Package: gqlr (via r-universe)

August 20, 2024

Title 'GraphQL' Server in R

Version 0.0.2.9000

Description Server implementation of 'GraphQL' <http://graphql.github.io/graphql-spec/>, a query language originally created by Facebook for describing data requirements on complex application data models. Visit <http://graphql.org> to learn more about 'GraphQL'.

Depends R (>= 3.2.2)

License MIT + file LICENSE

URL http://schloerke.com/gqlr/, https://github.com/schloerke/gqlr, http://graphql.github.io/graphql-spec/, http://graphql.org

BugReports https://github.com/schloerke/gqlr/issues

Imports graphql (>= 1.3), magrittr, pryr, R6, jsonlite

Suggests plumber, roxygen2 (>= 7.0.0), testthat (>= 3.1.5)

Roxygen list()

RoxygenNote 7.2.2

Encoding UTF-8

Config/testthat/edition 3

Config/Needs/website tidyverse/tidytemplate

Collate 'AAA-utils.R' 'R6--aaa-utils.R' 'S3--aaa-setup.R' 'R6--definition.R' 'R6-3.1.1-types-scalars.R' 'R6-6.1-executing-requests.R' 'R6-Schema.R' 'R6z-from-json.R' 'graphql_json.R' 'R6-3.2-directives.R' 'gqlr_schema.R' 'R6-4-introspection.R' 'R6-6.2-executing-operations.R' 'R6-6.3-executing-selection-sets.R' 'R6-6.4-executing-fields.R' 'R6-7-response.R' 'R6-ErrorList.R' 'R6-ObjectHelpers.R' 'R6-Result.R' 'R6-VariableValdationHelper.R' 'S3-3.1.2.3-validation-object-type.R' 'S3-str.R' 'gqlr-package.R' 'server.R' 'upgrade_query_remove_fragments.R' 'validation-arguments.R' 'validation-input-coercion.R' 'validation-selection-set-can-merge.R' 'validation-query.R' 'zzz.R'

Repository https://schloerke.r-universe.dev

RemoteUrl https://github.com/schloerke/gqlr

RemoteRef HEAD

RemoteSha ed77c56a24181c2006bf06815e541a2981788ea1

Contents

as_R6 2
ErrorList 3
execute_request 4
gqlr_schema 6
parse_ast 11
Schema 12
server 13

Index **16**

`as_R6` *As R6*

Description

Debug method that strips all gqlr classes and assigns the class as `'R6'`

Usage

```
as_R6(x)
```

Arguments

`x`	any object. If it inherits `'R6'`, then the class of `x` is set to `'R6'`

Examples

```
Int <- getFromNamespace("Int", "gqlr")$clone()
print(Int)
print(as_R6(Int))
```

ErrorList *ErrorList*

Description

Handles all errors that occur during query validation. This object is returned from execute request function (`ans <- execute_request(query, schema)`) under the field 'error_list' (`ans$error_list`).

Usage

```
answer <- execute_request(my_request, my_schema)
answer$error_list
```

Initialize

verbose boolean that determines if errors will be printed on occurrence. Defaults to `TRUE`

Details

`$n` count of errors received

`$errors` list of error information

`$verbose` boolean that determines of errors are printed when received

`$has_no_errors()` helper method to determine if there are no errors

`$has_any_errors()` helper method to determine if there are any errors

`$get_sub_source(loc)` helper method to display a subsection of source text given Location information

`$add(rule_code, ...)` add a new error according to the `rule_code` provided. Remaining arguments are passed directly to `paste(..., sep = "")` with extra error rule information

`$.format(...)` formats the error list into user friendly text. Remaining arguments are ignored

`$print(...)` prints the error list by calling `self$format(...)`

Examples

```
error_list <- ErrorList$new()
error_list
error_list$has_any_errors() # FALSE
error_list$has_no_errors() # TRUE

error_list$add("3.1.1", "Multiple part", " error about Scalars")
error_list
error_list$has_any_errors() # TRUE
error_list$has_no_errors() # FALSE
```

`execute_request` *Execute GraphQL server response*

Description

Executes a GraphQL server request with the provided request.

