Package ‘dittodb’

April 9, 2024

Title A Test Environment for Database Requests
Version 0.1.8
BugReports https://github.com/ropensci/dittodb/issues

Description Testing and documenting code that communicates with remote databases can be painful. Although the interaction with R is usually relatively simple (e.g. data(frames) passed to and from a database), because they rely on a separate service and the data there, testing them can be difficult to set up, unsustainable in a continuous integration environment, or impossible without replicating an entire production cluster. This package addresses that by allowing you to make recordings from your database interactions and then play them back while testing (or in other contexts) all without needing to spin up or have access to the database your code would typically connect to.

License Apache License (>= 2.0)
Encoding UTF-8
Depends R (>= 3.3.0), DBI
Imports digest, glue, methods, rlang, utils, lifecycle
Suggests bit64, callr, covr, dplyr, dbplyr, knitr, nycflights13, odbc, RMariaDB, RPostgres, RPostgreSQL, RSQLite, spelling, testthat, withr, rmarkdown

RoxygenNote 7.3.1
Language en-US
VignetteBuilder knitr
Config/testthat/edition 3


RdMacros lifecycle
capture_requests

Description

When creating database fixtures, it can sometimes be helpful to record the responses from the database for use in crafting tests.

Usage

start_db_capturing(path, redact_columns = NULL)

stop_db_capturing()

capture_db_requests(expr, path, redact_columns = NULL)
Arguments

- **path**: the path to record mocks (default if missing: the first path in `db_mock_paths()`).
- **redact_columns**: a character vector of columns to redact. Any column that matches an entry will be redacted with a standard value for the column type (e.g. characters will be replaced with "[redacted]"
- **expr**: an expression to evaluate while capturing requests (for `capture_db_requests()`)
con <- dbConnect(RSQLite::SQLite(), nycflights_path)
df_3 <- dbGetQuery(con, "SELECT * FROM airlines LIMIT 3")

dbDisconnect(con)
stop_db_capturing()

with_mock_db(
  con <- dbConnect(RSQLite::SQLite(), nycflights_path)

  # the result from df1 above
  print(dbGetQuery(con, "SELECT * FROM airlines LIMIT 1"))

  # the result from df3 above
  print(dbGetQuery(con, "SELECT * FROM airlines LIMIT 3"))
)
}

---

**expect_sql**

*Detect if a specific SQL statement is sent*

**Description**

[Experimental]

**Usage**

```r
expect_sql(object, regexp = NULL, ...)
```

**Arguments**

- `object` the expression to evaluate
- `regexp` the statement to match
- `...` arguments passed to `testthat::expect_error()`

**Details**

Sometimes all you need to check is if a specific SQL statement has been sent and you don’t care about retrieving the results.

This works by raising an error that contains the statement that is sent to the database as well as the location of the result. Currently, expect_sql() only works with `DBI::dbSendQuery()` (and most implementations of `DBI::dbGetQuery()` which call `DBI::dbSendQuery()` internally).

*Note:* this function is experimental and will likely evolve over time. Please be prepared that new releases might break backwards compatibility.
Examples

```r
if (check_for_pkg("RSQLite", message)) {
  with_mock_db({
    con <- dbConnect(RSQLite::SQLite(), dbname = "not_a_db")
    expect_sql(
      dbGetQuery(con, "SELECT carrier, name FROM airlines LIMIT 3"),
      "SELECT carrier, name FROM airlines LIMIT 3"
    )
  })
}
```

**Description**

Various methods (`dbSendQuery`, `dbFetchQuery`) that are mocks of the DBI methods of the same name. Instead of actually interacting with a database, they read in mock responses and the code proceeds after that. These aren’t used directly, but are part of how dttodb works.

