Package 'querychat'

September 14, 2026

Title Filter and Query Data Frames in 'shiny' Using an LLM Chat Interface

Version 0.4.0

Description Adds an LLM-powered chatbot to your 'shiny' app, that can turn your users' natural language questions into 'SQL' queries that run against your data, and return the result as a reactive data frame. Use it to drive reactive calculations, visualizations, downloads, and more.

License MIT + file LICENSE

URL https://posit-dev.github.io/querychat/r/, https://posit-dev.github.io/querychat/, https://github.com/posit-dev/querychat

BugReports https://github.com/posit-dev/querychat/issues

Depends R (>= 4.1.0)

Imports bsicons, bslib (>= 0.11.0), cli, coro, DBI, ellmer (>= 0.5.0), htmltools, jsonlite, lifecycle, promises, R6, rlang (>= 1.1.0), S7, shiny (>= 1.14.0), shinychat (>= 0.5.0), utils, whisker, yaml, zip

Suggests dbplyr, dplyr, DT, duckdb, ggsql, knitr, later, nanoparquet, palmerpenguins, pins, rmarkdown, RSQLite, rsvg, testthat (>= 3.0.0), V8, withr

VignetteBuilder knitr

Config/roxygen2/version 8.1.0

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

NeedsCompilation no

Author Garrick Aden-Buie [aut, cre] (ORCID: <https://orcid.org/0000-0002-7111-0077>), Joe Cheng [aut, ccp],

Carson Sievert [aut] (ORCID: <https://orcid.org/0000-0002-4958-2844>),
Posit Software, PBC [cph, fnd]

Maintainer Garrick Aden-Buie <garrick@posit.co>

Repository CRAN

Date/Publication 2026-09-13 22:40:02 UTC

Contents

`DataFrameSource` *Data Frame Source*

Description

A DataSource implementation that wraps a data frame using DuckDB or SQLite for SQL query execution.

Details

This class creates an in-memory database connection and registers the provided data frame as a table. All SQL queries are executed against this database table. See DBISource for the full description of available methods.

By default, DataFrameSource uses the first available engine from duckdb (checked first) or RSQLite. You can explicitly set the `engine` parameter to choose between `"duckdb"` or `"sqlite"`, or set the global option `querychat.DataFrameSource.engine` to choose the default engine for all DataFrameSource instances. At least one of these packages must be installed.

Super classes

`DataSource` -> `DBISource` -> `DataFrameSource`

Methods

Public methods:

- DataFrameSource$new()
- DataFrameSource$register_into()
- DataFrameSource$cleanup()
- DataFrameSource$clone()

`DataFrameSource$new()`: Create a new DataFrameSource

Usage:

```
DataFrameSource$new(
  df,
  table_name,
  engine = getOption("querychat.DataFrameSource.engine", NULL)
)
```

Arguments:

`df` A data frame.

`table_name` Name to use for the table in SQL queries. Must be a valid table name (start with letter, contain only letters, numbers, and underscores)

`engine` Database engine to use: "duckdb" or "sqlite". Set the global option `querychat.DataFrameSource.engine` to specify the default engine for all instances. If NULL (default), uses the first available engine from duckdb or RSQLite (in that order).

Returns: A new DataFrameSource object

`DataFrameSource$register_into()`: Register this data frame in a shared DuckDB connection. Internal hook for joining a shared `DuckDBExecutor`. The caller owns `con` and locks it down once all tables are registered.

Usage:

```
DataFrameSource$register_into(con, table_name = self$table_name)
```

Arguments:

`con` A DuckDB DBI connection, owned by the caller.

`table_name` Name for the table in `con`. Defaults to the source's own table name.

Returns: `NULL` (invisibly)

`DataFrameSource$cleanup()`: Disconnect from the database and shut down the DuckDB instance if used.

Usage:

```
DataFrameSource$cleanup()
```

Returns: NULL (invisibly)

`DataFrameSource$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a data frame source (uses first available: duckdb or sqlite)
df_source <- DataFrameSource$new(mtcars, "mtcars")

