Example of Plumber API that uses DuckDB and runs on RStudio Server

J. Christopher Westland

2025-03-11

```r
# plumber.R

# Install required packages if not already installed
if (!require("plumber")) install.packages("plumber")
if (!require("duckdb")) install.packages("duckdb")
if (!require("DBI")) install.packages("DBI")
if (!require("jsonlite")) install.packages("jsonlite")

# Load libraries
library(plumber)
library(duckdb)
library(DBI)
library(jsonlite)

# Create DuckDB connection
con <- dbConnect(duckdb::duckdb(), ":memory:")

# Create a sample table and insert some data
dbExecute(con, "CREATE TABLE plumbing_jobs (
    job_id INTEGER,
    customer_name VARCHAR,
    job_type VARCHAR,
    cost DOUBLE
)")

dbExecute(con, "INSERT INTO plumbing_jobs VALUES
    (1, 'John Doe', 'Pipe Repair', 150.50),
    (2, 'Jane Smith', 'Leak Fix', 89.99),
    (3, 'Bob Johnson', 'Installation', 250.00)")
```

```r
#* @apiTitle Plumbing API with DuckDB
#* @apiDescription Simple API for managing plumbing jobs using DuckDB

#* Get all plumbing jobs
#* @get /jobs
function() {
  result <- dbGetQuery(con, "SELECT * FROM plumbing_jobs")
  return(toJSON(result, pretty = TRUE))
}

#* Get job by ID
#* @param id The job ID to lookup
#* @get /job/<id>
function(id) {
  query <- sprintf("SELECT * FROM plumbing_jobs WHERE job_id = %s", id)
  result <- dbGetQuery(con, query)
  if (nrow(result) == 0) {
    return(list(error = "Job not found"))
  }
  return(toJSON(result, pretty = TRUE))
}

#* Add new plumbing job
#* @param customer_name Customer name
#* @param job_type Type of plumbing job
#* @param cost Cost of the job
#* @post /job
function(req, customer_name, job_type, cost) {
  # Get next job ID
  max_id <- dbGetQuery(con, "SELECT MAX(job_id) FROM plumbing_jobs")[[1]]
  new_id <- ifelse(is.na(max_id), 1, max_id + 1)

  # Insert new job
  query <- sprintf("INSERT INTO plumbing_jobs VALUES (%d, '%s', '%s', %f)",
                   new_id, customer_name, job_type, as.numeric(cost))
  dbExecute(con, query)

  return(list(
    status = "success",
    job_id = new_id,
    message = "Job added successfully"
  ))
}

#* Health check endpoint
#* @get /health
function() {
  return(list(
    status = "healthy",
    timestamp = Sys.time(),
    duckdb_version = dbGetQuery(con, "SELECT version()")[[1]]
  ))
}

# Cleanup connection when plumber stops
#* @preempt
function() {
  dbDisconnect(con)
  duckdb::duckdb_shutdown()
}
```

1. Save this code in a file named plumber.R in your RStudio Server working directory.
2. Create a simple launcher script (e.g., run.R):

```r
# run.R
library(plumber)
r <- plumb("plumber.R")
r$run(host = "0.0.0.0", port = 8000)
```

3. From the RStudio Server terminal or R console, run:

```r
source("run.R")
```

How it works:

- The API will be available at http://your-rstudio-server-ip:8000
- Available endpoints:
  - GET /jobs - List all plumbing jobs
  - GET /job/<id> - Get specific job by ID (e.g., /job/1)
  - POST /job - Add new job (with parameters: customer_name, job_type, cost)
  - GET /health - Check API status

To test the API from a browser or curl:

```
# Get all jobs
curl http://your-rstudio-server-ip:8000/jobs

# Get specific job
curl http://your-rstudio-server-ip:8000/job/1

# Add new job
curl -X POST "http://your-rstudio-server-ip:8000/job?customer_name=Alice&job_type=Drain Clea

# Check health
curl http://your-rstudio-server-ip:8000/health
```

Notes:

1. Replace your-rstudio-server-ip with your actual RStudio Server IP address
2. Ensure port 8000 is open in your firewall settings
3. Make sure you have appropriate permissions on RStudio Server
4. The DuckDB database runs in-memory for this example; modify the connection string if you want persistent storage
5. You might need to adjust the host binding (0.0.0.0) based on your server's security requirements

Persistent Storage Implementation

To modify the connection string for persistent storage in DuckDB with the Plumber API I provided, you need to change how the dbConnect() call is configured. In the original code, I already implemented persistent storage with a file path, but I'll explain the options and how to modify it.

Current Persistent Storage Implementation

In the enhanced version, the connection string is:

```
DB_PATH <- "/path/to/your/database.duckdb"
con <- dbConnect(duckdb::duckdb(), DB_PATH)
```

This creates a persistent DuckDB database stored at the specified file path (/path/to/your/database.duckdb). The data will persist between API restarts as long as the file exists and is accessible.

