November 2024

R Shiny Masterclass Series - Advanced

Session 5

Advanced data sources and performance of R Shiny apps

EPI-interactive

Agenda

Session 1 | October 30th | Introduction, recap and responsive interfaces in R Shiny

Session 2 | November 1st | Advanced reactivity and UX considerations

Session 3 | November 5th | Useful R packages to extend core Shiny functionality

Session 4 | November 6th | Managing complexity: modularizing with the module pattern

Session 5 | November 8th | Advanced data sources and processing

Session 6 | November 12th | Automated report generation

Session 7 | November 13th | User authentication, Extended exercise

Session 8 | November 15th | AI Tools, Programming sins and how to avoid them

Today

- Namespace Tips
- Databases in Shiny
- Performance considerations in Shiny apps

Debugging Namespaces

```
download_ui <- function(id){
  ns <- NS(id)
  message(ns('my_id'))
```

```
download_server <- function(id, data) {
  moduleServer(
    id,
    function(input, output, session) {
      message(session$ns('my_id'))
```

```
> runApp('challenge')

Listening on http://127.0.0.1:4713
table-download_data-my_id
region-region_summary_table-download_data-my_id
table-table-download_data-my_id
region-region_summary_table-region-region_summary_table-download_data-my_id
```

```
> runApp('challenge')

Listening on http://127.0.0.1:4713
table-download_data-my_id
region-region_summary_table-download_data-my_id
table-download_data-my_id
region-region_summary_table-download_data-my_id
```