**Usage**

```r
## S4 method for signature 'DBIMockConnection'
dbDisconnect(conn, ...)

dbMockConnect(drv, ...)

## S4 method for signature 'DBIMockConnection,character'
dbExistsTable(conn, name, ...)

## S4 method for signature 'DBIMockConnection,Id'
dbExistsTable(conn, name, ...)

## S4 method for signature 'DBIMockConnection'
dbListTables(conn, ...)

## S4 method for signature 'DBIMockConnection,character'
dbListFields(conn, name, ...)

## S4 method for signature 'DBIMockConnection,Id'
dbListFields(conn, name, ...)

## S4 method for signature 'DBIMockConnection,ANY'
dbListFields(conn, name, ...)

## S4 method for signature 'DBIMockConnection,character'
```
dbSendQuery(conn, statement, ...)  
## S4 method for signature 'DBIMockConnection,SQL'

dbSendQuery(conn, statement, ...)  
## S4 method for signature 'DBIMockConnection,character'

dbSendStatement(conn, statement, ...)  
## S4 method for signature 'DBIMockConnection,character'

dbFetch(res, n = -1, ...)  
## S4 method for signature 'DBIMockResult,ANY'

fetch(res, n = -1, ...)  
## S4 method for signature 'DBIMockResult,missing'

fetch(res, n = -1, ...)  
## S4 method for signature 'DBIMockResult'

dbClearResult(res, n, ...)  
## S4 method for signature 'DBIMockResult'

dbHasCompleted(res, ...)  
## S4 method for signature 'DBIMockResult'

dbGetQuery(conn, statement, ...)  
## S4 method for signature 'DBIMockRPostgreSQLConnection,character'

dbGetRowsAffected(res, ...)  
## S4 method for signature 'DBIMockResult'

dbNameInfo(dbObj, ...)  
## S4 method for signature 'DBIMockConnection'

dbNameInfo(dbObj, ...)  
## S4 method for signature 'DBIMockResult'

dbNameInfo(dbObj, ...)  
## S4 method for signature 'DBIMockResult'

dbQuoteIdentifier(conn, x, ...)  
## S4 method for signature 'DBIMockRPostgresConnection,character'

### S4 method for signature 'DBIMockRPostgresConnection,SQL'
dbQuoteIdentifier(conn, x, ...)
## S4 method for signature 'DBIMockRPostgresConnection,character'
## S4 method for signature 'DBIMockRPostgresConnection,SQL'
## S4 method for signature 'DBIMockMariaDBConnection,character'
## S4 method for signature 'DBIMockMariaDBConnection,SQL'
## S4 method for signature 'DBIMockConnection'
## S4 method for signature 'DBIMockConnection'
## S4 method for signature 'DBIMockConnection'

Arguments

- **conn**: a database connection (for dispatch with these methods, it should be of class DBIMockConnection)
- **...**: arguments passed on inside of the methods
- **drv**: a DB driver for use in dbConnect
- **name**: name of the table (for dbListFields, dbWriteTable, dbRemoveTable)
- **statement**: an SQL statement to execute
- **res**: a result object (for dispatch with these methods, it should be of class DBIMockResult)
- **n**: number of results to fetch (ignored)
- **dbObj**: a database object (a connection, result, etc.) for use in dbGetInfo
- **value**: a value (generally a data.frame) for use in dbWriteTable
- **x**: a name to quote (for dbQuoteIdentifier)

mockdb

Run DBI queries against a mocked database

Description

Wrap a chunk of code in with_mock_db() to use mocked databases that will use fixtures instead of connecting to a real database. Alternatively, you can start and stop using a mocked database with start_mock_db() and stop_mock_db() respectively to execute the whole thing without needing to remember to stop the mocking. When testing with dittodb, it will look for fixtures in all entries of db_mock_paths.
Usage

with_mock_db(expr)

start_mock_db()

stop_mock_db()

Arguments

expr the expression to execute

Details

You only need to use one approach: either use `start_mock_db()` to start using mocks and then `stop_mock_db()` to stop or use `with_mock_db()` wrapped around the code you want to execute against the mocked database. You don’t need to (and should not) use both at the same time. Generally `with_mock_db()` is preferred because it is slightly safer and you don’t have to remember to `stop_mock_db()` when you’re done. However, it is easier to step through tests interactively using `start_mock_db()/stop_mock_db()`.

