Software Requirements for ReunionLog

Reunion

LaTeX

Lead Programmer and Lead Designer - Nickgismokato

Design contributor - Bj

Design contributor & tester - Casper

Written by
Nick Laursen

Preface

This is the documentation of the requirements we want to follow when creating the program ReunionLog. We will be following the SOLID principles. This software is a program meant for the guild Reunion in the game World Of Warcraft. This software will use the API from WarcraftLogs.

Most of these requirements have been gathered from multiple months of pre-gathering data from the WarcraftLogs API. This has been done by creating a *proof-of-concept* program with Python.

Contents

Chapter 1

WarcraftLogs and the API

Abstract

This chapter will go through WarcraftLogs and the API correlating. For this chapter we will not go through the actual documentation but rather give a short refer to the documentation and lay out the most important aspect from the site and the API

1.1 Website

The website for WarcraftLogs[1] is a popular website used by guilds to gather data to one single site. This is done though multiple addons and their own in-house software.

1.1.1 Overall form

The data than can be collected is both in the form of Guild data, Raid data, Dungeon data, Character data and much more. These can be accessed for all through the website. This can be done by everyone anonymously.

The website uses graphQL to display most of their data. Both in tables both also in graphs and tables containing graphics. This gives an easier overview for most users. They to also have some options you can choose for the specific data you want to be showed.

A downside to this approach is that a lot of the specific options is not showed. One could reason the "*simpler*" design is because they want all users to use their website, no matter their technical background.

1.1.2 Specific to our needs

What described in **Section 1.1.1** sound really great and reasonable useful for most users case. This is indeed the case for most users, but if you want the information not available on the site or you want another way to represent the data, then the site is not for you. This is why we will be using the data given to us by the API.

The only need we have of the website is to check if the information we get is also the information displayed on the site.

[1] https://www.warcraftlogs.com/

1.2 Documentation for the API

The API and it's documentation is, for a lack of a better word, idiotic made. There exists two API's. Version 1 and version 2. We will be using the latter. This version has "*better*" documentation than it's counterpart and uses OAuth 2.0 for it's API authentication. The documentation can be found two places. For authentication documentation you can find it at Warcraftlogs 2023 and you can find the actual command call documentation at WarcraftLogs and Graphql 2023.

1.2.1 GraphQL

The first thing we should worry about is the authentication. As mentioned in the **Preface** we have already made successful connection, therefore this will be discussed later.

GraphQL is the schema of how to make calls to the API. This is done by sending you authentication and a "*Query*" call. This is in simple terms just a string with specific data. This data is both used to tell WarcraftLogs server what you would like to receive bout also where in the schema the server would have to lookup this data. Clearly there is a need to create these query strings.

Therefore we will be using GraphQL library from Graphql-dotnet. More information can be found at GraphQL 2023. If you were to look at WarcraftLogs and Graphql 2023 you would find there is a lot of commands you can send. Therefore it is important to create an object which can be agile and create all the necessary strings which we will be using.

1.2.2 Limitation of the API

When it comes to the API a lot of limits start to show. Some in the documentation and some in the actual API itself. Let us start talking about the API itself. When on a free tier, i.e not subscribed to WarcraftLogs website, you can at maximum make 3.600 calls to their server pr. hour. This inherently doesn't sound that bad, but you will later see in depths why this is a bad system.

I will agree that this stops most novice DDOS attacks to their server, but since the introduction of the second version of their API, multiple clients can easily be set up and could potentially help in the DDOS attacks. Let us now explain why this rate limit is bad when you want to make a software that pulls a great amount of data.