Usage

```
execute_request(
  request,
  schema,
  ...,
  operation_name = NULL,
  variables = list(),
  initial_value = NULL,
  verbose_errors = is_interactive()
)
```

Arguments

`request`	a valid GraphQL string
`schema`	a character string (to be used along side `initial_value`) or a schema object created from `gqlr_schema`
`...`	ignored for paramter expansion
`operation_name`	name of request operation to execute. If not value is provided it will use the operation in the request string. If more than one operations exist, an error will be produced. See https://graphql.github.io/graphql-spec/October2016/#GetOperation()
`variables`	a named list containing variable values. https://graphql.github.io/graphql-spec/October2016/#sec-Language.Variables
`initial_value`	default value for executing requests. This value can either be provided and/or combined with the resolve method of the query root type or mutation root type. The value provided should be a named list of the field name (key) and a value matching that field name type. The value may be a function that returns a value of the field name type.
`verbose_errors`	logical to determine if error-like messages should be displayed when processing a request that finds unknown structures. Be default, this is only enabled when `verbose_errors = rlang::is_interactive()` is `TRUE`.

References

https://graphql.github.io/graphql-spec/October2016/#sec-Execution

Examples

```
# bare bones
schema <- gqlr_schema("
  type Person {
    name: String
    friends: [Person]
  }
  schema {
    query: Person
  }
")

data <- list(
  name = "Barret",
  friends = list(
    list(name = "Ryan", friends = list(list(name = "Bill"), list(name = "Barret"))),
    list(name = "Bill", friends = list(list(name = "Ryan")))
  )
)

ans <- execute_request("{ name }", schema, initial_value = data)
ans$as_json()

execute_request("
  {
    name
    friends {
      name
      friends {
        name
        friends {
          name
        }
      }
    }
  }",
  schema,
  initial_value = data
)$as_json()




# Using resolve method to help with recursion
people <- list(
  "id_Barret" = list(name = "Barret", friends = list("id_Ryan", "id_Bill")),
  "id_Ryan" = list(name = "Ryan", friends = list("id_Barret", "id_Bill")),
  "id_Bill" = list(name = "Bill", friends = list("id_Ryan"))
)
schema <- gqlr_schema("
```

```
    type Person {
      name: String
      friends: [Person]
    }
    schema {
      query: Person
    }
  ",
  Person = list(
    resolve = function(name, schema, ...) {
      if (name %in% names(people)) {
        people[[name]]
      } else {
        NULL
      }
    }
  )
)

ans <- execute_request("{ name }", schema, initial_value = "id_Barret")
ans$as_json()

execute_request("
  {
    name
    friends {
      name
      friends {
        name
        friends {
          name
          friends {
            name
          }
        }
      }
    }
  }",
  schema,
  initial_value = "id_Barret"
)$as_json()
```

`gqlr_schema` *Create Schema definitions*

Description

Creates a Schema object from the defined GraphQL string and inserts the provided descriptions, resolve methods, resolve_type methods into the appropriate place.

Usage

```
gqlr_schema(schema, ...)
```

Arguments

schema	GraphQL schema string or Schema object
...	named lists of information to help produce the schema definition. See Details

Details

The ... should be named arguments whose values are lists of information. What information is needed for each type is described below.

ScalarTypeDefinition:

resolve function with two parameters: x (the raw to be parsed, such as 5.0) and schema (the full Schema definition). Should return a parsed value

description (optional) single character value that describes the Scalar definition

parse_ast (optional) function with two parameters: obj (a GraphQL wrapped raw value, such as an object of class IntValue with value 5) and schema (the full Schema definition). If the function returns NULL then the AST could not be parsed.

EnumTypeDefinition:

resolve (optional) function with two parameters: x and schema (the full Schema definition). Should return the value x represents, such as the Star Wars Episode enum value "4" could represent Episode "NEWHOPE". By default, EnumTypeDefinitions will return the current value.

description (optional) single character value that describes the Enum definition

values (optional) named list of enum value descriptions. Such as `values = list(ENUMA = "description for ENUMA", ENUMZ = "description for ENUMZ")`

ObjectTypeDefinition:

resolve function with two parameters: x (place holder value to be expanded into a named list) and schema (the full Schema definition). By using the resolve method, recursive relationships, such as friends, can easily be handled. The resolve function should return a fully named list of all the fields the definition defines. Missing fields are automatically interpreted as NULL.