# Get database type
df_source$get_db_type()  # Returns "DuckDB" or "SQLite"

# Execute a query
result <- df_source$execute_query("SELECT * FROM mtcars WHERE mpg > 25")

# Explicitly choose an engine
df_sqlite <- DataFrameSource$new(mtcars, "mtcars", engine = "sqlite")

# Clean up when done
df_source$cleanup()
df_sqlite$cleanup()
```

DataSource *Data Source Base Class*

Description

An abstract R6 class defining the interface that custom QueryChat data sources must implement. This class should not be instantiated directly; instead, use one of its concrete implementations like DataFrameSource or DBISource.

Public fields

`table_name` Name of the table to be used in SQL queries

Methods

Public methods:

- `DataSource$get_db_type()`
- `DataSource$get_schema()`
- `DataSource$get_schema_result()`
- `DataSource$execute_query()`
- `DataSource$test_query()`
- `DataSource$get_data()`
- `DataSource$get_data_description()`
- `DataSource$cleanup()`
- `DataSource$clone()`

`DataSource$get_db_type()`: Get the database type

Usage:

```
DataSource$get_db_type()
```

Returns: A string describing the database type (e.g., "DuckDB", "SQLite")

`DataSource$get_schema()`: Get schema information about the table

Usage:

```
DataSource$get_schema(categorical_threshold = 20, table_spec = NULL)
```

Arguments:

`categorical_threshold` Maximum number of unique values for a text column to be considered categorical

Returns: A string containing schema information formatted for LLM prompts

`DataSource$get_schema_result()`:

Usage:

```
DataSource$get_schema_result(categorical_threshold = 20, table_spec = NULL)
```

`DataSource$execute_query()`: Execute a SQL query and return results

Usage:

```
DataSource$execute_query(query)
```

Arguments:

`query` SQL query string to execute

Returns: A data frame containing query results

`DataSource$test_query()`: Test a SQL query by fetching only one row

Usage:

```
DataSource$test_query(query, require_all_columns = FALSE)
```

Arguments:

`query` SQL query string to test

`require_all_columns` If `TRUE`, validates that the result includes all original table columns (default: `FALSE`)

Returns: A data frame containing one row of results (or empty if no matches)

`DataSource$get_data()`: Get the unfiltered data as a data frame

Usage:

```
DataSource$get_data()
```

Returns: A data frame containing all data from the table

`DataSource$get_data_description()`: Get a human-readable data description for the system prompt.
Subclasses may override this to provide metadata-derived descriptions (e.g., pin title/description). The default returns an empty string.

Usage:

```
DataSource$get_data_description()
```

Returns: A string, or empty string if no description is available.

`DataSource$cleanup()`: Release resources this data source created. Only resources querychat opened itself are closed (for example the in-memory DuckDB connection a DataFrameSource creates). Connections passed in by the caller are never closed; their lifecycle stays with the caller.

Usage:

```
DataSource$cleanup()
```

Returns: NULL (invisibly)

`DataSource$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
MyDataSource <- R6::R6Class(
  "MyDataSource",
  inherit = DataSource,
  public = list(
    initialize = function(table_name) {
      self$table_name <- table_name
    }
    # Implement abstract methods here...
  )
)
```

`DBISource` *DBI Source*

Description

A DataSource implementation for DBI database connections (SQLite, PostgreSQL, MySQL, etc.). This class wraps a DBI connection and provides SQL query execution against a single table in the database.

Super class

`DataSource` -> `DBISource`

Active bindings

`conn` The DBI connection backing this source (read-only).

Methods

Public methods:

- DBISource$new()
- DBISource$get_db_type()
- DBISource$get_schema()
- DBISource$get_schema_result()
- DBISource$get_semantic_views_description()
- DBISource$execute_query()
- DBISource$test_query()
- DBISource$get_data()
- DBISource$cleanup()
- DBISource$clone()

`DBISource$new()`: Create a new DBISource

Usage:

```
DBISource$new(conn, table_name)
```

Arguments:

`conn` A DBI connection object

`table_name` Name of the table in the database. Can be a character string or a `DBI::Id()` object for tables in catalogs/schemas

Returns: A new DBISource object

`DBISource$get_db_type()`: Get the database type

Usage:

```
DBISource$get_db_type()
```

Returns: A string identifying the database type

`DBISource$get_schema()`: Get schema information for the database table

Usage:

```
DBISource$get_schema(categorical_threshold = 20, table_spec = NULL)
```

Arguments:

`categorical_threshold` Maximum number of unique values for a text column to be considered categorical (default: 20)

Returns: A string describing the schema

`DBISource$get_schema_result()`:

Usage:

```
DBISource$get_schema_result(categorical_threshold = 20, table_spec = NULL)
```

`DBISource$get_semantic_views_description()`: Get information about semantic views (if any) for the system prompt.

