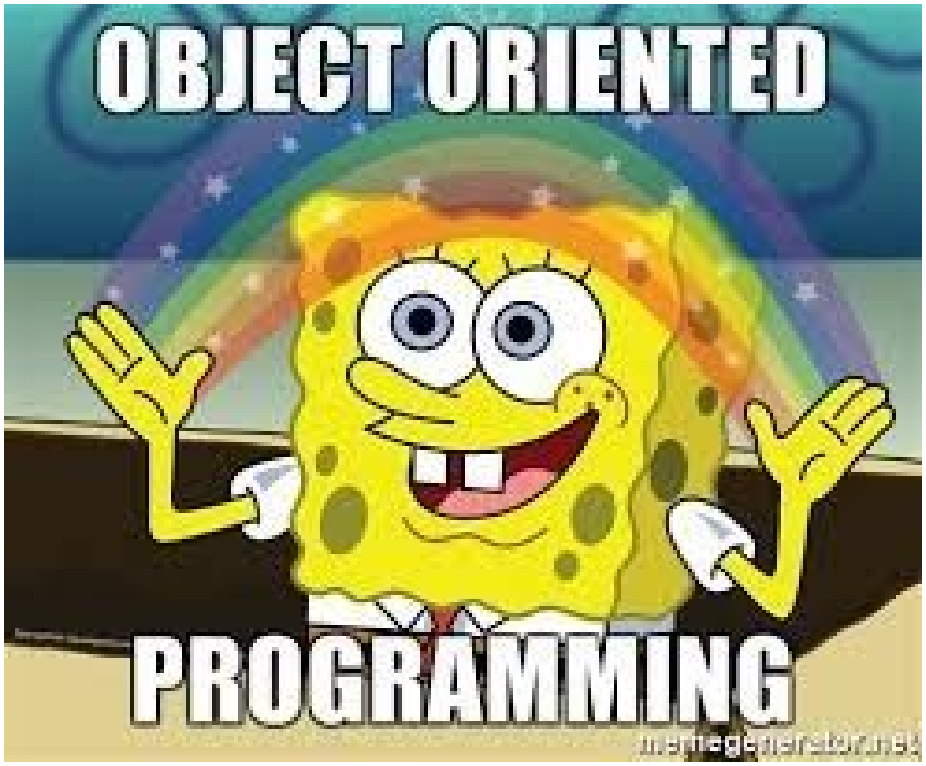
Course Project - GraphQL Client Framework



Emily Lin,

Caglar Kurtkaya,

Sakina Master, and

Aleksandar Knezevic

CS474

Spring 2020

Table of Contents

Project Overview

In this project, we will be creating a pure functional object-oriented framework for composing and executing external GraphQL commands using Scala. The Scala version we are using in the project is 2.13.1. In this project, users will be using our framework for generating the information of GitHub repositories. Users can define the languages, how many repositories they want to check, set how many commit comments they want to see and search the repositories based on REPOSITORY or USER. The result will be saved in the output.txt file in a format that is easy for users to see.

Project Design

Project Model

In this project, we used the sbt which is an open-source build tool for Scala and Java. The reason why we choose to use sbt is because it's easier to add dependencies for the project. The dependencies we used for implementing this project include typesafe logging, scala test, play-json parser, httpclient, json4 and typesafe config. We also include nine test cases for testing whether the framework is working properly. As the result of test cases, all the tests pass.

The project is composed of four packages:

- Builder_pattern

- Filter_objects
- Parser
- Queries

The builder_pattern package has classes which build GitHub connection objects, query objects and queries. Filter_objects has all objects that can be used to filter out the queries. The above two are explained in more detail under Design Pattern Usage. The parser package has two classes named ParseRepoResponse and ParseUserResponse. These classes extend LazyLogging. Those classes will parse the json object into a list of repositories. The Queries package contains the USER repo and REPOSITORY repo. Depending on what type of query the user wants to search, the corresponding query will be called.

Design Pattern Usage

Builder Pattern :

The implementation of GraphQL client framework uses builder pattern for :

- Establishing connection to GraphQL server
- Building search query for repository and user
- Building several case class objects of a query such as Repository, User, CommitComments and PullRequest

The GitHub.scala uses a type-safe builder pattern with phantom types using which we can create a basic HTTP client for GitHub GraphQL endpoint. The phantom types are defined by using sealed traits, which restricts them to be extended only in the same file as its declaration. New GitHub objects are returned in each method with the parametric type extended with what's computed in that specific method. Methods are restricted by implicit evidence which binds them to a specific GithubObj and preserves the structure. The build method uses implicit to check if the GitHubObj is a CompleteGitHubObj and it then returns a **GHQLResponse(httpUriRequest, Client)**. By using phantom types and implicits a specific call order is enforced, and a strongly type checked object is built.

