Package 'transltr'

February 14, 2025

Title Support Many Languages in R

Version 0.1.0

Description An object model for source text and translations. Find and extract translatable strings. Provide translations and seamlessly retrieve them at runtime.

License MIT + file LICENSE

URL https://transltr.ununoctium.dev

BugReports https://github.com/jeanmathieupotvin/transltr/issues

Encoding UTF-8

Language en

RoxygenNote 7.3.2

Config/testthat/edition 3

Depends R (>= 4.3)

Imports digest, R6, stringi, utils, yaml

Suggests covr, devtools, lifecycle, microbenchmark, pkgdown, testthat (>= 3.0.0), usethis, withr

Collate 'aaa.R' 'assert.R' 'class-location.R' 'class-text.R' 'class-translator.R' 'find-source-in-exprs.R' 'find-source.R' 'flat.R' 'hash.R' 'language.R' 'normalize.R' 'serialize.R' 'text-io.R' 'translator-io.R' 'transltr-package.R' 'utils-format-vector.R' 'utils-map.R' 'utils-nullish-op.R' 'utils-stop.R' 'utils-strings.R' 'uuid.R' 'zzz.R'

NeedsCompilation no

Author Jean-Mathieu Potvin [aut, cre, cph], Jérôme Lavoué [ctb, fnd, rev] (<https://orcid.org/0000-0003-4950-5475>)

Maintainer Jean-Mathieu Potvin <jeanmathieupotvin@ununoctium.dev>

Repository CRAN

Date/Publication 2025-02-14 16:40:02 UTC

Contents

find_source 2
language_set 5
translator 7
translator_read 15

Index **19**

`find_source` *Find Source Text*

Description

Find and extract source text that must be translated.

Usage

```
find_source(
  path = ".",
  encoding = "UTF-8",
  verbose = getOption("transltr.verbose", TRUE),
  tr = translator(),
  interface = NULL
)

find_source_in_files(
  paths = character(),
  encoding = "UTF-8",
  verbose = getOption("transltr.verbose", TRUE),
  algorithm = algorithms(),
  interface = NULL
)
```

Arguments

`path`	A non-empty and non-NA character string. A path to a directory containing R source scripts. All subdirectories are searched. Files that do not have a `.R`, or `.Rprofile` extension are skipped.
`encoding`	A non-empty and non-NA character string. The source character encoding. In almost all cases, this should be UTF-8. Other encodings are internally re-encoded to UTF-8 for portability.
`verbose`	A non-NA logical value. Should progress information be reported?
`tr`	A `Translator` object.
`interface`	A `name`, a `call` object, or a `NULL`. A reference to an alternative (custom) function used to translate text. If a `call` object is passed to `interface`, it **must** be to operator `::`. Calls to method `Translator$translate()` are ignored and calls to `interface` are extracted instead. See Details below.

`paths`	A character vector of non-empty and non-NA values. A set of paths to R source scripts that must be searched.
`algorithm`	A non-empty and non-NA character string equal to `"sha1"`, or `"utf8"`. The algorithm to use when hashing source information for identification purposes.

Details

`find_source()` and `find_source_in_files()` look for calls to method `Translator$translate()` in R scripts and convert them to `Text` objects. The former further sets these resulting objects into a `Translator` object. See argument `tr`.

`find_source()` and `find_source_in_files()` work on a purely lexical basis. The source code is parsed but never evaluated (aside from extracted literal character vectors).

- The underlying `Translator` object is never evaluated and does not need to exist (placeholders may be used in the source code).
- Only **literal** character vectors can be passed to arguments of method `Translator$translate()`.

Interfaces:

In some cases, it may not be desirable to call method `Translator$translate()` directly. A custom function wrapping (*interfacing*) this method may always be used as long as it has the same signature as method `Translator$translate()`. In other words, it must minimally have two formal arguments: `...` and `source_lang`.

Custom interfaces must be passed to `find_source()` and `find_source_in_files()` for extraction purposes. Since these functions work on a lexical basis, interfaces can be placeholders in the source code (non- existent bindings) at the time these functions are called. However, they must be bound to a function (ultimately) calling `Translator$translate()` at runtime.