Let us say you had a guild that was raiding and you both raided Mythic and Heroic during the same log. Let us furthermore assume you want the data for when they die and how many pulls they were on, but only for the mythic. By scouring through their schema documentation (WarcraftLogs and Graphql 2023), you find the `Report` object inside the documentation. This is essentially what you want. So you create a query string with `report` as first argument and and then you scour through the documentation and find the `event` option which deliver a `ReportEventPaginator` which is just a list event in the log file. This gets returned as data in form of a JSON string. Let us see the documentation for it:

> "A set of paginated report events, filterable via arguments like type, source,t arget, ability, etc. This data is not considered frozen, and it can change without notice. Use at your own risk." (WarcraftLogs and Graphql 2023)

This object has an argument you can filter from called `difficulty`. The documentation for this is as follow:

> "**difficulty**: Optional. Whether or not to filter the fights to a specific difficulty. By default all fights are included." (WarcraftLogs and Graphql 2023)

This is brilliant. We can now filter our `Event` to mythic only. Later down the line is argue that `difficulty` is an integer value. Great so now we just need to figure our what integer value. This is the first hurdle we found when trying this. No actual documentation is given for the arguments, except if it is an object or enum type which they have created. So we set up in Python a script to run through with value n where $n \in \{-1000, -999, \ldots, 999, 1000\}$. Then we checked if there where any difference between the `JSON` strings we received and we got that there was no difference for any n value.

Clearly any user would be confused. Why would an `API` give you an option and then not document said option and actually do nothing with this option. Your first instinct would now be to contact WarcraftLogs. So you write them an email and there response they send you back can be seen in fig. A.1. So you go to their discord find the thread about the `API` and ask you question. The intention of this documentation is not to out anyone from their support team and therefore I will not share the conversion I had. But trust us when we say that if was probably the most useless information and support we have ever received.

So how did we fix it, you may ask? We looked at the documentation and again read what was written under the `difficulty` argument. The difficulty argument says it filter fights. So we then took a look at the `ReportFight` object. We tested this and sure enough it worked. So to simplify what we now have to do if we wish to get the event data only for Mythic. First we need to get all the fight data that had that difficulty i.e the `fightIDs`. This is done with the `ReportFlight` object. Then we have to sort our `ReportEventPaginator` to the `fightIDs`. Now we have used two calls of our 3.600 rate limit.

In our demo in Python it became clear that we also had to pull a lot more data than just fight i.e pulls and so on. We now have to pull every single fiights, every single name from that fight and the event itself. This gives us a rough estimate:

$$l \cdot (a_{\text{Fights}} + a_{\text{Event}} + a_{\text{name}} \cdot p)\,,$$

where p is the amount of players in any fights and l is the amount of different logs we want to pull data from. Since we cannot get `fightIDs` whilst also getting the event to only get data from the mythics, we would need to do this over two separate calls and also we cannot pull names just from the event log the way we want it, so we have to do it separately.

This is probably the most irritating limit of the `API` and therefore we would need to make sure our software can handle this limit in case our request limit get met.

1.3 Integration from the `API` to C#

As discussed in 1.2 it is clear that we would need many objects to handle everything involving WarcraftLogs `API` and it's authentication. Clearly we would need objects handling our query strings, *all of them*, handling of the authentication and most importantly handling the data we receive after we have made a call, i.e the response.

1.3.1 Authentication

Creating an authentication for our software is actually quite easy. We don't need any libraries for the creation of the code or the requester. The only thing we need is an users `client_id` and `client_secret`. The `tokenURL` is a constant i.e

"*https://www.warcraftlogs.com/oauth/token*". Everything else is user input. This is also important but will be dealt with differently.

We will need the user to create the `client_id` and `client_secret` which we will save as local data for further use. This will be done at first time use of the software.

1.3.2 Query Strings

When dealing with query strings within this API, you are in fact dealing with a `json` call within the GraphQL schema. As noted in **Section 1.2.1 GraphQL**, we will be using the `GraphQL` library from `Graphql-dotnet` (GraphQL 2023). This library is also under the MIT license, so no issue in license agreements here. It is important to note that Facebook created GraphQL. This library is just an implementation to .NET. You can easily install it by the following command:

```
$ dotnet add package GraphQL
```

This is the only package needed for this library. We don't need serialization or document cashing since we will implement it ourself if needed.

