Package 'storr'

May 31, 2018

Title Simple Key Value Stores

Version 1.2.0

Description Creates and manages simple key-value stores. These can use a variety of approaches for storing the data. This package implements the base methods and support for file system, in-memory and DBI-based database stores.

Depends R (>= 3.1.0)

License MIT + file LICENSE

LazyData true

URL https://github.com/richfitz/storr

BugReports https://github.com/richfitz/storr/issues

Imports R6 (>= 2.1.0), digest

Suggests DBI (>= 0.6), RSQLite (>= 1.1-2), RPostgres, knitr, mockr, progress, rbenchmark, testthat (>= 1.0.0)

VignetteBuilder knitr

RoxygenNote 6.0.1

Encoding UTF-8

NeedsCompilation no

Author Rich FitzJohn [aut, cre]

Maintainer Rich FitzJohn <rich.fitzjohn@gmail.com>

Repository CRAN

Date/Publication 2018-05-31 08:02:55 UTC

R topics documented:

`driver_redis_api` *Defunct functions*

Description

Defunct functions

Usage

```
driver_redis_api(...)

storr_redis_api(...)
```

Arguments

`...`	parameters (now all dropped as dots)

Details

The redis functions (`driver_redis_api` and `storr_redis_api`) have been moved out of this package and into Redis. I don't believe anyone is using them at the time of the move so this is being done fairly abruptly - this is unfortunate, but necessary to avoid a circular dependency! The new functions are simply `redux::driver_redis_api` and `redux::storr_redis_api`, along with a helper function `redux::storr_hiredis` which also creates the connection.

`driver_remote` *Remote storr*

Description

Create a storr that keeps rds-serialised objects on a remote location. This is the abstract interface (which does not do anything useful) but which can be used with file operation driver to store files elsewhere. This is not intended for end-user use so there is no `storr_remote` function. Instead this function is designed to support external packages that implement the details. For a worked example, see the package tests (`helper-remote.R`). In the current implementation these build off of the `driver_rds` driver by copying files to some remote location.

Usage

```
driver_remote(ops, ..., path_local = NULL)
```

Arguments

ops	A file operations object. See tests for now to see what is required to implement one.
...	Arguments to pass through to driver_rds, including compress, mangle_key, mangle_key_pad and hash_algorithm.
path_local	Path to a local cache. This can be left as NULL, in which case a per-session cache will be used. Alternatively, explicitly set to a path and the cache can be reused over sessions. Only storr *values* (i.e., objects) are cached - the key-to-value mapping is always fetched from the remote storage.

Author(s)

Rich FitzJohn

---

encode64 *Base64 encoding and decoding*

Description

Base64 encoding. By default uses the RFC 4648 dialect (file/url encoding) where characters 62 and 63 are "-" and "_". Pass in "+" and "/" to get the RFC 1421 variant (as in other R packages that do base64 encoding).

Usage

```
encode64(x, char62 = "-", char63 = "_", pad = TRUE)

decode64(x, char62 = "-", char63 = "_", error = TRUE)
```

Arguments

x	A string or vector of strings to encode/decode
char62	Character to use for the 62nd index
char63	Character to use for the 63rd index
pad	Logical, indicating if strings should be padded with = characters (as RFC 4648 requires)
error	Throw an error if the decoding fails. If FALSE then NA_character_ values are returned for failures.

Examples

```
x <- encode64("hello")
x
decode64(x)

# Encoding things into filename-safe strings is the reason for
# this function:
encode64("unlikely/to be @ valid filename")
```

`fetch_hook_read` *Hook to fetch a resource from a file.*

Description

Hook to fetch a resource from a file, for use with driver_external. We take two functions as arguments: the first converts a key/namespace pair into a filename, and the second reads from that filename. Because many R functions support reading from URLs `fetch_hook_read` can be used to read from remote resources.

Usage

```
fetch_hook_read(fpath, fread)
```

Arguments

`fpath`	Function to convert `key, namespace` into a file path
`fread`	Function for converting `filename` into an R pobject

Details

For more information about using this, see `storr_external` (this can be used as a `fetch_hook` argument) and the vignette: `vignette("external")`

Examples

```
hook <- fetch_hook_read(
    function(key, namespace) paste0(key, ".csv"),
    function(filename) read.csv(filename, stringsAsFactors = FALSE))
```

join_key_namespace *Recycle key and namespace*

Description

Utility function for driver authors

Usage

```
join_key_namespace(key, namespace)
```

Arguments

key	A vector of keys
namespace	A vector of namespace

Details

This exists to join, predictably, keys and namespaces for operations like `mget`. Given a vector or scalar for `key` and `namespace` we work out what the required length is and recycle `key` and `namespace` to the appropriate length.

