...` Additional arguments to pass to the `fun` function.

Returns: A list of the same length as `x` containing the results of the `fun`. The output format resembles that of `base::lapply()`.

Method `apply()`**:** Execute a task in parallel akin to `parallel::parApply()`.

Usage:

```
UserApiConsumer$apply(backend, x, margin, fun, ...)
```

Arguments:

`backend` An object of class `Backend` as returned by the `start_backend()` function. It can also be `NULL` to run the task sequentially via `base::apply()`.

`x` An array to pass to the `fun` function.

`margin` A numeric vector indicating the dimensions of `x` the `fun` function should be applied over. For example, for a matrix, `margin = 1` indicates applying `fun` rows-wise, `margin = 2` indicates applying `fun` columns-wise, and `margin = c(1, 2)` indicates applying `fun` element-wise. Named dimensions are also possible depending on `x`. See `parallel::parApply()` and `base::apply()` for more details.

`fun` A function to apply to `x` according to the `margin`.

`...` Additional arguments to pass to the `fun` function.

Returns: The dimensions of the output vary according to the `margin` argument. Consult the documentation of `base::apply()` for a detailed explanation on how the output is structured.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
UserApiConsumer$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`start_backend()`, `stop_backend()`, `configure_bar()`, `par_sapply()`, and `par_lapply()`.

Examples

```
# Define a simple task.
task <- function(x) {
    # Perform computations.
    Sys.sleep(0.01)

    # Return the result.
    return(x + 1)
}

# Start an asynchronous backend.
backend <- start_backend(cores = 2, cluster_type = "psock", backend_type = "async")

# Change the progress bar options.
configure_bar(type = "modern", format = "[:bar] :percent")

# Create an user API consumer.
consumer <- UserApiConsumer$new()

# Execute the task using the `sapply` parallel operation.
output_sapply <- consumer$sapply(backend = backend, x = 1:200, fun = task)

# Print the head of the `sapply` operation output.
head(output_sapply)

# Execute the task using the `sapply` parallel operation.
output_lapply <- consumer$lapply(backend = backend, x = 1:200, fun = task)

# Print the head of the `lapply` operation output.
head(output_lapply)

# Stop the backend.
stop_backend(backend)
```

`Warning` *Package Warnings*

Description

This class contains static methods for throwing warnings with informative messages.

Format

`Warning$requested_cluster_cores_too_low()` Warning for not requesting enough cluster cores.

`Warning$requested_cluster_cores_too_high()` Warning for requesting too many cluster cores.

`Warning$requested_cluster_type_not_supported()` Warning for requesting an unsupported cluster type.

`Warning$progress_not_supported_for_backend()` Warning for using a backend incompatible with progress tracking.

`Warning$error_in_backend_finalizer()` Warning for errors in the backend finalizer during garbage collection.

Index