1.3.3 Data extracted from the API

After we have made a call, you would think we are done. Clearly this isn't the case. The JSON is what our call return. This data is not just a "pure" JSON string. It has layers within its own index. Therefore we need to go through these index layers if we wish to get to the actual data we need. This could either be implemented in the call function or in it's own class.

We will implement this within its own class. The main reason is we don't want to give the call object that responsibility. That lies solely within the respected event objects.

Chapter 2

Our software main requirements

Abstract

In this chapter we will go through all our requirements for our software. This will be done in a list form. This chapter will not be dealing with the responsibilities within the objects.

2.1 Authentication

The authentication part of the software has the following requirements:

- Create an authentication call
- Use the `client_id` and `client_secret` for the authentication call
- Create some `.credential.JSON` file to hold the credentials we get from authentication call
- If multiple calls is made to the `API`, it should be able to use the `.credential.JSON`
- Delete `.credential.JSON` when program is stopped.

Further more, it should have/use the following:

- Be a static class, maybe with some other type of classes (depends of the rest of the design).
- Check if `client_id` and `client_secret` is given. If not then it should prompt a creation page
- Hold a boolean value `createdCredentials` and it should use it to check whether or not the `.credential.JSON` has been created or exists
- Have a custom exception if authentication failed and should print the exception
- If an error was returned from the `API` it should also handle the error with an exception.

2.2 Events

2.3 Query Strings

The query string part has the following requirements:

- Create a `JSON String` type to cast to the event call of the software
- Handle different Master type objects from the `API`
  - This should be handled with a switch and using an `enum` object.
  - This should handle any error with an exception if wrong or non existing arguments is given.
- It will never hold the `JSON String` type. Only create and then return it

Further more, it should have/use the following:

- Use `enum`'s to handle different `API` types
  - The `enum`'s should be contained in the same `namespace` as the query string or should have it own
  - The `enum`'s value holder should not be used
  - The default value of the `enum`'s should return either `null` or some other empty value types
- It should be as agile as possible i.e it should follow the O principle in SOLID
- If it need to handle mutation of strings or value not a query, it should not handle it but instead call a function that mutate it
- It has to throw exceptions if either
  - Important query values is missing
  - default value from enums is received
  - Any errors are thrown
- No default query string should be returned if any error or exception is detected. It should instead return `null`.

2.4 .csv file

The most important part of the `.csv` file is not that we can use the software you change the design but instead that it is consistent with what design is used if multiple (*or single*) query's are used.

It is important that this software is meant for the Reunion guild and not for all guilds that want to use this software. That being said, we would still make the design part of the `.csv` handler/creator easy to redo and redesign on. Therefore we would have:

- The creation of the `.csv` files should be handled by a static object called `CreateCSVFile`
- This object should not hold any data, but instead be given the necessary date from the event object after it has received the data.
- It shouldn't have the need to manipulate the data but instead send the data of to me mutated into the necessary data needed to create the `.csv` data

Note: This is why we use a static class since we should be able to create multiple `.csv` files in one run or multiple runs where no exist of the software has happen. This is also more memory safe for the program

Note: Best practice would dictate us to create an `eventHandler` to handle these multiple calls but since this is a static class living in the `Main` class of our program, no real usage or performance gain would come of such practices

- If it encounter an error for any reason, no `.csv` files should be created. Instead a `.log` file should be created and show where in the process it was interrupted

Note: This log should be created every time but should be deleted if no exception was thrown or no error was detected

Note: A debug option should exists in the software to determine whether or not the `.log` file should be deleted after successfully handling a creation call

Note: If option to not delete the `.log` files, then it need to call the different files some different from each other. The reason is we don't want to overwrite previously written `.log` files and we want a way to distinguished the different files.

Chapter 3

Flow of ReunionLog

Abstract

In this section we will go through the simple flow of how the software will behave. This is both in terms of how the software access the API and how many times it does but also how calls for object is being handled. Clearly there is a lot to talk about so let us start with an simple overview before diving into the abstract parts of the software.