Similarly, Command.scala and CommandUser.scala build queries for querying repositories and users respectively. These classes use a type-safe builder pattern with phantom types defined by using sealed traits as well. In Command.Scala, some initializations such as **commitComments: (String, Int), pullRequests: (String, Int)** are assigned default values. Hence, they are optional and calling those respective methods is left to the choice of user. However, some assignments are required, and implicit evidence is used to preserve the structure of the calls made to build the object. The build method uses implicit to check if the QueryCommand is a CompleteQueryCommand and it then returns an Option**[List[Repository]].**

Repository, User, CommitComments and PullRequest are defined as case classes with default parameters. These objects do not require any validation and hence they are not built using type-safe builder patterns as above, but instead built using case classes.

The parameters of these case classes are immutable, and some of the parameters are defined using Option[].

Singleton Pattern:

The implementation of GHQL client framework uses singleton pattern for :

- The objects that can be used to filter the repository and user queries by int values
- The repository and user query objects

Commits, ContribRepos, Followers, Following, Forks, Repos, Stars and Watchers are declared as **objects** inside the package **filter_objects**. These objects are used to filter the repository and user queries by int values. All of these objects extend the class FilterObjects. These objects are instantiated as input parameters for filter().

For example :

filter(Stars(“>60000”)).filter(Forks(“>32000”)).filter(Watchers(“<4000”))

Since filter() takes a predicate function as its parameter and returns a new collection with elements that match the predicate, the objects instantiated should return a predicate function as well. All of these objects define an apply method which takes the userInput as input parameter and returns a predicate function of type: **Repository => Boolean.** The userInput gets parsed in the method predicateForFilter() of the class FilterObjects, and it returns a Boolean value for each Repository object in the collection.

The **RepoQuery and UserQuery** in the package **queries** are declared as objects as well. These objects define an apply method which takes the variables that need to be assigned in the query as input parameters and returns the complete query as a String object which is then sent to the GraphQL server.

The **Driver.scala** class where the control of the program starts is defined as an object as well.

Constraints and Rules

The GitHub connection objection is to be built as following :

new GitHub[EmptyObj].setHttp().setAuthorization(name, token).setHeader(name, jsonValue).build

It is required that all of these methods are called and the order of calls cannot be changed else it will result in a compile time error.

The repository query is formed as follows :

newCommand[QueryCommand].setRepo("REPOSITORY").setLanguages(["Java", "C++"]).setFirst(5).build()

The user is required to call all the above methods in the order shown above for querying repositories. Apart from these methods, the user can chain additional methods to set commitComments, pullRequests and issues. However, these methods should be called after .setFirst(IntVal).

For example :

newCommand[QueryCommand].setRepo(“REPOSITORY”).setLanguages([“Java”, “C++”]).setFirst(4).setCommitComments(“first”, 5).setPullRequests(“last”, 5).build()

The user query is formed as follows :

newCommandUser[QueryUserCommand].setUser(“USER”).setName(“emily”).setFirst(5).build()

The user is required to call all the above methods in the order shown above for querying the user.

The user can then chain filter() to the above query after .get(). The .get() extracts the List[Repository] from Option[List[Repository]]. The filter() takes the object which needs to be filtered along with a string input of the value we need to filter by. An example is shown below :

filter(Stars(“>60000”)).filter(Forks(“>32000”)).filter(Watchers(“<4000”))

Drawbacks

In the beginning of implementing this project, we weren’t that familiar with the Scala syntax. Therefore, we spent a lot of time reading the book, searching what phantom type is and how to use it and how to implement and design a pure functional program. The biggest challenge of completing this project is to come up with a design that not only meets the requirements but also concise. The few drawbacks of this projects are :

- The pagination technique which can get all the repositories has not been implemented. Hence, it is restricted with getting only the first 100 repositories.
- We did not wrap the Unit functions to IO[Unit]. Hence, the design is not completely pure.

Index

SBT : https://www.scala-sbt.org

GitHub Developer : https://developer.github.com/v4/

GraphQL : https://graphql.org

GitHub GraphQL API : https://developer.github.com/v4/explorer/

Scala API : https://docs.scala-lang.org/api/all.html

Scala Test : http://www.scalatest.org/install

Scala Logging : https://github.com/lightbend/scala-logging

Scala Configuration File : https://github.com/lightbend/config

Phantom Type :

https://medium.com/@maximilianofelice/builder-pattern-in-scala-with-phantom-types-3e29a167e863