Connections should be made after `start_mock_db()` if you’re using that function or they should be made inside of `with_mock_db()` if you’re using that function because dittodb uses the database name (given in `dbname` or `Database` argument of `dbConnect` depending on the driver) to separate different fixtures. For ODBC connections with only a dsn provided, the dsn is used for this directory.

Value

nothing

Examples

# Add the mocks included with dittodb to the db_mock_paths to use them below
db_mock_paths(system.file("nycflight_mocks", package = "dittodb"), last = TRUE)

if (check_for_pck("RSQLite", message) & check_for_pck("testthat", message)) {
  # using 'with_mock_db()'
  with_mock_db(
    con <- dbConnect(
      RSQLite::SQLite(),
      dbname = "nycflights"
    )
  )
  testthat::test_that("We get one airline", {
    one_airline <- dbGetQuery(
      con,
      "SELECT carrier, name FROM airlines LIMIT 1"
    )
    testthat::expect_s3_class(one_airline, "data.frame")
    testthat::expect_equal(nrow(one_airline), 1)
    testthat::expect_equal(one_airline$carrier, "9E")
    testthat::expect_equal(one_airline$name, "Endeavor Air Inc.")
  })
}
nycflights13_create_sql

Create a standardised database for testing

Description

Using the connection given in con, create a database including a few tables from the nycflights13 dataset.

Usage

nycflights13_create_sql(con, schema = "", ...)

Arguments

- **con**: an SQL connection (i.e., a PostgreSQL connection)
- **schema**: schema to write the tables ("", or no schema by default)
- **...**: additional parameters to connect to a database

Value

the connection given in con invisibly, generally called for the side effects of writing to the database
Examples

if (check_for_pkg("RSQLite", message)) {
  con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:"
)

  nycflights13_create_sql(con)

  DBI::dbGetQuery(
    con,
    "SELECT year, month, day, carrier, flight, tailnum FROM flights LIMIT 10"
  )

  DBI::dbDisconnect(con)
}

nycflights13_create_sqlite

Create an in-memory SQLite database for testing

Description

Create an in-memory SQLite database for testing

Usage

nycflights13_create_sqlite(location = ":memory:", ...)

Arguments

location where to store the database
... additional parameters to connect to a database (most are passed on to nycflights13_create_sql)

Value

RSQLiteConnection

Examples

if (check_for_pkg("RSQLite", message)) {
  con <- nycflights13_create_sqlite()

  DBI::dbGetQuery(
    con,
    "SELECT year, month, day, carrier, flight, tailnum FROM flights LIMIT 10"
  )

  DBI::dbDisconnect(con)
An SQLite connection to a subset of nycflights13

**Description**

Included with dittodb is a small subset of nycflights13 prepopulated into a sqlite database.

**Usage**

```
nycflights_sqlite()
```

**Details**

This database is helpful for getting to know dittodb and running example code. It contains a small subset of the data in nycflights13: namely only the flights and planes that had a destination of ORD or MDW (the codes for the two major airports in Chicago) in February of 2013. The airports table has also been limited to only the New York and Chicago area airports.

**Value**

an RSQLiteConnection

**Examples**

```
if (check_for_pkg("RSQLite", message)) {
  con <- nycflights_sqlite()

  DBI::dbGetQuery(con, "SELECT flight, tailnum, origin, dest FROM flights LIMIT 10")
  DBI::dbGetQuery(con, "SELECT faa, name, lat, lon, alt, tz FROM airports")

  DBI::dbDisconnect(con)
}
```

---

**redact_columns**

Redact columns from a dataframe with the default redactors

**Description**

This function redacts the columns specified in columns in the data given in data using dittodb’s standard redactors.

**Usage**

```
redact_columns(data, columns, ignore.case = TRUE, ...)
```
Arguments

- **data**: a dataframe to redact
- **columns**: character, the columns to redact
- **ignore.case**: should case be ignored? (default: TRUE)
- **...**: additional options to pass on to `grep()` when matching the column names

Details

The column names given in the `columns` argument are treated as regular expressions, however they always have ^ and $ added to the beginning and end of the strings. So if you would like to match any column that starts with the string `sensitive` (e.g. `sensitive_name, sensitive_date`) you could use "sensitive.*" and this would catch all of those columns (though it would not catch a column called `most_sensitive_name`).

