Run Queries Safely

We will review four options to run SQL commands safely using the DBI package:

SQL Injection Attack

The `dbGetQuery()` command allows us to write queries and retrieve the results. The query has to be written using the SQL syntax that matches to the database type.

For example, here is a database that contains the *airports* data from NYC Flights data:

```
dbGetQuery(con, "SELECT * FROM airports LIMIT 5")
```

```
##   faa                           name      lat       lon  alt tz dst
## 1 04G              Lansdowne Airport 41.13047 -80.61958 1044 -5   A
## 2 06A  Moton Field Municipal Airport 32.46057 -85.68003  264 -6   A
## 3 06C            Schaumburg Regional 41.98934 -88.10124  801 -6   A
## 4 06N                Randall Airport 41.43191 -74.39156  523 -5   A
## 5 09J          Jekyll Island Airport 31.07447 -81.42778   11 -5   A
```

Often you need to write queries that depend on user input. For example, you might want to allow the user to pick an airport to focus their analysis on. To do this, it’s tempting to create the SQL string yourself by pasting strings together:

```
airport_code <- "GPT"
dbGetQuery(con, paste0("SELECT * FROM airports WHERE faa = '", airport_
```

```
##   faa            name      lat       lon alt tz dst
## 1 GPT Gulfport-Biloxi 30.40728 -89.07011  28 -6   A
```

Here `airport_code` is created in the script, in real-life it might be an input typed into a Shiny app.

The problem with creating SQL strings with `paste0()` is that a careful attacker can create inputs that return more rows than you want:

```
airport_code <- "GPT' or faa = 'MSY"
dbGetQuery(con, paste0("SELECT * FROM airports WHERE faa = '", airport_
```

```
##   faa                           name      lat       lon alt tz dst
## 1 GPT                 Gulfport-Biloxi 30.40728 -89.07011  28 -6   A
## 2 MSY Louis Armstrong New Orleans Intl 29.99339 -90.25803   4 -6   A
```

Or take **destructive actions on your database**:

```
airport_code <- "GPT'; DROP TABLE 'airports"
dbGetQuery(con, paste0("SELECT * FROM airports WHERE faa = '", airport_
```

This is called **SQL injection attack**.

There are three ways to avoid this problem:

- Use a parameterised query with `dbSendQuery()` and `dbBind()`
- Use the `sqlInterpolate()` function to safely combine a SQL string with data
- Manually escape the inputs using `dbQuoteString()`

These are ordered by the level of safety they provide: if you can use `dbSendQuery()` and `dbBind()`, you should.

Parameterized queries

All modern database engines provide a way to write **parameterised queries**, queries that contain some placeholder that allows you to re-run the query multiple times with different inputs. This protects you from SQL injection attacks, and as an added benefit, the database can often optimise the query so it runs faster.

Using a parameterised query with DBI requires three steps.

1. You create a query containing a `?` placeholder and send it to the database with `dbSendQuery()`:

   ```
   airport <- dbSendQuery(con, "SELECT * FROM airports WHERE fa
   ```

2. Use `dbBind()` to execute the query with specific values, then `dbFetch()` to get the results:

   ```
   dbBind(airport, list("GPT"))
   ```

```
dbFetch(airport)
```

```
##   faa           name      lat       lon alt tz dst
## 1 GPT Gulfport-Biloxi 30.40728 -89.07011  28 -6   A
```

3. Once you're done using the parameterised query, clean it up by calling `dbClearResult()`

```
dbClearResult(airport)
```

Using `glue_sql()`

Parameterized queries are generally the safest and most efficient way to pass user defined values in a query, however not every database driver supports them. The function `glue_sql()`, part of the the `glue` package, is able to handle the SQL quoting and variable placement.

```
library(glue)

airport_sql <- glue_sql("SELECT * FROM airports WHERE faa = ?")
airport <- dbSendQuery(con, airport_sql)

dbBind(airport, list("GPT"))
dbFetch(airport)
```

```
##   faa           name      lat       lon alt tz dst
## 1 GPT Gulfport-Biloxi 30.40728 -89.07011  28 -6   A
```

```
dbClearResult(airport)
```

If you place an astersk `*` at the end of a glue expression the values will be collapsed with commas. This is useful for the SQL IN Operator for instance.

```
airport_sql <- glue_sql("SELECT * FROM airports WHERE faa IN ({airports
                        airports = c("GPT", "MSY"),
                        .con = con
                        )

airport <- dbSendQuery(con, airport_sql)

dbFetch(airport)
```

```
##   faa                           name      lat       lon alt tz dst
## 1 GPT                 Gulfport-Biloxi 30.40728 -89.07011  28 -6   A
## 2 MSY Louis Armstrong New Orleans Intl 29.99339 -90.25803   4 -6   A
```

```
dbClearResult(airport)
```

Interpolation by “hand”

While all modern databases support parameterised queries, they are not always supported in individual database drivers. If you find that `dbBind()` doesn’t work with the database connector you are using, you can fall back on `sqlInterpolate()`, which will safely do the interpolation for you.

```
airport_code <- "GPT"

sql <- sqlInterpolate(con,
  "SELECT * FROM airports  where faa = ?code",
  code = airport_code
)
sql
```

```
## <SQL> SELECT * FROM airports  where faa = 'GPT'
```

```
dbGetQuery(con, sql)
```

```
##   faa            name      lat       lon alt tz dst
## 1 GPT Gulfport-Biloxi 30.40728 -89.07011  28 -6   A
```

The query returns no records if we try the same SQL injection attack:

```
airport_code <- "GPT' or faa = 'MSY"

sql <- sqlInterpolate(con,
  "SELECT * FROM airports  where faa = ?code",
  code = airport_code
)
sql
```

```
## <SQL> SELECT * FROM airports  where faa = 'GPT'' or faa = ''MSY'
```

```
dbGetQuery(con, sql)
```

```
## [1] faa  name lat  lon  alt  tz   dst
## <0 rows> (or 0-length row.names)
```

Manual escaping

Sometimes you can't create the SQL you want using either of the previous methods. If you're in this unhappy situation, first make absolutely sure that you haven't missed an existing DBI helper function that does what you need. You need to be extremely careful when doing the escaping yourself, and it's better to rely on existing code that multiple people have carefully reviewed.

However, if there's no other way around it, you can use `dbQuoteString()` to add the quotes for you. This method will automatically take care of dangerous characters in the same way as `sqlInterpolate()` (*better*) and `dbBind()` (*best*).

```
airport_code <- "GPT' or faa = 'MSY"

sql <- paste0("SELECT * FROM airports WHERE faa = ", dbQuoteString(con,

sql
```

```
## [1] "SELECT * FROM airports WHERE faa = 'GPT'' or faa = ''MSY'"
```

```
dbGetQuery(con, sql)
```

```
## [1] faa  name lat  lon  alt  tz   dst
## <0 rows> (or 0-length row.names)
```

You may also need `dbQuoteIdentifier()` if you are creating tables or relying on user input to choose which column to filter on.