Custom interfaces are passed to `find_source()` and `find_source_in_files()` as `name` or `call` objects in a variety of ways. The most straightforward way is to use `base::quote()`. See Examples below.

Methodology:

`find_source()` and `find_source_in_files()` go through these steps to extract source text from a single R script.

1. It is read with `text_read()` and re-encoded to UTF-8 if necessary.
2. It is parsed with `parse()` and underlying tokens are extracted from parsed expressions with `utils::getParseData()`.
3. Each expression (`expr`) token is converted to language objects with `str2lang()`. Parsing errors and invalid expressions are silently skipped.
4. Valid `call` objects stemming from step 3 are filtered with `is_source()`.
5. Calls to method `Translator$translate()` or to `interface` stemming from step 4 are coerced to `Text` objects with `as_text()`.

These steps are repeated for each R script. `find_source()` further merges all resulting `Text` objects into a coherent set with `merge_texts()` (identical source code is merged into single `Text` entities).

Extracted character vectors are always normalized for consistency (at step 5). See `normalize()` for more information.

Limitations:

The current version of transltr can only handle **literal** character vectors. This means it cannot resolve non-trivial expressions that depends on a *state*. All values passed to argument ... of method Translator$translate() must yield character vectors (trivially).

Value

find_source() returns an R6 object of class Translator. If an existing Translator object is passed to tr, it is modified in place and returned.

find_source_in_files() returns a list of Text objects. It may contain duplicated elements, depending on the extracted contents.

See Also

Translator, Text, normalize(), translator_read(), translator_write(), base::quote(), base::call(), base::as.name()

Examples

```
# Create a directory containing dummy R scripts for illustration purposes.
temp_dir   <- file.path(tempdir(TRUE), "find-source")
temp_files <- file.path(temp_dir, c("ex-script-1.R", "ex-script-2.R"))
dir.create(temp_dir, showWarnings = FALSE, recursive = TRUE)

cat(
  "tr$translate('Hello, world!')",
  "tr$translate('Farewell, world!')",
  sep  = "\n",
  file = temp_files[[1L]])
cat(
  "tr$translate('Hello, world!')",
  "tr$translate('Farewell, world!')",
  sep  = "\n",
  file = temp_files[[2L]])

# Extract calls to method Translator$translate().
find_source(temp_dir)
find_source_in_files(temp_files)

# Use custom functions.
# For illustrations purposes, assume the package
# exports an hypothetical translate() function.
cat(
  "translate('Hello, world!')",
  "transtlr::translate('Farewell, world!')",
  sep  = "\n",
  file = temp_files[[1L]])
cat(
  "translate('Hello, world!')",
  "transltr::translate('Farewell, world!')",
  sep  = "\n",
```

```
  file = temp_files[[2L]])

# Extract calls to translate() and transltr::translate().
# Since find_source() and find_source_in_files() work on
# a lexical basis, these are always considered to be two
# distinct functions. They also don't need to exist in the
# R session calling find_source() and find_source_in_files().
find_source(temp_dir, interface = quote(translate))
find_source_in_files(temp_files, interface = quote(transltr::translate))
```

`language_set` *Get or Set Language*

Description

Get or set the current, and source languages.

They are registered as environment variables named `TRANSLTR_LANGUAGE`, and `TRANSLTR_SOURCE_LANGUAGE`.

Usage

```
language_set(lang = "en")

language_get()

language_source_set(lang = "en")

language_source_get()
```

Arguments

`lang`	A non-empty and non-NA character string. The underlying language. A language is usually a code (of two or three letters) for a native language name. While users retain full control over codes, it is best to use language codes stemming from well-known schemes such as IETF BCP 47, or ISO 639-1 to maximize portability and cross-compatibility.

Details

The language and the source language can always be temporarily changed. See argument `lang` of method `Translator$translate()` for more information.

The underlying locale is left as is. To change an R session's locale, use `Sys.setlocale()` or `Sys.setLanguage()` instead. See below for more information.

Value

`language_set()`, and `language_source_set()` return `NULL`, invisibly. They are used for their side-effect of setting environment variables `TRANSLTR_LANGUAGE` and `TRANSLTR_SOURCE_LANGUAGE`, respectively.