Value

A list with elements `n`, `key` and `namespace`

storr *Object cache*

Description

Create an object cache; a "storr". A storr is a simple key-value store where the actual content is stored in a content-addressible way (so that duplicate objects are only stored once) and with a caching layer so that repeated lookups are fast even if the underlying storage driver is slow.

Usage

```
storr(driver, default_namespace = "objects")
```

Arguments

driver	A driver object
default_namespace	Default namespace to store objects in. By default `"objects"` is used, but this might be useful to have two diffent `storr` objects pointing at the same underlying storage, but storing things in different namespaces.

Details

To create a storr you need to provide a "driver" object. There are three in this package: `driver_environment` for ephemeral in-memory storage, `driver_rds` for serialized storage to disk, and `driver_dbi` for use with DBI-compliant database interfaces. The `redux` package (on CRAN) provides a storr driver that uses Redis.

There are convenience functions (e.g., `storr_environment` and `storr_rds`) that may be more convenient to use than this function.

Once a storr has been made it provides a number of methods. Because storr uses `R6` (`R6Class`) objects, each method is accessed by using `$` on a storr object (see the examples). The methods are described below in the "Methods" section.

The `default_namespace` affects all methods of the storr object that refer to namespaces; if a namespace is not given, then the action (get, set, del, list, import, export) will affect the `default_namespace`. By default this is `"objects"`.

Methods

`destroy` Totally destroys the storr by telling the driver to destroy all the data and then deleting the driver. This will remove all data and cannot be undone.

Usage: `destroy()`

`flush_cache` Flush the temporary cache of objects that accumulates as the storr is used. Should not need to be called often.

Usage: `flush_cache()`

`set` Set a key to a value.

Usage: `set(key, value, namespace = self$default_namespace, use_cache = TRUE)`

Arguments:

- `key`: The key name. Can be any string.
- `value`: Any R object to store. The object will generally be serialized (this is not actually true for the environment storr) so only objects that would usually be expected to survive a `saveRDS`/`readRDS` roundtrip will work. This excludes Rcpp modules objects, external pointers, etc. But any "normal" R object will work fine.
- `namespace`: An optional namespace. By default the default namespace that the storr was created with will be used (by default that is "objects"). Different namespaces allow different types of objects to be stored without risk of names colliding. Use of namespaces is optional, but if used they must be a string.
- `use_cache`: Use the internal cache to avoid reading or writing to the underlying storage if the data has already been seen (i.e., we have seen the hash of the object before).

Value: Invisibly, the hash of the saved object.

`set_by_value` Like `set` but saves the object with a key that is the same as the hash of the object. Equivalent to `$set(digest::digest(value), value)`.

Usage: `set_by_value(value, namespace = self$default_namespace, use_cache = TRUE)`

Arguments:

- `value`: An R object to save, with the same limitations as `set`.
- `namespace`: Optional namespace to save the key into.

- use_cache: Use the internal cache to avoid reading or writing to the underlying storage if the data has already been seen (i.e., we have seen the hash of the object before).

`get` Retrieve an object from the storr. If the requested value is not found then a `KeyError` will be raised (an R error, but can be caught with `tryCatch`; see the "storr" vignette).

Usage: `get(key, namespace = self$default_namespace, use_cache = TRUE)`

Arguments:

- key: The name of the key to get.
- namespace: Optional namespace to look for the key within.
- use_cache: Use the internal cache to avoid reading or writing to the underlying storage if the data has already been seen (i.e., we have seen the hash of the object before).

`get_hash` Retrieve the hash of an object stored in the storr (rather than the object itself).

Usage: `get_hash(key, namespace = self$default_namespace)`

Arguments:

- key: The name of the key to get.
- namespace: Optional namespace to look for the key within.

`del` Delete an object fom the storr.

Usage: `del(key, namespace = self$default_namespace)`

Arguments:

- key: A vector of names of keys
- namespace: The namespace of the key.