Values in the returned list may be a function of the form `function(obj, args, schema) {...}`. This allows for fields to be determined dynamically and lazily. See how add_human makes a field for totalCredits, while the add_droid pre computes the information.

description (optional) single character value that describes the object

fields (optional) named list of field descriptions. Such as `fields = list(fieldA = "description for field A", fieldB = "description for field B")`

InterfaceTypeDefinition and UnionTypeDefinition:

resolve_type function with two parameters: x (a pre-resolved object value) and schema (the full Schema definition). This function is required to determine which object type is being used. resolve_type is called before any ObjectTypeDefinition resolve methods are called.

description (optional) single character value that describes the object

Examples

```
library(magrittr)

## Set up data
add_human <- function(human_data, id, name, appear, home, friend) {
 human <- list(id = id, name = name, appearsIn = appear, friends = friend, homePlanet = home)
  # set up a function to be calculated if the field totalCredits is required
  human$totalCredits <- function(obj, args, schema) {
    length(human$appearsIn)
  }
  human_data[[id]] <- human
  human_data
}
add_droid <- function(droid_data, id, name, appear, pf, friend) {
 droid <- list(id = id, name = name, appearsIn = appear, friends = friend, primaryFunction = pf)
  # set extra fields manually
  droid$totalCredits <- length(droid$appearsIn)
  droid_data[[id]] <- droid
  droid_data
}

human_data <- list() %>%
 add_human("1000", "Luke Skywalker", c(4, 5, 6), "Tatooine", c("1002", "1003", "2000", "2001")) %>%
 add_human("1002", "Han Solo",      c(4, 5, 6), "Corellia", c("1000", "1003", "2001")) %>%
 add_human("1003", "Leia Organa",   c(4, 5, 6), "Alderaan", c("1000", "1002", "2000", "2001"))

droid_data <- list() %>%
 add_droid("2000", "C-3PO", c(4, 5, 6), "Protocol", c("1000", "1002", "1003", "2001")) %>%
  add_droid("2001", "R2-D2", c(4, 5, 6), "Astromech", c("1000", "1002", "1003"))

all_characters <- list() %>% append(human_data) %>% append(droid_data) %>% print()
## End data set up



# Define the schema using GraphQL code
star_wars_schema <- Schema$new()

"
enum Episode { NEWHOPE, EMPIRE, JEDI }
" %>%
  gqlr_schema(
    Episode = list(
      resolve = function(episode_id, schema) {
        switch(as.character(episode_id),
          "4" = "NEWHOPE",
          "5" = "EMPIRE",
          "6" = "JEDI",
          "UNKNOWN_EPISODE"
        )
      }
    )
```

```
  ) ->
episode_schema
# display the schema
episode_schema$get_schema()
# add the episode definitions to the Star Wars schema
star_wars_schema$add(episode_schema)


"
interface Character {
  id: String!
  name: String
  friends: [Character]
  appearsIn: [Episode]
}
" %>%
  gqlr_schema(
    Character = list(
      resolve_type = function(id, schema) {
        if (id %in% names(droid_data)) {
          "Droid"
        } else {
          "Human"
        }
      }
    )
  ) ->
character_schema
# print the Character schema with no extra formatting
character_schema$get_schema() %>% format() %>% cat("\n")
star_wars_schema$add(character_schema)


"
type Droid implements Character {
  id: String!
  name: String
  friends: [Character]
  appearsIn: [Episode]
  primaryFunction: String
}
type Human implements Character {
  id: String!
  name: String
  friends: [Character]
  appearsIn: [Episode]
  homePlanet: String
}
" %>%
  gqlr_schema(
    Human = list(
      # Add a resolve method for type Human that takes in an id and returns the human data
       resolve = function(id, args, schema) {
```

```
        human_data[[id]]
      }
    ),
    Droid = list(
      # description for Droid
      description = "A mechanical creature in the Star Wars universe.",
     # Add a resolve method for type Droid that takes in an id and returns the droid data
      resolve = function(id, schema) {
        droid_data[[id]]
      }
    )
  ) ->
human_and_droid_schema
human_and_droid_schema$get_schema()
star_wars_schema$add(human_and_droid_schema)


"
type Query {
  hero(episode: Episode): Character
  human(id: String!): Human
  droid(id: String!): Droid
}
# the schema type must be provided if a query or mutation is to be executed
schema {
  query: Query
}
" %>%
  gqlr_schema(
    Query = function(null, schema) {
      list(
        # return a function for key 'hero'
        # the id will be resolved by the appropriate resolve() method of Droid or Human
        hero = function(obj, args, schema) {
          episode <- args$episode
          if (identical(episode, 5) || identical(episode, "EMPIRE")) {
            "1000" # Luke Skywalker
          } else {
            "2001" # R2-D2
          }
        },
        # the id will be resolved by the Human resolve() method
        human = function(obj, args, schema) {
          args$id
        },
        # the id will be resolved by the Droid resolve() method
        droid = function(obj, args, schema) {
          args$id
        }
      )
    }
  ) ->
schema_def
```

```
# print Schema with no extra formatting
schema_def$get_schema() %>% format() %>% cat("\n")
star_wars_schema$add(schema_def)


# view the final schema definitiion
star_wars_schema$get_schema()
```