Usage:

`DBISource$get_semantic_views_description()`

Returns: A string with semantic view information, or empty string if none

`DBISource$execute_query()`: Execute a SQL query

Usage:

`DBISource$execute_query(query)`

Arguments:

`query` SQL query string. If NULL or empty, returns all data

Returns: A data frame with query results

`DBISource$test_query()`: Test a SQL query by fetching only one row

Usage:

`DBISource$test_query(query, require_all_columns = FALSE)`

Arguments:

`query` SQL query string

`require_all_columns` If `TRUE`, validates that the result includes all original table columns (default: `FALSE`)

Returns: A data frame with one row of results

`DBISource$get_data()`: Get all data from the table

Usage:

`DBISource$get_data()`

Returns: A data frame containing all data

`DBISource$cleanup()`: No-op: the DBI connection is owned by the caller. Disconnect it yourself with `DBI::dbDisconnect()` when your application shuts down.

Usage:

`DBISource$cleanup()`

Returns: `NULL` (invisibly)

`DBISource$clone()`: The objects of this class are cloneable with this method.

Usage:

`DBISource$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Connect to a database
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
DBI::dbWriteTable(con, "mtcars", mtcars)

# Create a DBI source
db_source <- DBISource$new(con, "mtcars")
```

```
# Get database type
db_source$get_db_type()  # Returns "SQLite"

# Execute a query
result <- db_source$execute_query("SELECT * FROM mtcars WHERE mpg > 25")

# cleanup() is a no-op: you own `con`, so disconnect it yourself
db_source$cleanup()
DBI::dbDisconnect(con)
```

PinSource *Pin Source*

Description

A DataSource implementation that reads data from a pins board. When the "duckdb" engine is used and the pin type is one DuckDB can read natively (parquet, CSV, JSON), the data is loaded directly from the cached pin files into DuckDB without deserializing into R. For other pin types (e.g. RDS), or when the "sqlite" engine is used, the data is deserialized via pin_read() and must produce a data frame (or tibble), which is then registered with the chosen engine just like DataFrameSource.

When loaded into DuckDB, the connection's external file access is locked down so that LLM-generated SQL cannot reach the filesystem.

Multiple pins (and pins mixed with data frames) can be combined in one chat: every table is materialized into a shared DuckDB connection, so the LLM can join and filter across them. Pins using engine = "sqlite" can't join multi-table chats.

If the pin has a title, description, or tags, QueryChat uses them as the default data_description, which you can override.

Lazy queries with pins

PinSource materializes the full dataset into DuckDB. For large parquet pins where you want lazy query execution, read the pin files yourself and pass a tbl_sql to querychat() instead:

```
paths <- pins::pin_download(board, "my_pin")
con <- DBI::dbConnect(duckdb::duckdb())
DBI::dbExecute(
  con,
  sprintf("CREATE VIEW my_pin AS SELECT * FROM read_parquet('%s')", paths[1])
)
qc <- querychat(dplyr::tbl(con, "my_pin"))
```

The pin files are still downloaded to a local cache — pin_download() always fetches them. But rather than loading everything into memory, DuckDB reads the parquet file lazily through dbplyr.

This approach skips the security lockdown that PinSource applies, so LLM-generated SQL can access files on the local system.

Super classes

DataSource -> DBISource -> PinSource

Active bindings

engine The database engine backing this pin ("duckdb" or "sqlite", read-only).

Methods

Public methods:

- PinSource$new()
- PinSource$register_into()
- PinSource$get_data_description()
- PinSource$cleanup()
- PinSource$clone()

PinSource$new(): Create a new PinSource

Usage:

```
PinSource$new(
  board,
  name,
  ...,
  table_name = name,
  version = NULL,
  engine = getOption("querychat.DataFrameSource.engine", NULL)
)
```

Arguments:

board A pins board object (e.g. from pins::board_folder() or pins::board_connect()).

name Name of the pin to read.

... Not used; included for extensibility.

table_name Name to use for the table in SQL queries. Defaults to the pin name.

version Pin version to read. If NULL (default), reads the latest version.

engine Database engine to use: "duckdb" or "sqlite". Set the global option querychat.DataFrameSource.engine to specify the default engine. If NULL (default), uses the first available engine from duckdb or RSQLite (in that order). Parquet, CSV, and JSON pins are read most efficiently with the "duckdb" engine; with "sqlite" they are deserialized via pin_read() instead.

Returns: A new PinSource object

PinSource$register_into(): Materialize this pin into a shared DuckDB connection.

Internal hook for joining a shared DuckDBExecutor. The caller owns con and locks it down once all tables are materialized.

Usage:

```
PinSource$register_into(con, table_name = self$table_name)
```

Arguments:

con A DuckDB DBI connection, owned by the caller.

table_name Name for the table in con. Defaults to the pin's own table name.

Returns: NULL (invisibly)

`PinSource$get_data_description()`: Get a human-readable description of the pin for use in the system prompt.

Usage:

```
PinSource$get_data_description()
```

Returns: A string with the pin title, description, and tags, or an empty string if none are set.

`PinSource$cleanup()`: Disconnect the DuckDB or SQLite connection this PinSource opened, and shut down the DuckDB instance if used.

Unlike DBISource's `cleanup()`, this isn't a no-op: PinSource always opens its own connection (never a caller-supplied one), so it owns it.

Usage:

```
PinSource$cleanup()
```

Returns: NULL (invisibly)

`PinSource$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
if (rlang::is_installed(c("pins", "duckdb"))) {
  # Create a temporary board and pin some data
  board <- pins::board_temp()
  pins::pin_write(board, mtcars, "mtcars", type = "parquet")

  # Create a PinSource
  ps <- PinSource$new(board, "mtcars")

  # Query the pinned data
  ps$execute_query("SELECT * FROM mtcars WHERE mpg > 25")

  ps$cleanup()
}
```

QueryChat *QueryChat: Interactive Data Querying with Natural Language*

Description

QueryChat is an R6 class built on Shiny, shinychat, and ellmer to enable interactive querying of data using natural language. It leverages large language models (LLMs) to translate user questions into SQL queries, execute them against a data source (data frame or database), and various ways of accessing/displaying the results.

The QueryChat class takes your data (a data frame or database connection) as input and provides methods to:

- Generate a chat UI for natural language queries (e.g., $app(), $sidebar())
- Initialize server logic that returns session-specific reactive values (via $server())
- Access reactive data, SQL queries, and titles through the returned server values (use qc_vals$table("name") for multi-table access)

Usage in Shiny Apps

```
library(querychat)

# Create a QueryChat object
qc <- QueryChat$new(mtcars)

# Quick start: run a complete app
qc$app()

# Or build a custom Shiny app
ui <- page_sidebar(
  qc$sidebar(),
  verbatimTextOutput("sql"),
  dataTableOutput("data")
)

server <- function(input, output, session) {
  qc_vals <- qc$server()

  output$sql <- renderText(qc_vals$sql())
  output$data <- renderDataTable(qc_vals$df())
}

shinyApp(ui, server)
```

Public fields

greeting The greeting message displayed to users.

`history` Conversation history configuration.

`id` ID for the QueryChat instance.

`id_override` Whether the ID was explicitly set by the user.

`tools` The allowed tools for the chat client.

Active bindings

`greeter` The QueryChatGreeter controlling greeting generation; access its `$tables` and `$prompt`.

`system_prompt` Get the system prompt.

`data_source` Removed. Use `$add_table()` and `$remove_table()` to manage tables.

Methods

Public methods:

- `QueryChat$new()`
- `QueryChat$add_table()`
- `QueryChat$add_tables()`
- `QueryChat$remove_table()`
- `QueryChat$table_names()`
- `QueryChat$client()`
- `QueryChat$console()`
- `QueryChat$app()`
- `QueryChat$app_obj()`
- `QueryChat$sidebar()`
- `QueryChat$ui()`
- `QueryChat$page()`
- `QueryChat$server()`
- `QueryChat$generate_greeting()`
- `QueryChat$cleanup()`
- `QueryChat$clone()`

`QueryChat$new()`: Create a new QueryChat object.

Usage:

```
QueryChat$new(
  data_source,
  table_name = missing_arg(),
  ...,
  id = NULL,
  greeting = NULL,
  history = NULL,
  client = NULL,
  tools = c("filter", "query", "visualize"),
  data_description = NULL,
  categorical_threshold = 20,
```

```
  extra_instructions = NULL,
  prompt_template = NULL,
  data_dict = NULL,
  cleanup = NA
)
```

Arguments:

`data_source` Either a data.frame, a database connection (e.g., DBI connection), or `NULL` to defer setting the data source until later. When `NULL`, the data source must be added via `$add_table()` or passed to `$server()` before calling methods that require data access.

`table_name` A string specifying the table name to use in SQL queries. If `data_source` is a data.frame, this is the name to refer to it by in queries (typically the variable name). If not provided, will be inferred from the variable name for data.frame inputs. Required for database connections. Optional when `data_source` is `NULL`: if omitted, `$id` falls back to a generic default, and a table name must be supplied later via `$add_table()` or `$server(data_source =, table_name`

`...` Additional arguments (currently unused).

`id` Optional module ID for the QueryChat instance. If not provided, will be auto-generated from `table_name` (or a generic default when `data_source` is `NULL` and `table_name` is also omitted). The ID is used to namespace the Shiny module.

