Understanding the ‘duckdb_server.R’ app

J Christopher Westland

2025-02-19

The two functions in the ‘duckdb_server.R’ app code define API endpoints using **Plumber** in R to interact with a **DuckDB** database. Let’s break them down:

1. Function for Querying Data (`GET /query`)

```r
function(sql = ""){
  if (sql == "") {
    return(list(error = "No SQL query provided"))
  }
  result <- tryCatch({
    DBI::dbGetQuery(conn, sql)
  }, error = function(e) {
    list(error = e$message)
  })
  return(result)
}
```

What it does:

- This function is used as an API endpoint (`GET /query`) to **execute a SELECT SQL query** on the DuckDB database and return the results.
- **Input:** A `sql` string (expected to be a `SELECT` statement) is passed as a URL parameter.
- **Processing:**
- If `sql` is an empty string (`""`), it returns an error message (`"No SQL query provided"`).
- It attempts to execute the SQL query using `DBI::dbGetQuery(conn, sql)`, which fetches results from DuckDB.

- If an error occurs during query execution (e.g., invalid SQL syntax, non-existent table), it captures the error message and returns it in a list.
- **Output:**
- If successful, it returns the result of the SQL query (likely a dataframe converted to JSON by Plumber).
- If an error occurs, it returns a JSON response with an `error` message.

2. Function for Executing SQL Statements (`POST /execute`)

```r
function(sql = ""){
  if (sql == "") {
    return(list(error = "No SQL query provided"))
  }
  result <- tryCatch({
    DBI::dbExecute(conn, sql)
    list(status = "OK")
  }, error = function(e) {
    list(error = e$message)
  })
  return(result)
}
```

What it does:

- This function is used as an API endpoint (`POST /execute`) to **execute non-SELECT SQL statements** (e.g., `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE`).
- **Input:** A `sql` string (expected to be an `INSERT`, `UPDATE`, `DELETE`, or DDL command) is passed as form data.
- **Processing:**
- If `sql` is empty, it returns an error message.
- It attempts to execute the SQL statement using `DBI::dbExecute(conn, sql)`, which modifies the database.

```
-   If execution succeeds, it returns `list(status = "OK")`.

-   If an error occurs (e.g., invalid SQL syntax, constraint violation), it captures the err
```

- **Output:**
- If successful, it returns `{"status": "OK"}`.
- If an error occurs, it returns `{"error": "Error message here"}`.

Summary

- `GET /query`: Executes **SELECT** queries and returns results.
- `POST /execute`: Executes **non-SELECT** SQL statements (e.g., INSERT, UPDATE, DELETE) and returns `"OK"` if successful.
- Both functions use `tryCatch()` to handle errors gracefully.
- Uses `DBI::dbGetQuery()` for queries that return data and `DBI::dbExecute()` for modifying the database.

Explain `sql = ""` and `sql == ""`

The `sql = ""` inside the function signature is not a formula but a **default argument** in R function definitions. Let's break it down.

This means that the function **expects an argument named `sql`**, and if no value is provided when calling the function, it will default to an **empty string (`""`)**.

This is **not** a formula—it's a default parameter.

Why Is This Used?

1. **Ensures the function can be called without an argument**

- Without this default, calling the function without `sql` would throw an error.
- With it, calling `function()` without arguments just means `sql` will be `""`.

2. **Allows Error Handling for Missing SQL Statements**

- The function checks if `sql` is empty and returns an error:

```
if (sql == "") {
  return(list(error = "No SQL query provided"))
}
```

- This prevents running an empty query against the database.

What Happens If a SQL Query Is Passed?

If the function is called with an actual SQL query, for example:

```
GET /query?sql=SELECT%20*%20FROM%20my_table
```

Then `sql = "SELECT * FROM my_table"`, and it will **not** use the default value (`""`).

Conclusion

- `sql = ""` is a **default argument**, not a formula.
- It ensures the function works even if no query is provided.
- The function checks `if (sql == "")` to handle missing queries.
- When an actual query is passed, the default value is ignored.

Explain what are the variables preceded with @ … e.g., @apiTitle, or @param

The variables preceded by `@` in the code, such as `@apiTitle` and `@param`, are **Plumber annotations (or decorators)**. These annotations tell **Plumber** how to interpret the function and expose it as a web API.

Plumber is an R package that converts R functions into RESTful API endpoints.

Understanding Plumber Annotations

Annotations in Plumber start with `@` and are written as **comments** (`#* ...`). These provide metadata and instructions to Plumber.

Examples from the Code:

1. **`@apiTitle`** – Defines the title of the API.

- This specifies a name for the API when viewed in documentation (e.g., in Swagger UI).

2. **`@get /query`** – Defines an HTTP **GET** endpoint.

- This exposes the function as a **GET** API endpoint.
- The API expects a URL-encoded SQL query as a parameter.

3. **`@param`** – Describes a function parameter.

- This tells Plumber that the function expects an argument `sql`, which should be a SQL query.
- It helps document API parameters.

4. **`@post /execute`** – Defines an HTTP **POST** endpoint.

- This exposes the function as a **POST** API endpoint.
- The API expects an SQL statement to execute in the request body.

How Plumber Uses These Annotations

- Plumber **reads** these comments and **automatically documents** the API.
- When the API server starts, these annotations **define how endpoints work**.
- If you run Plumber with:

```r
pr <- plumber::plumb("duckdb_server.R")
pr$run(host = "0.0.0.0", port = 8000)
```

and then open `http://localhost:8000/__swagger__/` in a browser, you will see a **Swagger UI** interface where the API is documented.

Summary

Annotation	Purpose
`@apiTitle`	Defines the API title in the documentation.
`@get /endpoint`	Defines a **GET** request endpoint.
`@post /endpoint`	Defines a **POST** request endpoint.
`@param name`	Describes a function parameter for API documentation.

These annotations **do not affect how the R function runs**—they are only used by **Plumber** to expose functions as API endpoints and generate documentation.

Explain the use of the qualifiers #* in the code?

Note that `#*` is **not** part of the executable code. It is a **special comment format** used by **Plumber** to recognize API annotations.

What is #* Doing?

- In R, # starts a comment, meaning the interpreter **ignores** it when running the code.
- #* is a convention in **Plumber** that marks **special comments** that define API metadata and behavior.
- Plumber reads these comments when the script is loaded and uses them to **define endpoints and generate documentation**.

Example in Context

```r
#* @apiTitle DuckDB Remote API
#* @get /query
#* @param sql The SQL query to run (e.g., "SELECT * FROM my_table")
function(sql = ""){
  if (sql == "") {
    return(list(error = "No SQL query provided"))
  }
  result <- tryCatch({
    DBI::dbGetQuery(conn, sql)
  }, error = function(e) {
    list(error = e$message)
  })
  return(result)
}
```

```
function(sql = ""){
  if (sql == "") {
    return(list(error = "No SQL query provided"))
  }
  result <- tryCatch({
    DBI::dbGetQuery(conn, sql)
  }, error = function(e) {
    list(error = e$message)
  })
  return(result)
}
```

- The #* lines are **ignored** by R when the function runs.
- Plumber reads them when setting up the API.
- Without #*, Plumber wouldn't recognize the function as an API endpoint.

What Happens if You Remove #*?

- The R function will **still work**, but Plumber **won't expose it as an API**.
- It would behave like a normal R function without any web-accessible interface.

Summary

-#* is a **Plumber-specific comment format**
- It **isn't part of the R function execution**
- It tells Plumber **how to handle the function as an API**