Value: A logical vector the same length as the recycled length of key/namespace, with each element being `TRUE` if an object was deleted, `FALSE` otherwise.

`duplicate` Duplicate the value of a set of keys into a second set of keys. Because the value stored against a key is just the hash of its content, this operation is very efficient - it does not make a copy of the data, just the pointer to the data (for more details see the storr vignette which explains the storage model in more detail). Multiple keys (and/or namespaces) can be provided, with keys and nmespaces recycled as needed. However, the number of source and destination keys must be the same. The order of operation is not defined, so if the sets of keys are overlapping it is undefined behaviour.

Usage: `duplicate(key_src, key_dest, namespace = self$default_namespace, namespace_src = names`

Arguments:

- key_src: The source key (or vector of keys)
- key_dest: The destination key
- namespace: The namespace to copy keys within (used only of `namespace_src` and `namespace_dest` are not provided
- namespace_src: The source namespace - use this where keys are duplicated across namespaces.
- namespace_dest: The destination namespace - use this where keys are duplicated across namespaces.

`fill` Set one or more keys (potentially across namespaces) to the same value, without duplication effort serialisation, or duplicating data.

Usage: `fill(key, value, namespace = self$default_namespace, use_cache = TRUE)`

Arguments:

- key: A vector of keys to get; zero to many valid keys
- value: A single value to set all keys to
- namespace: A vector of namespaces (either a single namespace or a vector)
- use_cache: Use the internal cache to avoid reading or writing to the underlying storage if the data has already been seen (i.e., we have seen the hash of the object before).

clear Clear a storr. This function might be slow as it will iterate over each key. Future versions of storr might allow drivers to implement a bulk clear method that will allow faster clearing.

Usage: `clear(namespace = self$default_namespace)`

Arguments:

- namespace: A namespace, to clear a single namespace, or NULL to clear all namespaces.

exists Test if a key exists within a namespace

Usage: `exists(key, namespace = self$default_namespace)`

Arguments:

- key: A vector of names of keys
- namespace: The namespace of the key.

Value: A logical vector the same length as the recycled length of key/namespace, with each element being TRUE if the object exists and FALSE otherwise.

exists_object Test if an object with a given hash exists within the storr

Usage: `exists_object(hash)`

Arguments:

- hash: Hash to test

mset Set multiple elements at once

Usage: `mset(key, value, namespace = self$default_namespace, use_cache = TRUE)`

Arguments:

- key: A vector of keys to set; zero to many valid keys
- value: A vector of values
- namespace: A vector of namespaces (either a single namespace or a vector)
- use_cache: Use the internal cache to avoid reading or writing to the underlying storage if the data has already been seen (i.e., we have seen the hash of the object before).

Details: The arguments key and namespace are recycled such that either can be given as a scalar if the other is a vector. Other recycling is not allowed.

mget Get multiple elements at once

Usage: `mget(key, namespace = self$default_namespace, use_cache = TRUE, missing = NULL)`

Arguments:

- key: A vector of keys to get; zero to many valid keys
- namespace: A vector of namespaces (either a single namespace or a vector)
- use_cache: Use the internal cache to avoid reading or writing to the underlying storage if the data has already been seen (i.e., we have seen the hash of the object before).
- missing: Value to use for missing elements; by default NULL will be used. IF NULL is a value that you might have stored in the storr you might want to use a different value here to distinguish "missing" from "set to NULL". In addition, the missing attribute will indicate which values were missing.

Details: The arguments `key` and `namespace` are recycled such that either can be given as a scalar if the other is a vector. Other recycling is not allowed.

Value: A list with a length of the recycled length of `key` and `namespace`. If any elements are missing, then an attribute `missing` will indicate the elements that are missing (this will be an integer vector with the indices of values were not found in the storr).

`mset_by_value` Set multiple elements at once, by value. A cross between `mset` and `set_by_value`.

Usage: `mset_by_value(value, namespace = self$default_namespace, use_cache = TRUE)`

Arguments:

- `value`: A list or vector of values to set into the storr.
- `namespace`: A vector of namespaces
- `use_cache`: Use the internal cache to avoid reading or writing to the underlying storage if the data has already been seen (i.e., we have seen the hash of the object before).

`gc` Garbage collect the storr. Because keys do not directly map to objects, but instead map to hashes which map to objects, it is possible that hash/object pairs can persist with nothing pointing at them. Running `gc` will remove these objects from the storr.