Databases in Shiny

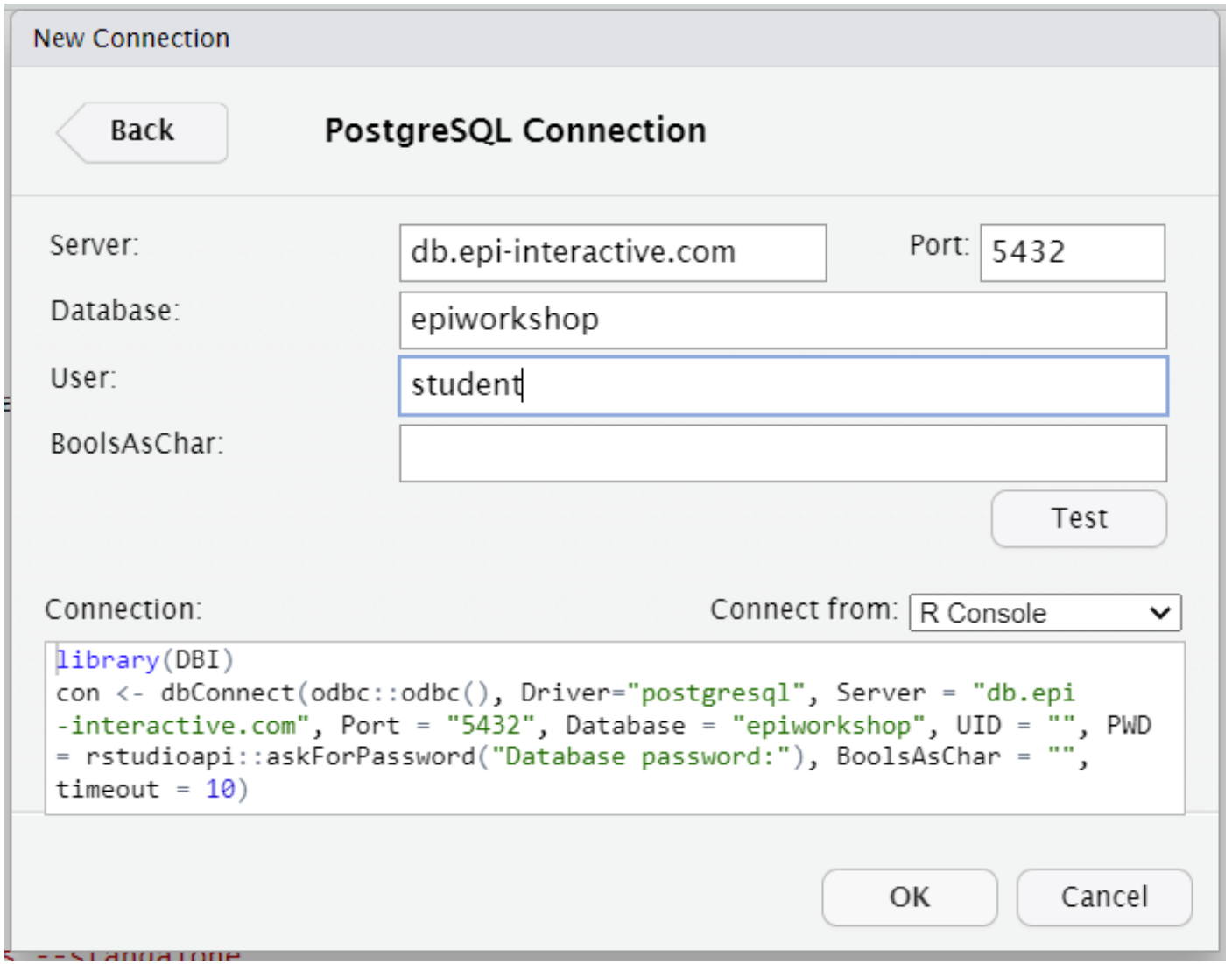
When to use databases over files

- Well defined, structured data
- Need to frequently insert/update
  - Safety against multiple users doing so at the same time
- More complexity to the data
- Remote data

Integrating to our app: R Libraries

- odbc: Allows us to use the correct database drivers for our connection
- DBI: Connection management for singular database connections
- pool: Connection management for the database – multiple connections

Database connections in Posit Cloud



Integrating to our app: global.R

```
library(odbc)
library(DBI)


dbConn <- dbConnect(
RPostgres::Postgres(),
host = "db.epi-interactive.com",
port = 5432,
dbname = "epiworkshop",
user = "student",
password = “student”
)
```

Where to create the connection?

- Global environment
  - One connection can be shared by multiple user sessions
  - Need to leave an open connection – not ideal
  - Closing the connection may impact other users

- Server.R
  - One connection per user session
  - Slower initial application load
  - No interference with other users

Make sure to close the connection when we are finished!

Storing DB connection details

- Don’t want to store sensitive information directly in code!
- Use environment variables instead
- Stored in a .env file in format:

  VAR_NAME = value

- In global.R:
  - readRenviron(".env")
- Using environment variables (watch out for object types):
  - Sys.getenv(“VAR_NAME”)

Query types

- **SELECT [cols] from [table]**
  - Retrieving rows from a specified database table where some conditions are met
  - ***Does not make any changes to the data***
- INSERT INTO [table]([cols])
  - Add new rows to a specified database table, with the provided column values
- UPDATE [table]
  - Modify existing rows in a specified database table
- DELETE FROM [table]
  - Remove rows from a specified database table where some conditions are met

Integrating to our app: Sample query

```
getTableName <- function() {

  query <- 'SELECT
  column1, [column2]
  FROM schema.tableName
  WHERE column1 = 'var''
  result <- dbGetQuery(dbPool, query)
}
```

[PostgreSQL Query Cheat Sheet]

Integrating to our app: server.R

```
tableData <- reactive({
     getTableName()
})

filteredData <- reactive({
     req(tableData())
     …
})

onStop(function() {  dbDisconnect(dbConn)})
```

Parameterised SQL queries

- SQL queries often need to be customised to accept different parameters
- In Shiny, we often need SQL queries based on user inputs
- Parameterised SQL queries let us make query templates, then fill in with variables when we evaluate the query
- Need to do this carefully to avoid SQL injection!

```r
# Not parameterised
query <- "SELECT * FROM world_temp_data
          WHERE iso_a2 IN ('CA');"

#the parameter in question
var <- "CA"

# dangerously parameterised
query <- paste0(
  "SELECT * FROM world_temp_data
  WHERE iso_a2 IN ('", var, "');"
)

# safely parameterised
query <- sqlInterpolate(
  "SELECT * FROM world_temp_data
  WHERE iso_a2 IN ?tag",
  tag = var
)
```