Original In-Memory Implementation

In the first version, I used an in-memory database:

```
con <- dbConnect(duckdb::duckdb(), ":memory:")
```

This creates a temporary database that exists only in RAM and is lost when the connection closes or the API shuts down.

How to Modify for Persistent Storage

To explicitly modify the connection string for persistent storage, you replace :memory: with a file path. Here are the steps and options:

1. Choose a File Path:

- Decide where you want to store the database file on your server's filesystem.
- Ensure the RStudio Server process has write permissions to that location.
- Use a .duckdb extension (conventional but not required).

Example:

```
DB_PATH <- "/home/user/plumbing_data.duckdb"  # Linux/Mac path
# OR
DB_PATH <- "C:/Users/User/plumbing_data.duckdb"  # Windows path
```

2. Update the Connection:

- Modify the dbConnect() call to use this path instead of :memory:.

Example:

```
con <- dbConnect(duckdb::duckdb(), DB_PATH)
```

3. Full Modified Code Snippet: Replace the connection part in your plumber.R file:

```
# Configuration
DB_PATH <- "/home/user/plumbing_data.duckdb"  # Set your persistent storage path
LOG_FILE <- "plumber_api.log"
API_KEY <- "your-secret-api-key"

# Create DuckDB connection with persistent storage
con <- dbConnect(duckdb::duckdb(), DB_PATH)
```

Additional Options for Persistent Storage

DuckDB supports a few additional parameters you can add to the connection string or as arguments to duckdb::duckdb():

- Read-Only Mode: If you want to prevent modifications to the database file:

‘

```
con <- dbConnect(duckdb::duckdb(), DB_PATH, read_only = TRUE)
```

- Custom Configuration: You can pass additional DuckDB configuration options:

```
con <- dbConnect(duckdb::duckdb(),
                 dbdir = DB_PATH,
                 config = list("memory_limit" = "2GB", "threads" = "4"))
```

Here, dbdir is equivalent to the path argument, and config allows setting DuckDB parameters like memory limits or thread counts.

Verifying Persistence

To confirm the data persists:

1. Start the API, add some jobs via the POST endpoint.
2. Stop the API (Ctrl+C in the RStudio terminal).
3. Restart the API and check the /jobs endpoint—your data should still be there.

Notes

- File Location: Replace /home/user/plumbing_data.duckdb with a path that makes sense for your RStudio Server setup. For example, on RStudio Server, you might use a path in your home directory (~/plumbing_data.duckdb) or a shared data directory.
- Permissions: Ensure the RStudio Server user has write access to the directory.
- Backup: Since this is a file-based database, consider backing up the .duckdb file regularly in production.

