Explanation of the Client Code (duckdb_client.R)

J Christopher Westland

2025-02-19

The client script interacts with the **Plumber-based DuckDB server** using HTTP requests to execute SQL commands and retrieve data. Below is a **detailed explanation** of each part.

1. Load Required Libraries

- `httr`: Provides functions for making **HTTP requests** (`GET`, `POST`).
- `jsonlite`: Used to **parse JSON responses** from the server.

New Concepts:

- **HTTP Requests**: The `httr` package allows the R client to send requests to a web API.
- **JSON Parsing**: The `jsonlite` package converts JSON responses into R data structures (`data.frame`, `list`).

2. Define the API Base URL

```
base_url <- "http://your-server-ip:8000"
```

- This specifies the **server address** where the DuckDB API is running.
- `"your-server-ip"` should be replaced with:

- `"127.0.0.1"` (for local testing).
- A **real IP** (e.g., `"192.168.1.100"`) if hosted on a network.
- A **domain name** if hosted remotely (e.g., `"https://myduckdbapi.com"`).

New Concept:

- **Base URL Construction**: The API base URL is used in multiple places, allowing for easy modification.

3. Create a Table (`POST /execute`)

```r
create_table_sql <- "CREATE TABLE IF NOT EXISTS test_table (id INTEGER, name TEXT)"
res <- POST(
  url = paste0(base_url, "/execute"),
  body = list(sql = create_table_sql),
  encode = "form"
)
result <- fromJSON(content(res, as = "text", encoding = "UTF-8"))
print("Create table result:")
print(result)
```

What This Does:

1. **Defines an SQL statement** (`CREATE TABLE IF NOT EXISTS test_table`).
   - Ensures that a table `test_table` with columns `id` (integer) and `name` (text) exists.
2. **Sends an HTTP `POST` request** to `/execute`:
   - `body = list(sql = create_table_sql)`: Sends the SQL query as a **form parameter**.
   - `encode = "form"`: Ensures the request is sent in the correct **form-encoded format**.
3. **Processes the response:**
   - `content(res, as = "text", encoding = "UTF-8")`: Extracts the response as text.
   - `fromJSON(...)`: Converts the JSON response into an R object (likely a `list`).

4. **Prints the result to confirm if the table creation was successful.**

New Concepts:

- **POST Requests (`POST /execute`)**:
  - Used to send **modification** commands (INSERT, DELETE, UPDATE, CREATE).
  - Uses `body = list(sql = create_table_sql)`, ensuring the SQL is **form-encoded**.

4. Insert Data into the Table (`POST /execute`)

```
insert_sql <- "INSERT INTO test_table (id, name) VALUES (1, 'Alice'), (2, 'Bob')"
res <- POST(
  url = paste0(base_url, "/execute"),
  body = list(sql = insert_sql),
  encode = "form"
)
result <- fromJSON(content(res, as = "text", encoding = "UTF-8"))
print("Insert data result:")
print(result)
```

What This Does:

1. **Defines an SQL `INSERT` statement**:
   - Inserts two rows: `(1, 'Alice')` and `(2, 'Bob')` into `test_table`.
2. **Sends an HTTP `POST` request** to `/execute`:
   - Sends the SQL as **form-encoded data**.
3. **Processes the response** to check if the insertion was successful.

New Concepts:

- **Inserting Multiple Records in SQL (`INSERT INTO ... VALUES (...), (...)`)**:
  - Allows batch insertion rather than making multiple requests.

5. Query the Table (`GET /query`)

```
select_sql <- "SELECT * FROM test_table"
res <- GET(
  url = paste0(base_url, "/query"),
  query = list(sql = select_sql)
)
result <- fromJSON(content(res, as = "text", encoding = "UTF-8"))
print("Query result:")
print(result)
```

What This Does:

1. **Defines a `SELECT` SQL statement**:
   - Retrieves all rows from `test_table`.
2. **Sends an HTTP `GET` request** to `/query`:
   - `query = list(sql = select_sql)`: Passes SQL as a **URL query parameter**.
3. **Processes the JSON response** into an R object.
4. **Prints the retrieved data.**

New Concepts:

- **GET Requests (`GET /query`)**:
  - Used for **retrieving** data (does not modify the server state).
  - The SQL query is passed in the **URL query string** instead of the body.

6. JSON Response Handling

Each response is converted using:

```
result <- fromJSON(content(res, as = "text", encoding = "UTF-8"))
```

Why?

- The server responds in **JSON format**.
- `fromJSON()` converts JSON into an **R list or dataframe**, making it usable in R.

Summary of REST API Calls in This Code

Operation	HTTP Method	Endpoint	Request Data	Response
Create Table	`POST`	`/execute`	SQL (`CREATE TABLE ...`)	`{ "status": "OK" }`
Insert Data	`POST`	`/execute`	SQL (`INSERT INTO ...`)	`{ "status": "OK" }`
Query Data	`GET`	`/query`	SQL (`SELECT * FROM ...`)	`{ "id": [1,2], "name": ["Alice", "Bob"] }`

Key Takeaways

1. Uses RESTful HTTP Requests

- `GET /query`: Retrieves data.
- `POST /execute`: Modifies the database.

2. Uses `httr` for HTTP Requests

- `POST()`: Sends SQL commands to be executed.
- `GET()`: Retrieves query results.

3. Uses `jsonlite` for JSON Handling

- Converts JSON responses into R objects (`data.frame`, `list`).

4. Uses Plumber Server's API Endpoints

- The client calls the API hosted on `"http://your-server-ip:8000"`.

5. Works with a Remote Database

- The DuckDB server **runs separately**, and the client **sends requests** via HTTP.

How This Relates to the Server Code

Server Code (Plumber API)	Client Code
Defines `GET /query` endpoint	Calls `GET /query` using `GET()`
Defines `POST /execute` endpoint	Calls `POST /execute` using `POST()`
Uses `DBI::dbGetQuery()` to fetch data	Parses JSON response with `fromJSON()`
Uses `DBI::dbExecute()` for SQL commands	Sends SQL queries as form data

Testing the Code

- Replace `"your-server-ip"` with the **actual** server IP.
- Ensure the **Plumber server is running** (`pr$run(...)`).
- Test the client script and verify the responses.