`greeting` Optional initial message to display to users. Can be a character string (in Markdown format) or a file path. If not provided, a greeting will be generated at the start of each conversation using the LLM, which adds latency and cost. Use `$generate_greeting()` to create a greeting to save and reuse.

`history` Conversation history configuration: `NULL` (default; resolves to `TRUE` when `$server()`/`$app()` is called and nothing else was set), `TRUE`/`FALSE`, or a `shinychat::history_options()` object. Passed straight through to `shinychat::chat_server(history = )`.

`client` Optional chat client. Can be:

- An ellmer::Chat object
- A string to pass to `ellmer::chat()` (e.g., `"openai/gpt-4o"`)
- `NULL` (default): Uses the `querychat.client` option, the `QUERYCHAT_CLIENT` environment variable, or defaults to `ellmer::chat_openai()`

`tools` Which querychat tools to include in the chat client, by default. `"filter"` includes the tools for filtering and resetting the dashboard, `"query"` includes the tool for executing SQL queries, and `"visualize"` includes the tool for rendering visualizations (requires the **ggsql** package; if it is not installed, the tool is dropped with a warning). The default is `c("filter", "query", "visualize")`. Use `tools = "filter"` when you only want the dashboard filtering tools, or when you want to disable the querying tool entirely to prevent the LLM from seeing any of the data in your dataset. The legacy name `"update"` is still accepted as an alias for `"filter"`.

`data_description` Optional description of the data in plain text or Markdown. Can be a string or a file path. This provides context to the LLM about what the data represents.

`categorical_threshold` For text columns, the maximum number of unique values to consider as a categorical variable. Default is 20.

`extra_instructions` Optional additional instructions for the chat model in plain text or Markdown. Can be a string or a file path.

`prompt_template` Optional path to or string of a custom prompt template file. If not provided, the default querychat template will be used. See the package prompts directory for the default template format.

`data_dict` Optional data dictionary. A path to a YAML file, or a list of YAML file paths. See `read_data_dict()` for the expected format.

`cleanup` Whether or not to automatically run `$cleanup()`. By default, cleanup only occurs if `QueryChat` gets created while a Shiny app is running: when created inside a session (e.g., in the server function), cleanup runs when that session ends; when created outside a session (e.g., at the top level of `app.R`), it runs when the app stops. Set to `TRUE` to always clean up, or `FALSE` to never clean up automatically.

Returns: A new `QueryChat` object.

`QueryChat$add_table()`**:** Add a table to this QueryChat instance.

Replacing or removing an existing table after a session has started is an error; adding a new one warns.

Usage:

```
QueryChat$add_table(
  data_source,
  table_name,
  replace = FALSE,
  include_in_greeting = FALSE
)
```

Arguments:

`data_source` A data frame, database connection, or DataSource object.

`table_name` The SQL table name for this data source.

`replace` Whether to replace an existing table with this name. Default is `FALSE`.

`include_in_greeting` Whether to include this table in the greeting context. Default is `FALSE`.

Returns: Invisibly returns `self` for chaining.

`QueryChat$add_tables()`**:** Add multiple tables from a DBI connection in a single call.

Unlike calling `$add_table()` repeatedly, this method builds the system prompt exactly once after all tables have been staged, avoiding N-1 spurious intermediate rebuilds.

Replacing or removing an existing table after a session has started is an error; adding a new one warns.

Usage:

```
QueryChat$add_tables(
  conn,
  tables = NULL,
  replace = FALSE,
  include_in_greeting = FALSE
)
```

Arguments:

`conn` A DBI connection. Only DBI connections are supported; pass individual data frames or other sources via `$add_table()`.

`tables` Table names to register. When `NULL`, all tables returned by `DBI::dbListTables(conn)` are used.

`replace` Whether to replace existing tables with the same name. Default is `FALSE`.

include_in_greeting Whether to include added tables in the greeting context. TRUE includes all tables; FALSE (default) includes none; a character vector includes only those named tables (intersected with the tables being added). Any other type raises an error.

Returns: Invisibly returns self for chaining.

QueryChat$remove_table(): Remove a table from this QueryChat instance.

Removing an existing table after a session has started is an error.

Usage:

```
QueryChat$remove_table(table_name)
```

Arguments:

table_name The name of the table to remove.

Returns: Invisibly returns self for chaining.

QueryChat$table_names(): Return the names of all registered tables.

Usage:

```
QueryChat$table_names()
```

QueryChat$client(): Create a chat client, complete with registered tools, for the current data source.