`parse_ast` *Parse AST*

Description

This is a helper function for Scalars. Given a particular kind and a resolve function, it produces a function that will only parse values of a particular kind.

Usage

```
parse_ast(kind, resolve)
```

Arguments

`kind`	single character name of a class to parse
`resolve`	function to parse the value if the kind is correct

Details

Typically, `kind` is the same as the class of the Scalar. When making a new Scalar, parse_ast defaults to use the name of the scalar and the scalar's parse value function.

This function should only need to be used when defining a schema in `gqlr_schema()`

Value

function that takes `obj` and `schema` that will only parse the value if the `kind` is inherited in the `obj`

Examples

```
parse_date_value <- function(obj, schema) {
  as.Date(obj)
}
parse_ast("Date", parse_date_value)

# Example from Int scalar
parse_int <- function(value, ...) {
  MAX_INT <-  2147483647
  MIN_INT <- -2147483648
  num <- suppressWarnings(as.integer(value))
  if (!is.na(num)) {
```

```
    if (num <= MAX_INT && num >= MIN_INT) {
      return(num)
    }
  }
  return(NULL)
}
parse_ast("IntValue", parse_int)
```

Schema *GraphQL Schema object*

Description

Manages a GraphQL schema definition. A Schema can add more GraphQL type definitions, assist in determining definition types, retrieve particular definitions, and can combine with other schema definitions.

Typically, Schema class objects are created using `gqlr_schema`. Creating a `Schema$new()` object should be reserved for when multiple Schema objects are combined.

Usage

```
## using star_wars_schema from
# example(gqlr_schema)
star_wars_schema$get_schema()
star_wars_schema$is_enum("Episode") # TRUE
star_wars_schema$is_object("Episode") # FALSE
execute_request("{ hero { name } }", star_wars_schema)
```

Initialize

schema Either a character GraphQL definition of a schema or another Schema object. Extending methods and descriptions should be added with `gqlr_schema`.

The initialize function will automatically add

- Scalars: Int, Float, String, Boolean
- Directives: @skip and @include
- Introspection Capabilities

Details

`$add(obj)`: function to add either another Schema's definitions or Document of definitions. `obj` must inherit class of either `'Schema'` or `'Document'`

`$is_scalar(name)`, `$is_enum(name)`, `$is_object(name)`, `$is_interface(name)`, `$is_union(name)`, `$is_input_object(name)`, `$is_directive(name)`, `$is_value(name)`: methods to determine if there is a definition of the corresponding definition type for the provided name.

`$get_scalar(name)`, `$get_enum(name)`, `$get_object(name)`, `$get_interface(name)`, `$get_union(name)`, `$get_input_object(name)`, `$get_directive(name)`, `$get_value(name)`: methods to retrieve a

definition of the corresponding definition type for the provided name. If the object can't be found, NULL is returned. When printed, it quickly conveys all known information of the definition. Due to the nature of R6 objects, definitions may be retrieved and altered after retrieval. This is helpful for adding descriptions or resolve after the initialization.

`$get_scalars(name)`, `$get_enums(name)`, `$get_objects(name)`, `$get_interfaces(name)`, `$get_unions(name)`, `$get_input_objects(name)`, `$get_directives(name)`, `$get_values(name)`: methods to retrieve all definitions of the corresponding definition type.