The standard redactors replace all values in the column with the following values based on the columns type:

- integer – 9L
- numeric – 9
- character – "[redacted]"
- `POSIXct` (date times) – `as.POSIXct("1988-10-11T17:00:00", tz = tzone)``

Value

data, with the columns specified in `columns` duly redacted

Examples

```r
if (check_for_pkg("nycflights13", message)) {
  small_flights <- head(nycflights13::flights)

  # with no columns specified, redacting does nothing
  redact_columns(small_flights, columns = NULL)

  # integer
  redact_columns(small_flights, columns = "arr_time")

  # numeric
  redact_columns(small_flights, columns = "arr_delay")

  # characters
  redact_columns(small_flights, columns = c("origin", "dest")

  # datetiems
  redact_columns(small_flights, columns = "time_hour")
}
```
**set_dittodb_debug_level**

*Set dittodb’s debug level*

**Description**

It can be helpful to see what’s going on by increasing dittodb’s verbosity which will show what’s going on under the hood (e.g. what queries are being requested, from where). This sets the option dittodb.debug to the value given in the level argument. The option can be set directly with options(dittodb.debug = n) as well.

**Usage**

```r
set_dittodb_debug_level(level)
```

**Arguments**

- `level` a numeric, the level to set to (e.g. 1)

**Details**

The `level` argument is a numeric, where 0 is the default and (relatively) silent. The higher the level, the more verbose dittodb will be.

Currently, dittodb only has one level of debugging (any value 1 or greater), but more might be used in the future.

**Value**

the level, invisibly

**Examples**

```r
set_dittodb_debug_level(1)
set_dittodb_debug_level(0)
```

---

**use_dittodb**

*Use dittodb in your tests*

**Description**

If you would like to use dittodb in your package, and you are already using testthat, use this function to add dittodb to Suggests in the package DESCRIPTION and loads it in tests/testthat/helper.R. Call it once when you’re setting up a new package test suite.
Usage

use_dittodb(path = ".")

Arguments

path character path to the package

details

This function should be called with the path to your package source as the path argument. The function is idempotent: if dittodb is already added to these files, no additional changes will be made.

It will:

- add dittodb to the Suggests field of the DESCRIPTION file in the current working directory
- add library(dittodb) to the file tests/testthat/helper.R (creating it if it doesn’t already exist)

Value

Nothing: called for file system side effects.

Examples

## Not run:
use_dittodb()
use_dittodb("/path/to/package")

## End(Not run)

Description

When testing with dittodb, wrap your tests in with_mock_path({}) to use the database fixtures located in other directories. dittodb will look for fixtures in the directory specified by the user, which can be a temporary or permanent location.

Usage

with_mock_path(path, expr, replace = FALSE)

Arguments

path the alternate directory
expr the expression to execute
replace logical, should the path replace the current mock paths (TRUE) or should they be appended (to the beginning) of the current mock paths (default, FALSE)
with_mock_path

Value
nothing, called to execute the expression(s) in expr

Examples

# Only run if RSQLite and testthat are available
if (check_for_pkg("RSQLite", message) & check_for_pkg("testthat", message)) {
  with_mock_path(
    system.file("nycflight_mocks", package = "dittodb"),
    with_mock_db({
      con <- DBI::dbConnect(
        RSQLite::SQLite(),
        dbname = "nycflights"
      )
      one_airline <- dbGetQuery(
        con,
        "SELECT carrier, name FROM airlines LIMIT 1"
      )
      testthat::test_that("We get one airline", {
        testthat::expect_s3_class(one_airline, "data.frame")
        testthat::expect_equal(nrow(one_airline), 1)
        testthat::expect_equal(one_airline$carrier, "9E")
        testthat::expect_equal(one_airline$name, "Endeavor Air Inc.")
      })
      one_airline
    })
  )
}
Index