Safely parameterised SQL queries

- To protect against SQL injection, we can use the *sqlInterpolate* function
- This allows us to include variables safely in our SQL queries
- Protects against malicious user inputs
- Use ***dbExecute(con, query)*** or ***dbGetQuery(con, query)*** to process

```
sql <- "SELECT * FROM X WHERE name = ?name"
sqlInterpolate(ANSI(), sql, name = "Nick")

# This is safe because the single quote has been double escaped
sqlInterpolate(ANSI(), sql, name = "H'); DROP TABLE--;")

# Using paste0() could lead to dangerous SQL with carefully crafted inputs
# (SQL injection)
name <- "H'); DROP TABLE--;"
paste0("SELECT * FROM X WHERE name = '", name, "'")

# Use SQL() or dbQuoteIdentifier() to avoid escaping
sql2 <- "SELECT * FROM ?table WHERE name in ?names"
sqlInterpolate(ANSI(), sql2,
               table = dbQuoteIdentifier(ANSI(), "X"),
               names = SQL("('a', 'b')")
)

# Don't use SQL() to escape identifiers to avoid SQL injection
sqlInterpolate(ANSI(), sql2,
               table = SQL("X; DELETE FROM X; SELECT * FROM X"),
               names = SQL("('a', 'b')")
)
```

dbplyr - dplyr for SQL queries

- Using dbplyr, we can read data directly from our SQL tables
- Automatically loaded when dplyr detects database connectivity
- Use regular dplyr syntax to transform data
- ‘Lazy’ evaluation – SQL code is only evaluated in the database when the data is needed (like reactivity)
- Some more complex data transformations may be hard to replicate in dbplyr

dbplyr – dplyr for SQL queries

- Create a reference to the database table using ***tbl(con, table)***
- Write dplyr data transformations as usual
- When this code runs, dbplyr generates a SQL query (visible with **show_query()**) and evaluates against the database table
- Use the **collect()** function to retrieve results

```
> query <- tbl(con, "world_temp_data") %>%
+   filter(iso_a2 %in% "CA") %>%
+   select(country, city, region, month, year)
> show_query(query)
<SQL>
SELECT "country", "city", "region", "month", "year"
FROM "world_temp_data"
WHERE ("iso_a2" IN ('CA'))
> collect(query)
# A tibble: 74,245 × 5
   country city    region        month  year
   <chr>   <chr>   <chr>         <int> <int>
 1 Canada  Calgary North America     1  1995
 2 Canada  Calgary North America     1  1995
 3 Canada  Calgary North America     1  1995
 4 Canada  Calgary North America     1  1995
 5 Canada  Calgary North America     1  1995
```

Performance considerations

Improving performance

Think of a Shiny app like a restaurant:

- The R process is represented by the chef
- User requests are customer's meal orders

- Different approaches to optimising the restaurant:
  - **Make the chef more efficient (optimise R code)**
  - Hire a prep cook (pre-processing)
  - Hire more chefs (add more R processes)
  - Purchase more equipment (more server memory)
  - Construct a new restaurant (multiple servers)

How to fix our pain points

- Minimise access to files / databases
- Calculations should be done once
- Don’t Repeat Yourself (DRY principle)
- Filter datasets by most common feature first
- Separate filtering into staged reactives

What pain points have you run into?

Performance tips

- Preprocessing
- Profiling
- Caching
- Computer resources
- Network monitoring (in browser)
- Load testing

Reactive functions and caching

- Reactive functions in Shiny already enable some caching
- When a reactive is called first time, results are stored in memory
- Next time that reactive is called, results are looked up
- Change in any reactive values in expression recalculates the reactive

- *What if we want to cache different combinations of those values?*

bindCache

- Allows reactive outputs to be cached in memory for quick access
- Reactive variables form the ‘key’ for that reactive
- Different levels of caching available:
  - Session – one cache per user session.
  - Application – cache is shared across multiple sessions. Users can take access calculations done in another user’s session

```
city_data <- reactive({
    fetchData(input$city)
  }) %>%
  bindCache(input$city)
```