`language_get()` returns a character string. It is the current value of environment variable `TRANSLTR_LANGUAGE`. It is empty if the latter is unset.

`language_source_get()` returns a character string. It is the current value of environment variable `TRANSLTR_SOURCE_LANGUAGE`. It returns `"en"` if the latter is unset.

Locales versus languages

A locale is a set of multiple low-level settings that relate to the user's language and region. The *language* itself is just one parameter among many others.

Modifying a locale on-the-fly *can* be considered risky in some situations. It may not be the optimal solution for merely changing textual representations of a program or an application at runtime, as it may introduce unintended changes and induce subtle bugs that are harder to fix.

Moreover, it makes sense for some applications and/or programs such as Shiny applications to decouple the front-end's current language (what *users* see) from the back-end's locale (what *developers* see). A UI may be displayed in a certain language while keeping logs and R internal messages, warnings, and errors as is.

Consequently, the language setting of `transltr` is purposely kept separate from the underlying locale and removes the complexity of having to support many of them. Users can always change both the locale and the `language` parameter of the package. See Examples.

Note

Environment variables are used because they can be shared among different processes. This matters when using parallel and/or concurrent R sessions. It can further be shared among direct and transitive dependencies (other packages that rely on `transltr`).

Examples

```
# Change the language parameters (globally).
language_source_set("en")
language_set("fr")

language_source_get()  ## Outputs "en"
language_get()         ## Outputs "fr"

# Change both the language parameter and the locale.
# Note that while users control how languages are named
# for language_set(), they do not for Sys.setLanguage().
language_set("fr")
Sys.setLanguage("fr-CA")

# Reset settings.
language_source_set(NULL)
language_set(NULL)
```

```
# Source language has a default value.
language_source_get()  ## Outputs "en"
```

`translator` *Source Text and Translations*

Description

Structure and manipulate the source text of a project and its translations.

Usage

```
translator(..., id = uuid(), algorithm = algorithms())

is_translator(x)

## S3 method for class 'Translator'
format(x, ...)

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

Arguments

`...`	Usage depends on the underlying function. • Any number of `Text` objects and/or named character strings for `translator()` (in no preferred order). • Further arguments passed to or from other methods for `format()`, and `print()`.
`id`	A non-empty and non-NA character string. A globally unique identifier for the `Translator` object. Beware of collisions when using user-defined values.
`algorithm`	A non-empty and non-NA character string equal to `"sha1"`, or `"utf8"`. The algorithm to use when hashing source information for identification purposes.
`x`	Any R object.

Details

A `Translator` object encapsulates the source text of a project (or any other *context*) and all related translations. Under the hood, `Translator` objects are collections of `Text` objects. These do most of the work. They are treated as lower-level component and in typical situations, users rarely interact with them.

`Translator` objects can be saved and exported with `translator_write()`. They can be imported back into an R session with `translator_read()`.

Value

`translator()` returns an `R6` object of class `Translator`.

`is_translator()` returns a logical value.

`format()` returns a character vector.

`print()` returns argument `x` invisibly.

Active bindings

`id` A non-empty and non-NA character string. A globally unique identifier for the underlying object. Beware of plausible collisions when using user-defined values.

`algorithm` A non-empty and non-NA character string equal to `"sha1"`, or `"utf8"`. The algorithm to use when hashing source information for identification purposes.

`hashes` A character vector of non-empty and non-NA values, or `NULL`. The set of all `hash` exposed by registered `Text` objects. If there is none, `hashes` is `NULL`. This is a **read-only** field updated whenever field `algorithm` is updated.

`source_texts` A character vector of non-empty and non-NA values, or `NULL`. The set of all registered source texts. If there is none, `source_texts` is `NULL`. This is a **read-only** field.

`source_langs` A character vector of non-empty and non-NA values, or `NULL`. The set of all registered source languages. This is a **read-only** field.

- If there is none, `source_langs` is `NULL`.
- If there is one unique value, `source_langs` is an unnamed character string.
- Otherwise, it is a named character vector.

`languages` A character vector of non-empty and non-NA values, or `NULL`. The set of all registered `languages` (codes). If there is none, `languages` is `NULL`. This is a **read-only** field.

`native_languages` A named character vector of non-empty and non-NA values, or `NULL`. A map (bijection) of `languages` (codes) to native language names. Names are codes and values are native languages. If there is none, `native_languages` is `NULL`.

While users retain full control over `native_languages`, it is best to use well-known schemes such as IETF BCP 47, or ISO 639-1. Doing so maximizes portability and cross-compatibility between packages.

Update this field with method `$set_native_languages()`. See below for more information.

Methods

Public methods:

- `Translator$new()`
- `Translator$translate()`
- `Translator$get_translation()`
- `Translator$get_text()`
- `Translator$set_text()`
- `Translator$set_texts()`
- `Translator$rm_text()`
- `Translator$set_native_languages()`

- `Translator$set_default_value()`

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

Usage:

```
Translator$new(id = uuid(), algorithm = algorithms())
```

Arguments:

`id` A non-empty and non-NA character string. A globally unique identifier for the `Translator` object. Beware of collisions when using user-defined values.

`algorithm` A non-empty and non-NA character string equal to `"sha1"`, or `"utf8"`. The algorithm to use when hashing source information for identification purposes.

Returns: An `R6` object of class `Translator`.

Examples:

```
# Consider using translator() instead.
tr <- Translator$new()
```

Method `translate()`**:** Translate source text.

Usage:

```
Translator$translate(
  ...,
  lang = language_get(),
  source_lang = language_source_get()
)
```

Arguments:

`...` Any number of vectors containing atomic elements. Each vector is normalized as a paragraph.

- Elements are coerced to character values.
- NA values and empty strings are discarded.
- Multi-line strings are supported and encouraged. Blank lines are interpreted (two or more newline characters) as paragraph separators.

`lang` A non-empty and non-NA character string. The underlying language.

A language is usually a code (of two or three letters) for a native language name. While users retain full control over codes, it is best to use language codes stemming from well-known schemes such as IETF BCP 47, or ISO 639-1 to maximize portability and cross-compatibility.

`source_lang` A non-empty and non-NA character string. The language of the source text. See argument `lang` for more information.

Details: See `normalize()` for further details on how `...` is normalized.

Returns: A character string. If there is no corresponding translation, the value passed to method `$set_default_value()` is returned. `NULL` is returned by default.

Examples:

```
tr <- Translator$new()
tr$set_text(en = "Hello, world!", fr = "Bonjour, monde!")
tr$translate("Hello, world!", lang = "en")  ## Outputs "Hello, world!"
tr$translate("Hello, world!", lang = "fr")  ## Outputs "Bonjour, monde!"
```

Method `get_translation()`**:** Extract a translation or a source text.

Usage:

```
Translator$get_translation(hash = "", lang = "")
```

Arguments:

`hash` A non-empty and non-NA character string. The unique identifier of the requested source text.

`lang` A non-empty and non-NA character string. The underlying language.
A language is usually a code (of two or three letters) for a native language name. While users retain full control over codes, it is best to use language codes stemming from well-known schemes such as IETF BCP 47, or ISO 639-1 to maximize portability and cross-compatibility.

Returns: A character string. If there is no corresponding translation, the value passed to method `$set_default_value()` is returned. `NULL` is returned by default.

Examples:

```
tr <- Translator$new()
tr$set_text(en = "Hello, world!")

# Consider using translate() instead.
tr$get_translation("256e0d7", "en")  ## Outputs "Hello, world!"
```

Method `get_text()`**:** Extract a source text and its translations.

Usage:

```
Translator$get_text(hash = "")
```

Arguments:

`hash` A non-empty and non-NA character string. The unique identifier of the requested source text.

Returns: A `Text` object, or `NULL`.

Examples:

```
tr <- Translator$new()
tr$set_text(en = "Hello, world!")

tr$get_translation("256e0d7", "en")  ## Outputs "Hello, world!"
```

Method `set_text()`**:** Register a source text.

Usage:

```
Translator$set_text(..., source_lang = language_source_get())
```

Arguments:

`...` Passed as is to `text()`.

`source_lang` Passed as is to `text()`.

Returns: A `NULL`, invisibly.

Examples:

```
tr <- Translator$new()

tr$set_text(en = "Hello, world!", location())
```

Method `set_texts()`**:** Register one or more source texts.

Usage:

```
Translator$set_texts(...)
```

Arguments:

`...` Any number of `Text` objects.

Details: This method calls `merge_texts()` to merge all values passed to `...` together with previously registered `Text` objects. The underlying registered source texts, translations, and `Location` objects won't be duplicated.

Returns: A `NULL`, invisibly.

Examples:

```
# Set source language.
language_source_set("en")

tr <- Translator$new()

# Create Text objects.
txt1 <- text(
  location("a", 1L, 2L, 3L, 4L),
  en = "Hello, world!",
  fr = "Bonjour, monde!")

txt2 <- text(
  location("b", 5L, 6L, 7L, 8L),
  en = "Farewell, world!",
  fr = "Au revoir, monde!")

tr$set_texts(txt1, txt2)
```

Method `rm_text()`**:** Remove a registered source text.

Usage:

```
Translator$rm_text(hash = "")
```

Arguments:

`hash` A non-empty and non-NA character string identifying the source text to remove.

Returns: A `NULL`, invisibly.

Examples:

```
tr <- Translator$new()
tr$set_text(en = "Hello, world!")

tr$rm_text("256e0d7")
```

Method `set_native_languages()`**:** Map a language code to a native language name.

Usage:

```
Translator$set_native_languages(...)
```

Arguments:

`...` Any number of named, non-empty, and non-NA character strings. Names are codes and values are native languages. See field `native_languages` for more information.

Returns: A `NULL`, invisibly.

Examples:

```
tr <- Translator$new()

tr$set_native_languages(en = "English", fr = "Français")

# Remove existing entries.
tr$set_native_languages(fr = NULL)
```

Method `set_default_value()`**:** Register a default value to return when there is no corresponding translations for the requested language.

Usage:

```
Translator$set_default_value(value = NULL)
```

Arguments:

`value` A `NULL` or a non-NA character string. It can be empty. The former is returned by default.

Details: This modifies what methods `$translate()` and `$get_translation()` returns when there is no translation for `lang`.

Returns: A `NULL`, invisibly.

Examples:

```
tr <- Translator$new()
tr$set_default_value("<unavailable>")
```

See Also

`find_source()`, `translator_read()`, `translator_write()`

Examples

```
# Set source language.
language_source_set("en")

# Create a Translator object.
# This would normally be done automatically
# by find_source(), or translator_read().
tr <- translator(
  id = "test-translator",
  en = "English",
  es = "Español",
  fr = "Français",
  text(
    location("a", 1L, 2L, 3L, 4L),
```

```
    en = "Hello, world!",
    fr = "Bonjour, monde!"),
  text(
    location("b", 1L, 2L, 3L, 4L),
    en = "Farewell, world!",
    fr = "Au revoir, monde!"))

is_translator(tr)

# Translator objects has a specific format.
# print() calls format() internally, as expected.
print(tr)


## ------------------------------------------------
## Method `Translator$new`
## ------------------------------------------------

# Consider using translator() instead.
tr <- Translator$new()

## ------------------------------------------------
## Method `Translator$translate`
## ------------------------------------------------

tr <- Translator$new()
tr$set_text(en = "Hello, world!", fr = "Bonjour, monde!")
tr$translate("Hello, world!", lang = "en")  ## Outputs "Hello, world!"
tr$translate("Hello, world!", lang = "fr")  ## Outputs "Bonjour, monde!"

## ------------------------------------------------
## Method `Translator$get_translation`
## ------------------------------------------------

tr <- Translator$new()
tr$set_text(en = "Hello, world!")

# Consider using translate() instead.
tr$get_translation("256e0d7", "en")  ## Outputs "Hello, world!"

## ------------------------------------------------
## Method `Translator$get_text`
## ------------------------------------------------

tr <- Translator$new()
tr$set_text(en = "Hello, world!")

tr$get_translation("256e0d7", "en")  ## Outputs "Hello, world!"