capture_db_requests (capture_requests), 2
capture_requests, 2
db_mock_paths, 7
dbBegin, DBIMockConnection-method (mock-db-methods), 5
dbClearResult, DBIMockResult-method (mock-db-methods), 5
dbColumnInfo, DBIMockResult-method (mock-db-methods), 5
dbCommit, DBIMockConnection-method (mock-db-methods), 5
dbColumnsInfo, DBIMockConnection, ANY-method (mock-db-methods), 5
dbColumnsInfo, DBIMockConnection, character-method (mock-db-methods), 5
dbColumnsInfo, DBIMockConnection, Id-method (mock-db-methods), 5
dbConnect, 7, 8
dbDisconnect, DBIMockConnection-method (mock-db-methods), 5
dbExistsTable, DBIMockConnection, character-method (mock-db-methods), 5
dbExistsTable, DBIMockConnection, character-method (mock-db-methods), 5
dbExistsTable, DBIMockConnection, Id-method (mock-db-methods), 5
dbFetch, DBIMockResult-method (mock-db-methods), 5
dbGetInfo, 7, 9
dbGetInfo, DBIMockConnection-method (mock-db-methods), 5
dbGetInfo, DBIMockResult-method (mock-db-methods), 5
dbGetQuery, DBIMockRPostgreSQLConnection, character-method (mock-db-methods), 5
dbGetRowsAffected, DBIMockResult-method (mock-db-methods), 5
dbHasCompleted, DBIMockResult-method (mock-db-methods), 5
DBI::dbConnect, 3
DBI::dbGetQuery(), 4
DBI::dbSendQuery(), 4
DBIMockConnection-class (mock-db-methods), 5
DBIMockResult-class (mock-db-methods), 5
dbListFields, DBIMockConnection, ANY-method (mock-db-methods), 5
dbListFields, DBIMockConnection, character-method (mock-db-methods), 5
dbListFields, DBIMockConnection, Id-method (mock-db-methods), 5
dbListTables, DBIMockConnection-method (mock-db-methods), 5
dbMockConnect (mock-db-methods), 5
dbQuoteIdentifier, 7
dbQuoteIdentifier, DBIMockRPostgresConnection, character-method (mock-db-methods), 5
dbQuoteIdentifier, DBIMockRPostgresConnection, SQL-method (mock-db-methods), 5
dbQuoteString, DBIMockMariaDBConnection, character-method (mock-db-methods), 5
dbQuoteString, DBIMockMariaDBConnection, SQL-method (mock-db-methods), 5
dbQuoteString, DBIMockRPostgresConnection, character-method (mock-db-methods), 5
dbQuoteString, DBIMockRPostgresConnection, SQL-method (mock-db-methods), 5
dbRemoveTable, 7
dbRemoveTable, DBIMockConnection, character-method (mock-db-methods), 5
dbRollback, DBIMockConnection-method (mock-db-methods), 5
dbSendQuery, DBIMockConnection, character-method (mock-db-methods), 5
dbSendQuery, DBIMockConnection, SQL-method (mock-db-methods), 5
dbSendStatement, DBIMockConnection, character-method (mock-db-methods), 5
dbWriteTable, 7
dbWriteTable, DBIMockConnection, character, data.frame-method (mock-db-methods), 5
expect_sql, 4
fetch, DBIMockResult, ANY-method
(mock-db-methods), 5
fetch, DBIMockResult, missing-method
  (mock-db-methods), 5
fetch, DBIMockResult-method
  (mock-db-methods), 5

mock-db-methods, 5
mockdb, 7

nycflights13_create_sql, 9, 10
nycflights13_create_sqlite, 10
nycflights_sqlite, 11

redact_columns, 3, 11

set_dittodb_debug_level, 13
start_db_capturing(capture_requests), 2
start_mock_db(mockdb), 7
stop_db_capturing(capture_requests), 2
stop_mock_db(mockdb), 7

testthat::expect_error(), 4

use_dittodb, 13

with_mock_db, 3
with_mock_db(mockdb), 7
with_mock_path, 14