Exercise

In RStudio Cloud, open **Session-5**, then /stage1

- Create a .env file (using .env.example as a template), fill it with DB connection details
- In global.R
  - Load the .env file in global.R using readRenviron(), then use the environment variables with Sys.getenv() to create a db connection
  - Complete the getCountries function, using dbplyr to query the *world_temp_data* table from the database, filter by country iso_a2 codes, and summarise an average monthly temperature
- In *temperature-page-module.R,*
  - Create a reactive to store the iso_a2 codes from filtered_data
  - Create a reactive to retrieve temperature data from DB based on the selected countries using your getCountries function
  - Create a reactive to filter the country temperature data by year, then use this reactive in the *temperature_table*
  - Apply bindCache as appropriate to these reactives to improve performance.

Temperature data structure

	id	region	country	iso_a2	city	month	day	year	date	avg_temp
1	1089209	North America	Canada	CA	Calgary	1	1	1995	1995-01-01	12.6
2	1089210	North America	Canada	CA	Calgary	1	2	1995	1995-01-02	4.5
3	1089211	North America	Canada	CA	Calgary	1	3	1995	1995-01-03	2.5
4	1089212	North America	Canada	CA	Calgary	1	4	1995	1995-01-04	11.4
5	1089213	North America	Canada	CA	Calgary	1	5	1995	1995-01-05	11.3
6	1089214	North America	Canada	CA	Calgary	1	6	1995	1995-01-06	4.0
7	1089215	North America	Canada	CA	Calgary	1	7	1995	1995-01-07	5.4
8	1089216	North America	Canada	CA	Calgary	1	8	1995	1995-01-08	4.5
9	1089217	North America	Canada	CA	Calgary	1	9	1995	1995-01-09	7.1
10	1089218	North America	Canada	CA	Calgary	1	10	1995	1995-01-10	17.1