## ------------------------------------------------
## Method `Translator$set_text`
## ------------------------------------------------
```

```
tr <- Translator$new()

tr$set_text(en = "Hello, world!", location())

## ------------------------------------------------
## Method `Translator$set_texts`
## ------------------------------------------------

# Set source language.
language_source_set("en")

tr <- Translator$new()

# Create Text objects.
txt1 <- text(
  location("a", 1L, 2L, 3L, 4L),
  en = "Hello, world!",
  fr = "Bonjour, monde!")

txt2 <- text(
  location("b", 5L, 6L, 7L, 8L),
  en = "Farewell, world!",
  fr = "Au revoir, monde!")

tr$set_texts(txt1, txt2)

## ------------------------------------------------
## Method `Translator$rm_text`
## ------------------------------------------------

tr <- Translator$new()
tr$set_text(en = "Hello, world!")

tr$rm_text("256e0d7")

## ------------------------------------------------
## Method `Translator$set_native_languages`
## ------------------------------------------------

tr <- Translator$new()

tr$set_native_languages(en = "English", fr = "Français")

# Remove existing entries.
tr$set_native_languages(fr = NULL)

## ------------------------------------------------
## Method `Translator$set_default_value`
## ------------------------------------------------

tr <- Translator$new()
tr$set_default_value("<unavailable>")
```

translator_read *Read and Write Translations*

Description

Export `Translator` objects to text files and import such files back into R as `Translator` objects.

Usage

```
translator_read(
  path = getOption("transltr.path"),
  encoding = "UTF-8",
  verbose = getOption("transltr.verbose", TRUE),
  translations = TRUE
)

translator_write(
  tr = translator(),
  path = getOption("transltr.path"),
  overwrite = FALSE,
  verbose = getOption("transltr.verbose", TRUE),
  translations = TRUE
)

translations_read(path = "", encoding = "UTF-8", tr = NULL)

translations_write(tr = translator(), path = "", lang = "")

translations_paths(
  tr = translator(),
  parent_dir = dirname(getOption("transltr.path"))
)
```

Arguments

`path`	A non-empty and non-NA character string. A path to a file to read from, or write to. • This file must be a Translator file for `translator_read()`. • This file must be a translations file for `translations_read()`. See Details for more information. `translator_write()` automatically creates the parent directories of `path` (recursively) if they do not exist.
`encoding`	A non-empty and non-NA character string. The source character encoding. In almost all cases, this should be UTF-8. Other encodings are internally re-encoded to UTF-8 for portability.
`verbose`	A non-NA logical value. Should progress information be reported?

translations	A non-NA logical value. Should translations files also be read, or written along with path (the Translator file)?
tr	A Translator object. This argument is NULL by default for translations_read(). If a Translator object is passed to this function, it will read translations and further register them (as long as they correspond to an existing source text).
overwrite	A non-NA logical value. Should existing files be overwritten? If such files are detected and overwrite is set equal to TRUE, an error is thrown.
lang	A non-empty and non-NA character string. The underlying language. A language is usually a code (of two or three letters) for a native language name. While users retain full control over codes, it is best to use language codes stemming from well-known schemes such as IETF BCP 47, or ISO 639-1 to maximize portability and cross-compatibility.
parent_dir	A non-empty and non-NA character string. A path to a parent directory.

Details

The information contained within a Translator object is split: translations are reorganized by language and exported independently from other fields.

translator_write() creates two types of file: a single *Translator file*, and zero, or more *translations files*. These are plain text files that can be inspected and modified using a wide variety of tools and systems. They target different audiences:

- the Translator file is useful to developers, and
- translations files are meant to be shared with non-technical collaborators such as translators.

translator_read() first reads a Translator file and creates a Translator object from it. It then calls translations_paths() to list expected translations files (that should normally be stored alongside the Translator file), attempts to read them, and registers successfully imported translations.

There are two requirements.

- All files must be stored in the same directory. By default, this is set equal to inst/transltr/ (see getOption("transltr.path")).
- Filenames of translations files are standardized and must correspond to languages (language codes, see lang).

The inner workings of the serialization process are thoroughly described in serialize().

Translator file:

A Translator file contains a YAML (1.1) representation of a Translator object stripped of all its translations except those that are registered as source text.

Translations files:

A translations file contains a FLAT representation of a set of translations sharing the same target language. This format attempts to be as simple as possible for non-technical collaborators.

Value

`translator_read()` returns an `R6` object of class `Translator`.

`translator_write()` returns `NULL`, invisibly. It is used for its side-effects of

- creating a Translator file to the location given by `path`, and
- creating further translations file(s) in the same directory if `translations` is `TRUE`.

`translations_read()` returns an S3 object of class `ExportedTranslations`.

`translations_write()` returns `NULL`, invisibly.

`translations_paths()` returns a named character vector.

See Also

`Translator`, `serialize()`

Examples