`$get_type(name)`: method to retrieve an object of unknown type. If the object can't be found, NULL is returned. When printed, it quickly conveys all known information of the definition.

`$get_type(name)`: method to retrieve an object of unknown type. If the object can't be found, NULL is returned.

`$get_schema()`: method to retrieve full definition of schema. When printed, it quickly conveys all types in the schema.

`$get_query_object()`, `$get_mutation_object()`: helper method to retrieve the schema definition query or mutation object.

`$implements_interface()`: helper method to retrieve all objects who implement a particular interface.

`$is_valid`: boolean that determines if a Schema object has been validated. All Schema objects are validated at the time of request execution. The Schema will remain valid until new definitions are added.

See Also

gqlr_schema

---

server *Run basic GraphQL server with GraphiQL support*

Description

Run a basic GraphQL server using plumber. This server is provided to show basic interaction with GraphQL. The server will run until the function execution is canceled.

Usage

```
server(
  schema,
  port = 8000L,
  ...,
  graphiql = interactive(),
  log = TRUE,
  initial_value = NULL
)
```

Arguments

schema	Schema object to use execute requests
port	web port to serve the server from. Set port to NULL to not run the plumber server and return it.
...	ignored for paramter expansion
graphiql	logical to determine if the GraphiQL interface should be enabled. By default, this route is only available when running the server interactively.
log	boolean that determines if server logging is done. Defaults to TRUE
initial_value	default value to use in execute_request()

Details

To view the GraphiQL user interface, navigate to the URL provided when the server is started. The default location is http://localhost:8000/graphiql/. By default, this route is only available when running the server interactively (graphiql = rlang::is_interactive()).

server() implements the basic necessities described in http://graphql.org/learn/serving-over-http/. There are four routes implemented:

'/' GET. If run interactively, forwards to /graphiql for user interaction with the GraphQL server. This route is diabled if graphiql = rlang::is_interactive() is not TRUE.

'/graphiql/' GET. Returns a GraphiQL formatted schema definition interface to manually interact with the GraphQL server. By default this route is disabled if graphiql = rlang::is_interactive() is not TRUE.

'/graphql' GET. Executes a query. The parameter 'query' (which contains a GraphQL formatted query string) must be included. Optional parameters include: 'variables' a JSON string containing a dictionary of variables (defaults to an empty named list), 'operationName' name of the particular query operation to execute (defaults to NULL), and 'pretty' boolean to determine if the response should be compact (FALSE, default) or expanded (TRUE)

'/graphql' POST. Executes a query. Must provide Content-Type of either 'application/json' or 'application/graphql'.

If 'application/json' is provided, a named JSON list containing 'query', 'operationName' (optional, default = NULL), 'variables' (optional, default = list()) and 'pretty' (optional, default = TRUE). The information will used just the same as the GET-'/graphql' route.

If 'application/graphql' is provided, the POST body will be interpreted as the query string. All other possible parameters will take on their default value.

Using bash's curl, we can ask the server questions:

```
#R
 # load Star Wars schema from 'execute_request' example
 example(gqlr_schema)
 # run server
 server(star_wars_schema, port = 8000)
```

```
#bash
 # GET Schema definition
 curl '127.0.0.1:8000/'

 ## POST for R2-D2 and his friends' names
 # defaults to parse as JSON
curl --data '{"query":"{hero{name, friends { name }}}", "pretty": true}' '127.0.0.1:8000/graphql'
 # send json header
curl --data '{"query":"{hero{name, friends { name }}}"}' '127.0.0.1:8000/graphql' --header "Content-T
 # send graphql header
curl --data '{hero{name, friends { name }}}' '127.0.0.1:8000/graphql' --header "Content-Type:applicat
 # use variables
curl --data '{"query":"query Droid($someId: String!) {droid(id: $someId) {name, friends { name }}}", "

 # GET R2-D2 and his friends' names
 curl '127.0.0.1:8000/graphql?query=
 # ... using a variable
 curl '127.0.0.1:8000/graphql?query=query
```

Index

as_R6, 2

ErrorList, 3
execute_request, *3*, 4, *14*

gqlr_schema, *4*, 6, *11–13*

parse_ast, 11

Schema, 12
server, 13