Usage: `gc()`

`get_value` Get the content of an object given its hash.

Usage: `get_value(hash, use_cache = TRUE)`

Arguments:

- `hash`: The hash of the object to retrieve.
- `use_cache`: Use the internal cache to avoid reading or writing to the underlying storage if the data has already been seen (i.e., we have seen the hash of the object before).

Value: The object if it is present, otherwise throw a `HashError`.

`set_value` Add an object value, but don't add a key. You will not need to use this very often, but it is used internally.

Usage: `set_value(value, use_cache = TRUE)`

Arguments:

- `value`: An R object to set.
- `use_cache`: Use the internal cache to avoid reading or writing to the underlying storage if the data has already been seen (i.e., we have seen the hash of the object before).

Value: Invisibly, the hash of the object.

`mset_value` Add a vector of object values, but don't add keys. You will not need to use this very often, but it is used internally.

Usage: `mset_value(values, use_cache = TRUE)`

Arguments:

- `values`: A list of R objects to set
- `use_cache`: Use the internal cache to avoid reading or writing to the underlying storage if the data has already been seen (i.e., we have seen the hash of the object before).

`list` List all keys stored in a namespace.

Usage: `list(namespace = self$default_namespace)`

Arguments:

- `namespace`: The namespace to list keys within.

Value: A sorted character vector (possibly zero-length).

`list_hashes` List all hashes stored in the storr

Usage: `list_hashes()`

Value: A sorted character vector (possibly zero-length).

`list_namespaces` List all namespaces known to the database

Usage: `list_namespaces()`

Value: A sorted character vector (possibly zero-length).

`import` Import R objects from an environment.

Usage: `import(src, list = NULL, namespace = self$default_namespace, skip_missing = FALSE)`

Arguments:

- `src`: Object to import objects from; can be a list, environment or another storr.
- `list`: Names of of objects to import (or `NULL` to import all objects in `envir`. If given it must be a character vector. If named, the names of the character vector will be the names of the objects as created in the storr.
- `namespace`: Namespace to get objects from, and to put objects into. If `NULL`, all namespaces from `src` will be imported. If named, then the same rule is followed as `list`; `namespace = c(a = b)` will import the contents of namespace `b` as namespace `a`.
- `skip_missing`: Logical, indicating if missing keys (specified in `list`) should be skipped over, rather than being treated as an error (the default).

`export` Export objects from the storr into something else.

Usage: `export(dest, list = NULL, namespace = self$default_namespace, skip_missing = FALSE)`

Arguments:

- `dest`: A target destination to export objects to; can be a list, environment, or another storr. Use `list()` to export to a brand new list, or use `as.list(object)` for a shorthand.
- `list`: Names of objects to export, with the same rules as `list` in `$import`.
- `namespace`: Namespace to get objects from, and to put objects into. If `NULL`, then this will export namespaces from this (source) storr into the destination; if there is more than one namespace,this is only possible if `dest` is a storr (otherwise there will be an error).
- `skip_missing`: Logical, indicating if missing keys (specified in `list`) should be skipped over, rather than being treated as an error (the default).

Value: Invisibly, `dest`, which allows use of `e <- st$export(new.env())` and `x <- st$export(list())`.

`archive_export` Export objects from the storr into a special "archive" storr, which is an `storr_rds` with name mangling turned on (which encodes keys with base64 so that they do not voilate filesystem naming conventions).

Usage: `archive_export(path, names = NULL, namespace = NULL)`

Arguments:

- `path`: Path to create the storr at; can exist already.
- `names`: As for `$export`
- `namespace`: Namespace to get objects from. If `NULL`, then exports all namespaces found in this (source) storr.

`archive_import` Inverse of `archive_export`; import objects from a storr that was created by `archive_export`.

Usage: `archive_import(path, names = NULL, namespace = NULL)`

Arguments:

- path: Path of the exported storr.
- names: As for $import
- namespace: Namespace to import objects into. If NULL, then imports all namespaces from the source storr.

index_export Generate a data.frame with an index of objects present in a storr. This can be saved (for an rds storr) in lieu of the keys/ directory and re-imported with index_import. It will provide a more version control friendly export of the data in a storr.

Usage: index_export(namespace = NULL)

Arguments:

- namespace: Optional character vector of namespaces to export. The default is to export all namespaces.

index_import Import an index.