Usage:

```
QueryChat$client(
  tools = NA,
  update_dashboard = function(query, title, table) {
 },
  reset_dashboard = function(table) {
 },
  visualize = function(data) {
 },
  session = NULL
)
```

Arguments:

tools Which querychat tools to include in the chat client. "filter" includes the tools for filtering and resetting the dashboard and "query" includes the tool for executing SQL queries. By default, when tools = NA, the values provided at initialization are used. The legacy name "update" is still accepted as an alias for "filter".

update_dashboard Optional function to call with the query, title, and table generated by the LLM for the update_dashboard tool.

reset_dashboard Optional function to call when the reset_dashboard tool is called. Takes a table argument.

visualize Optional function to call with a list containing ggsql, title, and widget_id when a visualization succeeds.

session A Shiny session object. Required when "visualize" is in tools and you want interactive chart rendering. When NULL (the default), visualizations still execute but are not rendered as Shiny outputs.

`QueryChat$console()`: Launch a console-based chat interface with the data source.

Usage:

```
QueryChat$console(new = FALSE, ..., tools = "query")
```

Arguments:

`new` Whether to create a new chat client instance or continue the conversation from the last console chat session (the default).

`...` Additional arguments passed to the `$client()` method.

`tools` Which querychat tools to include in the chat client. See `$client()` for details. Ignored when not creating a new chat client. By default, only the `"query"` tool is included, regardless of the `tools` set at initialization.

`QueryChat$app()`: Create and run a Shiny gadget for chatting with data

Usage:

```
QueryChat$app(..., history = NULL)
```

Arguments:

`...` Arguments passed to `$app_obj()`.

`history` Conversation history configuration for the generated app. Defaults to `shinychat::history_options(restore = "bookmark")` when neither this nor `$new()`'s `history` was set, since `$app()`'s whole purpose is a single, shareable demo. When the resolved value has `restore_mode = "bookmark"`, the generated app automatically enables Shiny's own server-side bookmarking.

Returns: Invisibly returns a list of session-specific values.

`QueryChat$app_obj()`: A streamlined Shiny app for chatting with data

Usage:

```
QueryChat$app_obj(..., history = NULL)
```

Arguments:

`...` Additional arguments (currently unused).

`history` Conversation history configuration for the generated app. See `$app()`.

Returns: A Shiny app object that can be run with `shiny::runApp()`.

`QueryChat$sidebar()`: Create a sidebar containing the querychat UI.

Usage:

```
QueryChat$sidebar(
  ...,
  width = 400,
  height = "100%",
  fillable = TRUE,
  id = NULL
)
```

Arguments:

`...` Additional arguments passed to `bslib::sidebar()`.

`width` Width of the sidebar in pixels. Default is 400.

`height` Height of the sidebar. Default is "100%".

`fillable` Whether the sidebar should be fillable. Default is `TRUE`.

`id` Optional ID for the QueryChat instance.

Returns: A `bslib::sidebar()` UI component.

`QueryChat$ui()`**:** Create the UI for the querychat chat interface.

Usage:

```
QueryChat$ui(..., id = NULL)
```

Arguments:

`...` Additional arguments passed to `shinychat::chat_ui()`.

`id` Optional ID for the QueryChat instance.

Returns: A UI component containing the chat interface.

`QueryChat$page()`**:** Create a full-window page containing the querychat UI.

This wraps `shinychat::page_chat()`, making the chat the primary surface of the app, with optional navigation pages, sidebars, and a drawer. Use this instead of `$sidebar()` or `$ui()` when the chat should own the full browser window.

Usage:

```
QueryChat$page(title, ..., id = NULL)
```

Arguments:

`title` Page title displayed in the header. When it is a string and `window_title` is omitted, it is also used as the document title.

`...` Additional arguments passed to `shinychat::page_chat()`.

`id` Optional ID for the QueryChat instance.

Returns: A fillable page UI component suitable for use as the app's UI.

`QueryChat$server()`**:** Initialize the querychat server logic.

Usage:

```
QueryChat$server(
  data_source = NULL,
  client = NULL,
  history = NULL,
  enable_bookmarking = NULL,
  ...,
  table_name = NULL,
  id = NULL,
  session = shiny::getDefaultReactiveDomain()
)
```

Arguments:

`data_source` Optional data source to register for this session only, for the deferred pattern where the source can't be created until the server function runs (for example a per-user database connection). The instance's own tables are not modified; a same-named instance table is shadowed for this session; any connection querychat created for it is cleaned up when the session ends.

`client` Optional chat client override for this session.