```
# Set source language.
language_source_set("en")

# Create a path to a temporary Translator file.
temp_path <- tempfile(pattern = "translator_", fileext = ".yml")
temp_dir  <- dirname(temp_path)  ## tempdir() could also be used

# Create a Translator object.
# This would normally be done by find_source(), or translator_read().
tr <- translator(
  id = "test-translator",
  en = "English",
  es = "Español",
  fr = "Français",
  text(
    en = "Hello, world!",
    fr = "Bonjour, monde!"),
  text(
    en = "Farewell, world!",
    fr = "Au revoir, monde!"))

# Export it. This creates 3 files: 1 Translator file, and 2 translations
# files because two non-source languages are registered. The file for
# language "es" contains placeholders and must be completed.
translator_write(tr, temp_path)
translator_read(temp_path)

# Translations can be read individually.
translations_files <- translations_paths(tr, temp_dir)
translations_read(translations_files[["es"]])
translations_read(translations_files[["fr"]])

# This is rarely useful, but translations can also be exported individually.
# You may use this to add a new language, as long as it has an entry in the
```

```
# underlying Translator object (or file).
tr$set_native_languages(el = "Greek")

translations_files <- translations_paths(tr, temp_dir)

translations_write(tr, translations_files[["el"]], "el")
translations_read(file.path(temp_dir, "el.txt"))
```

Index

$get_translation(), *12*
$set_default_value(), *9*, *10*
$set_native_languages(), *8*
$translate(), *12*

as_text(), *3*
atomic, *9*

base::as.name(), *4*
base::call(), *4*
base::quote(), *3*, *4*

call, *2*, *3*

errors, *6*
ExportedTranslations, *17*

find_source, 2
find_source(), *3*, *4*, *12*
find_source_in_files *(*find_source*)*, 2
find_source_in_files(), *3*, *4*
FLAT, *16*
format(), *7*, *8*
format.Translator *(*translator*)*, 7

is_source(), *3*
is_translator *(*translator*)*, 7
is_translator(), *8*

language_get *(*language_set*)*, 5
language_get(), *6*
language_set, 5
language_set(), *6*
language_source_get *(*language_set*)*, 5
language_source_get(), *6*
language_source_set *(*language_set*)*, 5
language_source_set(), *6*
Location, *11*

merge_texts(), *3*, *11*
messages, *6*

name, *2*, *3*
normalize(), *3*, *4*, *9*

parse(), *3*
print(), *7*, *8*
print.Translator *(*translator*)*, 7

R6, *4*, *8*, *9*, *17*

serialize(), *16*, *17*
str2lang(), *3*
Sys.setLanguage(), *5*
Sys.setlocale(), *5*

Text, *3*, *4*, *7*, *8*, *10*, *11*
text(), *10*
text_read(), *3*
translations_paths *(*translator_read*)*, 15
translations_paths(), *16*, *17*
translations_read *(*translator_read*)*, 15
translations_read(), *15–17*
translations_write *(*translator_read*)*, 15
translations_write(), *17*
Translator, *2–4*, *7–9*, *15–17*
Translator *(*translator*)*, 7
translator, 7
translator(), *7*, *8*
Translator$translate(), *2–5*
translator_read, 15
translator_read(), *4*, *7*, *12*, *15–17*
translator_write *(*translator_read*)*, 15
translator_write(), *4*, *7*, *12*, *15–17*
transltr, *4*, *6*

utils::getParseData(), *3*

warnings, *6*