Usage: index_import(index)

Arguments:

- index: Must be a data.frame with columns 'namespace', 'key' and 'hash' (in any order). It is an error if not all hashes are present in the storr.

Examples

```
st <- storr(driver_environment())
## Set "mykey" to hold the mtcars dataset:
st$set("mykey", mtcars)
## and get the object:
st$get("mykey")
## List known keys:
st$list()
## List hashes
st$list_hashes()
## List keys in another namespace:
st$list("namespace2")
## We can store things in other namespaces:
st$set("x", mtcars, "namespace2")
st$set("y", mtcars, "namespace2")
st$list("namespace2")
## Duplicate data do not cause duplicate storage: despite having three
## keys we only have one bit of data:
st$list_hashes()
st$del("mykey")

## Storr objects can be created that have a default namespace that is
## not "objects" by using the \code{default_namespace} argument (this
## one also points at the same memory as the first storr).
st2 <- storr(driver_environment(st$driver$envir),
             default_namespace = "namespace2")
## All functions now use "namespace2" as the default namespace:
st2$list()
st2$del("x")
st2$del("y")
```

`storr_dbi` *DBI storr driver*

Description

Object cache driver using the "DBI" package interface for storage. This means that storr can work for any supported "DBI" driver (though practically this works only for SQLite and Postgres until some MySQL dialect translation is done). To connect, you must provide the *driver* object (e.g., `RSQLite::SQLite()`, or `RPostgres::Postgres()` as the first argument.

Usage

```
storr_dbi(tbl_data, tbl_keys, con, args = NULL, binary = NULL,
  hash_algorithm = NULL, default_namespace = "objects")

driver_dbi(tbl_data, tbl_keys, con, args = NULL, binary = NULL,
  hash_algorithm = NULL)
```

Arguments

`tbl_data`	Name for the table that maps hashes to values
`tbl_keys`	Name for the table that maps keys to hashes
`con`	Either A DBI connection or a DBI driver (see example)
`args`	Arguments to pass, along with the driver, to `DBI::dbConnect` if `con` is a driver.
`binary`	Optional logical indicating if the values should be stored in binary. If possible, this is both (potentially faster) and more accurate. However, at present it is supported only under very recent DBI and RSQLite packages, and for no other DBI drivers, and is not actually any faster. If not given (i.e., `NULL`), then binary storage will be used where possible when creating new tables, and where tables exist, we use whatever was used in the existing tables.
`hash_algorithm`	Name of the hash algorithm to use. Possible values are "md5", "sha1", and others supported by `digest`. If not given, then we will default to "md5".
`default_namespace`	Default namespace (see `storr`).

Details

Because the DBI package specifies a uniform interface for the using DBI compliant databases, you need only to provide a connection object. storr does not do anything to help create the connection object itself.

The DBI storr driver works by using two tables; one mapping keys to hashes, and one mapping hashes to values. Two table names need to be provided here; they must be different and they should be treated as opaque (don't use them for anything else - reading or writing). Apart from that the names do not matter.

Because of treatment of binary data by the underlying DBI drivers, binary serialistion is not any faster (and might be slightly slower than) string serialisation, in contrast with my experience with other backends.

storr uses DBI's "prepared query" approach to safely interpolate keys, namespaces and values into the database - this should allow odd characters without throwing SQL syntax errors. Table names can't be interpolated in the same way - these storr simply quotes, but validates beforehand to ensure that `tbl_data` and `tbl_keys` do not contain quotes.

Be aware that `$destroy()` will close the connection to the database.

Examples

```
if (requireNamespace("RSQLite", quietly = TRUE)) {
  st <- storr::storr_dbi("tblData", "tblKeys", RSQLite::SQLite(),
                         ":memory:")

  # Set some data:
  st$set("foo", runif(10))
  st$list()

  # And retrieve the data:
  st$get("foo")

  # These are the data tables; treat these as read only
  DBI::dbListTables(st$driver$con)

  # With recent RSQLite you'll get binary storage here:
  st$driver$binary

  # The entire storr part of the database can be removed using
  # "destroy"; this will also close the connection to the database
  st$destroy()

  # If you have a connection you want to reuse (which will the the
  # case if you are using an in-memory SQLite database for
  # multiple things within an application) it may be useful to
  # pass the connection object instead of the driver:
  con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  st <- storr::storr_dbi("tblData", "tblKeys", con)
  st$set("foo", runif(10))

  # You can then connect a different storr to the same underlying
  # storage
  st2 <- storr::storr_dbi("tblData", "tblKeys", con)
  st2$get("foo")
}
```

`storr_environment` *Environment object cache driver*

Description

Fast but transient environment driver. This driver saves objects in a local R environment, without serialisation. This makes lookup fast but it cannot be saved across sesssions. The environment storr can be made persistent by saving it out as a file storr though.