`history` Conversation history configuration for this call. Overrides the value set on `$new()`. Resolves to `TRUE` when neither this nor the constructor's `history` was set.

`enable_bookmarking` **[Deprecated]** Use `history = shinychat::history_options(restore_mode = "bookmark")` instead (set on `$new()`, or passed here).

`...` Ignored.

`table_name` Table name to register `data_source` under. Only used when `data_source` is provided. Named-only (placed after `...`) so it can't shift the meaning of existing positional calls.

`id` Optional module ID override.

`session` The Shiny session object.

Returns: A list containing session-specific reactive values and the chat client. For single-table usage, includes `df`, `sql`, `title` directly. For multi-table, use `qc_vals$table("name")` to get a TableAccessor with per-table reactive state. Also includes `table_names()` to list tables. `current_table()` returns the name of the most recently queried table, or `NULL` before any query.

`QueryChat$generate_greeting()`**:** Generate a welcome greeting for the chat.

Usage:

```
QueryChat$generate_greeting(echo = c("none", "output"))
```

Arguments:

`echo` Whether to print the greeting to the console.

Returns: The greeting string in Markdown format.

`QueryChat$cleanup()`**:** Clean up resources this object created.

Closes the query executors and data-source connections querychat opened (in-memory DuckDB), including those of table sets superseded by a late `$add_table()`. Connections you passed in are never closed.

Usage:

```
QueryChat$cleanup()
```

Returns: Invisibly returns `NULL`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Basic usage with a data frame
qc <- QueryChat$new(mtcars)
## Not run:
app <- qc$app()

## End(Not run)
```

```
# With a custom greeting
greeting <- "Welcome! Ask me about the mtcars dataset."
qc <- QueryChat$new(mtcars, greeting = greeting)

# With a specific LLM provider
qc <- QueryChat$new(mtcars, client = "anthropic/claude-sonnet-4-5")

# Generate a greeting for reuse (requires internet/API access)
## Not run:
qc <- QueryChat$new(mtcars)
greeting <- qc$generate_greeting(echo = "text")
# Save greeting for next time
writeLines(greeting, "mtcars_greeting.md")

## End(Not run)

# Or specify greeting and additional options at initialization
qc <- QueryChat$new(
  mtcars,
  greeting = "Welcome to the mtcars explorer!",
  client = "openai/gpt-4o",
  data_description = "Motor Trend car road tests dataset"
)


# Create a QueryChat object from a database connection
# 1. Set up the database connection
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")

# 2. (For this demo) Create a table in the database
DBI::dbWriteTable(con, "mtcars", mtcars)

# 3. Pass the connection and table name to `QueryChat`
qc <- QueryChat$new(con, "mtcars")
```

`querychat` *QueryChat convenience functions*

Description

Convenience functions for wrapping QueryChat creation (i.e., `querychat()`) and app launching (i.e., `querychat_app()`).

Usage

```
querychat(
  data_source,
  table_name = missing_arg(),
  ...,
```

```
  id = NULL,
  greeting = NULL,
  history = NULL,
  client = NULL,
  tools = c("filter", "query", "visualize"),
  data_description = NULL,
  categorical_threshold = 20,
  extra_instructions = NULL,
  prompt_template = NULL,
  data_dict = NULL,
  cleanup = NA
)

querychat_app(
  data_source,
  table_name = missing_arg(),
  ...,
  id = NULL,
  greeting = NULL,
  client = NULL,
  tools = c("filter", "query", "visualize"),
  data_description = NULL,
  categorical_threshold = 20,
  extra_instructions = NULL,
  prompt_template = NULL,
  data_dict = NULL,
  cleanup = NA,
  history = NULL
)
```

Arguments

`data_source`	Either a data.frame or a database connection (e.g., DBI connection).
`table_name`	A string specifying the table name to use in SQL queries.
`...`	Additional arguments (currently unused).
`id`	Optional module ID for the QueryChat instance.
`greeting`	Optional initial message to display to users.
`history`	Conversation history configuration for the generated app. See QueryChat's `$app()` method.
`client`	Optional chat client.
`tools`	Which querychat tools to include in the chat client.
`data_description`	Optional description of the data.
`categorical_threshold`	For text columns, the maximum number of unique values to consider as a categorical variable. Default is 20.

`extra_instructions`
: Optional additional instructions for the chat model.

`prompt_template`
: Optional path to or string of a custom prompt template.

`data_dict`
: Optional data dictionary. A path to a YAML file or a list of paths.

`cleanup`
: Whether or not to automatically run `$cleanup()` when the Shiny session/app stops.

Value

A `QueryChat` object. See QueryChat for available methods.

Invisibly returns the chat object after the app stops.

Examples