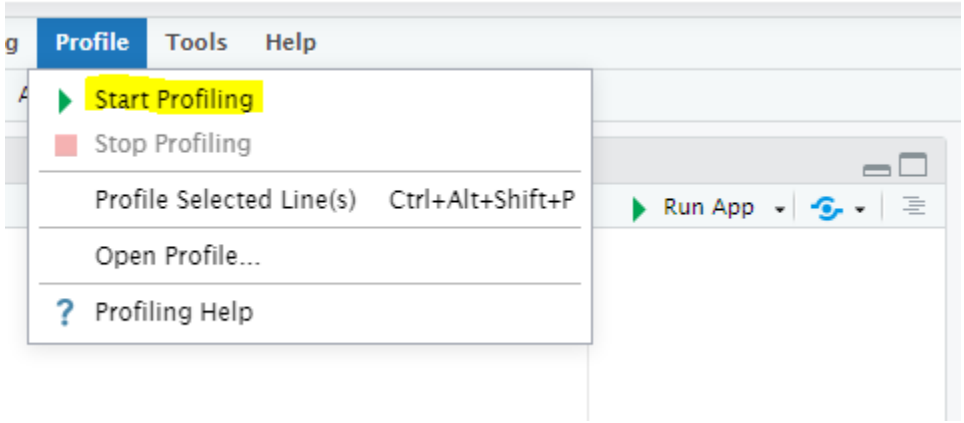
Profiling

Profiling / Profvis

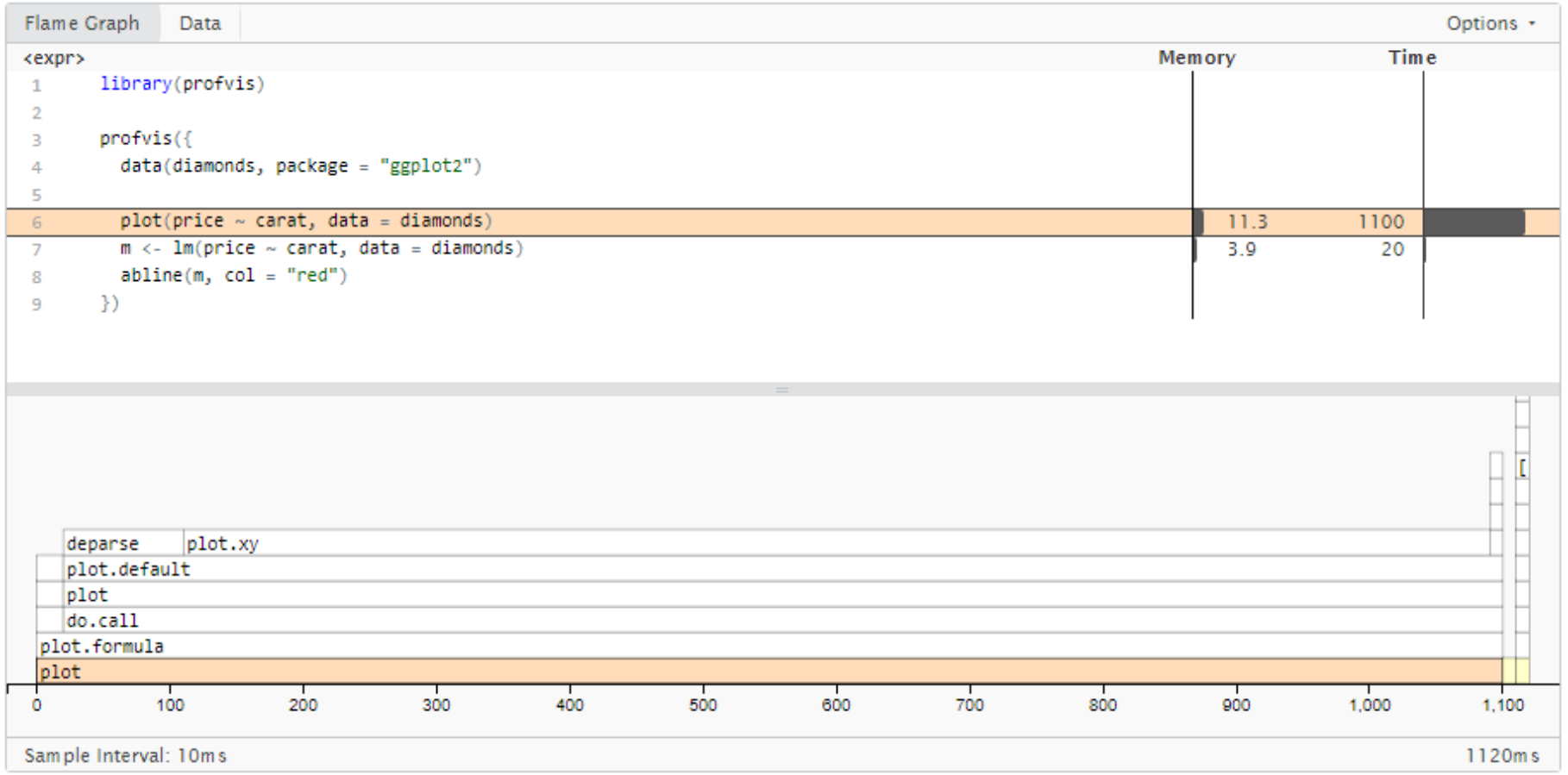
- Built into RStudio IDE
- Records app performance over time
- Allows you to find your pain points
- NOTE: Only optimize if needed

Profiling: How to

- Profile > Start Profiling
  - This can be done before or during the Shiny app runtime
- Take actions in the Shiny app to make the server do some work
- Profile > Stop Profiling



Results: flame graph



Next time

- Profiling
- Automated Reports with Shiny and Quarto

Challenge:

- Move your connection into *server.R* so you only make one connection per user
- Rework the *getCountries()* method to include the year as part of the query
- Change the *getCountries()* method to write a *sqlInterpolate* query instead of using *dbplyr*
- Profile the app to identify other performance areas to focus on, see how you can improve the performance further

- Share your work on the session 5 forum!