Usage

```
storr_environment(envir = NULL, hash_algorithm = NULL,
  default_namespace = "objects")

driver_environment(envir = NULL, hash_algorithm = NULL)
```

Arguments

envir	The environment to point the storr at. The default creates an new empty environment which is generally the right choice. However, if you want multiple environment storrs pointing at the same environment then pass the envir argument along.
hash_algorithm	Name of the hash algorithm to use. Possible values are "md5", "sha1", and others supported by digest. If not given, then we will default to "md5".
default_namespace	Default namespace (see storr).

Examples

```
# Create an environment and stick some random numbers into it:
st <- storr_environment()
st$set("foo", runif(10))
st$get("foo")

# To make this environment persistent we can save it to disk:
path <- tempfile()
st2 <- st$archive_export(path)
# st2 is now a storr_rds (see ?storr_rds), and will persist across
# sessions.

# or export to a new list:
lis <- st$export(list())
lis
```

---

storr_external *Storr that looks for external resources*

Description

storr for fetching external resources. This driver is used where will try to fetch from an external data source if a resource can not be found locally. This works by checking to see if a key is present in the storr (and if so returning it). If it is not found, then the function fetch_hook is run to fetch it.

Usage

```
storr_external(storage_driver, fetch_hook, default_namespace = "objects")
```

Arguments

storage_driver	Another storr driver to handle the actual storage.
fetch_hook	A function to run to fetch data when a key is not found in the store. This function must take arguments key and namespace and return an R object. It must throw an error if the external resource cannot be resolved.
default_namespace	Default namespace (see storr)

Details

See the vignette vignette("external") for much more detail. This function is likely most useful for things like caching resources from websites, or computing long-running quantities on demand.

---

storr_multistorr *Storr with multiple storage drivers*

Description

Create a special storr that uses separate storage drivers for the keys (which tend to be numerous and small in size) and the data (which tends to be somewhat less numerous and much larger in size). This might be useful to use storage models with different characteristics (in memory/on disk, etc).

Usage

```
storr_multistorr(keys, data, default_namespace = "objects")
```

Arguments

keys	Driver for the keys
data	Driver for the data
default_namespace	Default namespace (see storr).

Details

This is an experimental feature and somewhat subject to change. In particular, the driver may develop the ability to store small data in the same storr as the keys (say, up to 1kb) based on some tunable parameter.

You can attach another storr to either the data or the key storage (see the example), but it will not be able to see keys or data (respectively). If you garbage collect the data half, all the data will be lost!

Examples

```
# Create a storr that is stores keys in an environment and data in
# an rds
path <- tempfile()
st <- storr::storr_multistorr(driver_environment(),
                              driver_rds(path))
st$set("a", runif(10))
st$get("a")

# The data can be also seen by connecting to the rds store
rds <- storr::storr_rds(path)
rds$list() # empty
rds$list_hashes() # here's the data
rds$get_value(rds$list_hashes())

st$destroy()
```

`storr_rds` *rds object cache driver*

Description

Object cache driver that saves objects using R's native serialized file format (see `saveRDS`) on the filesystem.

Usage

```
storr_rds(path, compress = NULL, mangle_key = NULL, mangle_key_pad = NULL,
  hash_algorithm = NULL, default_namespace = "objects")

driver_rds(path, compress = NULL, mangle_key = NULL,
  mangle_key_pad = NULL, hash_algorithm = NULL)
```

Arguments

`path`	Path for the store. `tempdir()` is a good choice for ephemeral storage, The `rappdirs` package (on CRAN) might be nice for persistent application data.
`compress`	Compress the generated file? This saves a small amount of space for a reasonable amount of time.
`mangle_key`	Mangle keys? If TRUE, then the key is encoded using base64 before saving to the filesystem. See Details.
`mangle_key_pad`	Logical indicating if the filenames created when using `mangle_key` should also be "padded" with the = character to make up a round number of bytes. Padding is required to satisfy the document that describes base64 encoding (RFC 4648) but can cause problems in some applications (see this issue. The default is to not pad *new* storr archives. This should be generally safe to leave alone.