Enhanced version of the Plumber API with error handling, input validation, authentication, persistent storage, and logging

```r
# plumber.R

# Install required packages if not already installed
required_packages <- c("plumber", "duckdb", "DBI", "jsonlite", "logger", "httr")
for (pkg in required_packages) {
    if (!require(pkg, character.only = TRUE)) install.packages(pkg)
}

# Load libraries
library(plumber)
library(duckdb)
library(DBI)
library(jsonlite)
library(logger)
library(httr)

# Configuration
DB_PATH <- "/path/to/your/database.duckdb"  # Set your persistent storage path
LOG_FILE <- "plumber_api.log"
API_KEY <- "your-secret-api-key"  # Replace with your actual API key

# Setup logging
log_appender(appender_file(LOG_FILE))
log_threshold(INFO)

# Create DuckDB connection with persistent storage
con <- dbConnect(duckdb::duckdb(), DB_PATH)
```

```r
# Initialize database if not exists
dbExecute(con, "CREATE TABLE IF NOT EXISTS plumbing_jobs (
    job_id INTEGER PRIMARY KEY,
    customer_name VARCHAR,
    job_type VARCHAR,
    cost DOUBLE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)")

# Authentication filter
#* @filter check-auth
function(req, res) {
    api_key <- req$HTTP_X_API_KEY
    if (is.null(api_key) || api_key != API_KEY) {
        res$status <- 401
        log_error("Authentication failed for IP: %{req$REMOTE_ADDR}s")
        return(list(error = "Unauthorized: Invalid or missing API key"))
    }
    plumber::forward()
}

#* @apiTitle Plumbing API with DuckDB
#* @apiDescription Enhanced API for managing plumbing jobs

#* Get all plumbing jobs
#* @get /jobs
function(req, res) {
    tryCatch({
        result <- dbGetQuery(con, "SELECT * FROM plumbing_jobs")
        log_info("Retrieved all jobs - Count: %{nrow(result)}s")
        return(toJSON(result, pretty = TRUE))
    }, error = function(e) {
        res$status <- 500
        log_error("Error retrieving jobs: %{conditionMessage(e)}s")
        return(list(error = "Internal server error"))
    })
}

#* Get job by ID
#* @param id The job ID to lookup
#* @get /job/<id>
function(req, res, id) {
    # Validate ID
    if (!grepl("^[0-9]+$", id)) {
        res$status <- 400
        log_warn("Invalid job ID format: %{id}s")
        return(list(error = "Invalid job ID: must be numeric"))
    }

    tryCatch({
        query <- sprintf("SELECT * FROM plumbing_jobs WHERE job_id = %s", id)
        result <- dbGetQuery(con, query)
        if (nrow(result) == 0) {
            res$status <- 404
            log_info("Job not found: %{id}s")
            return(list(error = "Job not found"))
        }
        log_info("Retrieved job: %{id}s")
        return(toJSON(result, pretty = TRUE))
    }, error = function(e) {
        res$status <- 500
        log_error("Error retrieving job %{id}s: %{conditionMessage(e)}s")
        return(list(error = "Internal server error"))
    })
}

#* Add new plumbing job
#* @param customer_name Customer name
#* @param job_type Type of plumbing job
#* @param cost Cost of the job
#* @post /job
function(req, res, customer_name, job_type, cost) {
    # Input validation
    if (is.null(customer_name) || nchar(customer_name) < 2) {
        res$status <- 400
        log_warn("Invalid customer name: %{customer_name}s")
        return(list(error = "Customer name must be at least 2 characters"))
    }

    if (is.null(job_type) || !job_type %in% c("Pipe Repair", "Leak Fix", "Installation", "Dra
        res$status <- 400
        log_warn("Invalid job type: %{job_type}s")
        return(list(error = "Invalid job type"))
    }
    cost_num <- as.numeric(cost)
    if (is.na(cost_num) || cost_num <= 0) {
        res$status <- 400
        log_warn("Invalid cost: %{cost}s")
        return(list(error = "Cost must be a positive number"))
    }

    tryCatch({
        # Get next job ID
        max_id <- dbGetQuery(con, "SELECT MAX(job_id) FROM plumbing_jobs")[[1]]
        new_id <- ifelse(is.na(max_id), 1, max_id + 1)

        # Insert new job
        query <- sprintf("INSERT INTO plumbing_jobs (job_id, customer_name, job_type, cost) V
                        new_id, customer_name, job_type, cost_num)
        dbExecute(con, query)

        log_info("Added new job: %{new_id}s for %{customer_name}s")
        return(list(
            status = "success",
            job_id = new_id,
            message = "Job added successfully"
        ))
    }, error = function(e) {
        res$status <- 500
        log_error("Error adding job: %{conditionMessage(e)}s")
        return(list(error = "Internal server error"))
    })
}

#* Health check endpoint
#* @get /health
function(req, res) {
    tryCatch({
        db_version <- dbGetQuery(con, "SELECT version()")[[1]]
        log_info("Health check requested")
        return(list(
            status = "healthy",
            timestamp = Sys.time(),
            duckdb_version = db_version
        ))
    }, error = function(e) {
        res$status <- 500
        log_error("Health check failed: %{conditionMessage(e)}s")
        return(list(error = "Health check failed"))
    })
}

#* @preempt
function() {
    log_info("Shutting down API")
    dbDisconnect(con)
    duckdb::duckdb_shutdown()
}
```

To use this enhanced version:

1. Update the configuration values:

```
DB_PATH <- "/path/to/your/database.duckdb"  # Set a valid path
API_KEY <- "your-secret-api-key"           # Set your API key
```

Launch script remains the same:

```
# run.R
library(plumber)
r <- plumb("plumber.R")
r$run(host = "0.0.0.0", port = 8000)
```

New features added:

1. Error Handling:

- Try-catch blocks around all database operations
- Appropriate HTTP status codes (400, 401, 404, 500)
- Meaningful error messages

2. Input Validation:

- Job ID must be numeric
- Customer name minimum length
- Job type from predefined list

- Cost must be positive number

3. Authentication:

    - API key required in header (X-API-KEY)
    - 401 Unauthorized response for invalid/missing key

4. Persistent Storage:

- Uses file-based DuckDB instead of in-memory
- Table creation with timestamp

```
-   Data persists between restarts
```

5. Logging:

- Uses logger package
- Logs to file with timestamp
- Different log levels (INFO, WARN, ERROR)

To test with authentication:

```
# Get all jobs
curl -H "X-API-KEY: your-secret-api-key" http://your-rstudio-server-ip:8000/jobs

# Add new job
curl -X POST -H "X-API-KEY: your-secret-api-key" \
     "http://your-rstudio-server-ip:8000/job?customer_name=Alice&job_type=Drain%20Cleaning&c
```

Additional setup:

- Ensure the DB_PATH directory is writable
- Set a secure API_KEY
- Configure your server firewall appropriately
- Consider adding HTTPS in production

The log file will contain entries like:

```
[INFO] 2025-03-11 12:00:00 Retrieved all jobs - Count: 3
[ERROR] 2025-03-11 12:00:01 Authentication failed for IP: 127.0.0.1
```