```
# Quick start - chat with mtcars dataset in one line
querychat_app(mtcars)
```

`read_data_dict` *Read a Data Dictionary from YAML*

Description

Loads a data dictionary from a YAML file conforming to the data-dict spec. The dictionary is returned as a plain list and can be passed directly to QueryChat via the `data_dict` argument.

If `name` is absent from the YAML file, it defaults to the file stem.

Usage

```
read_data_dict(path)
```

Arguments

`path`
: Path to the YAML file.

Value

A named list with the structure of the YAML file.

TblSqlSource *Data Source: SQL Tibble*

Description

A DataSource implementation for lazy SQL tibbles connected to databases via `dbplyr::tbl_sql()` or `dplyr::sql()`.

Super classes

`DataSource` -> `DBISource` -> `TblSqlSource`

Public fields

`table_name` Name of the table to be used in SQL queries

Methods

Public methods:

- `TblSqlSource$new()`
- `TblSqlSource$get_db_type()`
- `TblSqlSource$get_schema()`
- `TblSqlSource$get_schema_result()`
- `TblSqlSource$execute_query()`
- `TblSqlSource$test_query()`
- `TblSqlSource$prep_query()`
- `TblSqlSource$get_data()`
- `TblSqlSource$cleanup()`
- `TblSqlSource$clone()`

`TblSqlSource$new()`: Create a new TblSqlSource

Usage:

```
TblSqlSource$new(tbl, table_name = missing_arg())
```

Arguments:

`tbl` A `dbplyr::tbl_sql()` (or SQL tibble via `dplyr::tbl()`).

`table_name` Name of the table in the database. Can be a character string, or will be inferred from the `tbl` argument, if possible.

Returns: A new TblSqlSource object

`TblSqlSource$get_db_type()`: Get the database type

Usage:

```
TblSqlSource$get_db_type()
```

Returns: A string describing the database type (e.g., "DuckDB", "SQLite")

`TblSqlSource$get_schema()`: Get schema information about the table

Usage:

```
TblSqlSource$get_schema(categorical_threshold = 20, table_spec = NULL)
```

Arguments:

`categorical_threshold` Maximum number of unique values for a text column to be considered categorical

Returns: A string containing schema information formatted for LLM prompts

`TblSqlSource$get_schema_result()`:

Usage:

```
TblSqlSource$get_schema_result(categorical_threshold = 20, table_spec = NULL)
```

`TblSqlSource$execute_query()`: Execute a SQL query and return results

Usage:

```
TblSqlSource$execute_query(query)
```

Arguments:

`query` SQL query string to execute

Returns: A data frame containing query results

`TblSqlSource$test_query()`: Test a SQL query by fetching only one row

Usage:

```
TblSqlSource$test_query(query, require_all_columns = FALSE)
```

Arguments:

`query` SQL query string to test

`require_all_columns` If `TRUE`, validates that the result includes all original table columns (default: `FALSE`)

Returns: A data frame containing one row of results (or empty if no matches)

`TblSqlSource$prep_query()`: Prepare a generic `SELECT * FROM ____` query to work with the SQL tibble

Usage:

```
TblSqlSource$prep_query(query)
```

Arguments:

`query` SQL query as a string

Returns: A complete SQL query string

`TblSqlSource$get_data()`: Get the unfiltered data as a SQL tibble

Usage:

```
TblSqlSource$get_data()
```

Returns: A `dbplyr::tbl_sql()` containing the original, unfiltered data

`TblSqlSource$cleanup()`: No-op: the connection behind the `tbl_sql` is owned by the caller.

Usage:

```
TblSqlSource$cleanup()
```

Returns: `NULL` (invisibly)

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
con <- DBI::dbConnect(duckdb::duckdb())
DBI::dbWriteTable(con, "mtcars", mtcars)

mtcars_source <- TblSqlSource$new(dplyr::tbl(con, "mtcars"))
mtcars_source$get_db_type()  # "DuckDB"

result <- mtcars_source$execute_query("SELECT * FROM mtcars WHERE cyl > 4")

# Note, the result is not the *full* data frame, but a lazy SQL tibble
result

# You can chain this result into a dplyr pipeline
dplyr::count(result, cyl, gear)

# Or collect the entire data frame into local memory
dplyr::collect(result)

# cleanup() is a no-op: you own `con`, so disconnect it yourself when done
mtcars_source$cleanup()
DBI::dbDisconnect(con, shutdown = TRUE)
```

Index