`hash_algorithm` Name of the hash algorithm to use. Possible values are "md5", "sha1", and others supported by `digest`. If not given, then we will default to "md5".

`default_namespace`
Default namespace (see `storr`).

Details

The `mangle_key` argument will run each key that is created through a "base 64" encoding. This means that keys that include symbols that are invalid on filesystems (e.g, "/", ":") will be replaced by harmless characters. The RFC 4648 dialect is used where "-" and "_" are used for character 62 and 63 (this differs from most R base64 encoders). This mangling is designed to be transparent to the user – the storr will appear to store things with unmangled keys but the names of the stored files will be different.

Note that the *namespace* is not mangled (at least not yet) so needs to contain characters that are valid in a filename.

Because the actual file will be stored with mangled names it is not safe to use the same path for a storr with and without mangling. So once an rds storr has been created its "mangledness" is set. Using `mangle_key = NULL` uses whatever mangledness exists (or no mangledness if creating a new storr).

Corrupt keys

Some file synchronisation utilities like dropbox can create file that confuse an rds storr (e.g., `"myobject (Someone's conflicted copy)"`. If `mangle_key` is `FALSE` these cannot be detected but at the same time are not a real problem for storr. However, if `mangle_key` is `TRUE` and keys are base64 encoded then these conflicted copies can break parts of storr.

If you see a warning asking you to deal with these files, please delete the offending files; the path will be printed along with the files that are causing the problem.

Alternatively, you can try (assuming a storr object `st`) running

```
st$driver$purge_corrupt_keys()
```

which will delete corrupted keys with no confirmation. The messages that are printed to screen will be printed by default at most once per minute per namespace. You can control this by setting the R option `storr.corrupt.notice.period` - setting this to `NA` suppresses the notice and otherwise it is interpreted as the number of seconds.

Examples

```
# Create an rds storr in R's temporary directory:
st <- storr_rds(tempfile())

# Store some data (10 random numbers against the key "foo")
st$set("foo", runif(10))
st$list()

# And retrieve the data:
```

```
st$get("foo")

# Keys that are not valid filenames will cause issues.  This will
# cause an error:
## Not run:
st$set("foo/bar", letters)

## End(Not run)

# The solution to this is to "mangle" the key names.  Storr can do
# this for you:
st2 <- storr_rds(tempfile(), mangle_key = TRUE)
st2$set("foo/bar", letters)
st2$list()
st2$get("foo/bar")

# Behind the scenes, storr is safely encoding the filenames with base64:
dir(file.path(st2$driver$path, "keys", "objects"))

# Clean up the two storrs:
st$destroy()
st2$destroy()
```

`test_driver` *Test a storr driver*

Description

Test that a driver passes all storr tests. This page is only of interest to people developing storr drivers; nothing here is required for using storr.

Usage

```
test_driver(create)
```

Arguments

`create`	A function with one arguments that when run with `NULL` as the argument will create a new driver instance. It will also be called with a driver (of the same type) as an argument - in this case, you must create a new driver object pointing at the same underlying storage (see the examples). Depending on your storage model, temporary directories, in-memory locations, or random-but-unique prefixes may help create isolated locations for the test (the tests assume that a storr created with `create` is entirely empty).

Details

This will run through a suite of functions to test that a driver is likely to behave itself. As bugs are found they will be added to the test suite to guard against regressions.

The test suite is included in the package as `system.file("spec", package = "storr")`.

The procedure for each test block is:

1. Create a new driver by running `dr <- create()`.
2. Run a number of tests.
3. Destroy the driver by running `dr$destroy()`.

So before running this test suite, make sure this will not harm any precious data!

Examples

```
## Testing the environment driver is nice and fast:
if (requireNamespace("testthat")) {
  create_env <- function(dr = NULL, ...) {
    driver_environment(dr$envir, ...)
  }
  test_driver(create_env)
}

# To test things like the rds driver, I would run:
## Not run:
if (requireNamespace("testthat")) {
  create_rds <- function(dr = NULL) {
    driver_rds(if (is.null(dr)) tempfile() else dr$path)
  }
  test_driver(create_rds)
}

## End(Not run)
```

Index