Clearly, this section in not meant as a substitution for an actual UML diagram and responsibilities table. It is merely meant to give the reader a better understanding of the software before diving into it. Since no actual standard is made for this section, I will try to do my best to come around the software.

3.1 Simple Overview

3.2 Needs for each main Requirements

Chapter 4

Responsibilities for our requirements

Abstract

Appendix A

Figures

A.1 Pictures

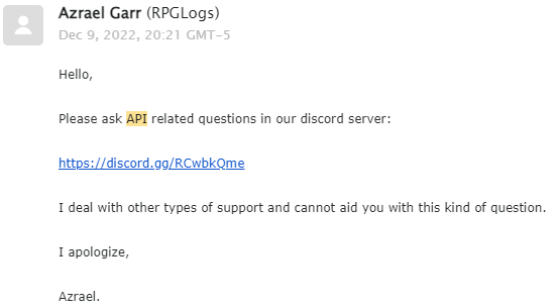

Figure A.1: Mail received from the support team at WarcraftLogs

Appendix B

Schema

B.1 Overview

Main Call: query

- characterData: CharacterData
- gameData: GameData
- guildData: GuildData
- progressRaceData: ProgressRaceData
- rateLimitData: RateLimitData
- reportData: ReportData
- userData: UserData
- worldData: WorldData

B.2 `CharacterData`

B.3 `GameData`

B.4 `GuildData`

B.5 `ProgressRaceData`

B.6 `RateLimitData`

B.7 ReportData

- reportData
  - report(code: String): Report
  - reports(endTime: Float,
    guildID: Int,
    guildName: String,
    guildServerSlug: String,
    guildServerRegion: String,
    guildTagID: Int,
    userID: Int,
    limit: Int,
    page: Int,
    startTime: Float,
    zoneID: Int,
    gameZoneID: Int): ReportPagination

B.7.1 Report

- report
  - code: String!
  - endTime: Float!
  - events(abilityID: Float,
    dataType: EventDataType,
    death: Int,
    difficulty: Int,
    encounterID: Int,
    endTime: Float,
    fightsIDs: [Int],
    filterExpression: String,
    hostilityType: HostilityType,
    includeResources: Boolean,
    killType: KillType,
    limit: Int,
    sourceAurasAbsent: String,
    sourceAurasPresent: String,
    sourceClass: Int,
    sourceID: Int,
    sourceInstanceID: Int,
    startTime: Float,
    targetAurasAbsent: String,
    targetAurasPresent: String,
    targetClass: String,
    targetID: Int,
    targetInstanceID: Int,
    translate: Boolean,
    useAbilityIDs: Boolean,
    useActorIDs: Boolean,
    viewOptions: Int,
    wipeCutoff: Int): ReportEventPaginator
  - exportedSegments: Int!

```
- fights(difficulty: Int,
  encounterID: Int,
  fightIDs: [Int],
  killType: KillType,
  Translate: Boolean): [ReportFight]
- graph(abilityID: Float,
  dataType: GraphDataType,
  death: Int,
  difficulty: Int,
  encounterID: Int,
  endTime: Float,
  fightsIDs: [Int],
  filterExpression: String,
  hostilityType: HostilityType,
  killType: KillType,
  sourceAurasAbsent: String,
  sourceAurasPresent: String,
  sourceClass: Int,
  sourceID: Int,
  sourceInstanceID: Int,
  startTime: Float,
  targetAurasAbsent: String,
  targetAurasPresent: String,
  targetClass: String,
  targetID: Int,
  targetInstanceID: Int,
  translate: Boolean,
  viewOptions: Int,
  viewBy: ViewType,
  wipeCutoff: Int): JSON
- guild: Guild
- guildTag: GuildTag
- owner: User
- masterData(translate: Boolean): ReportMasterData
- playerDetails(difficulty: Int,
  encounterID: Int,
  endTime: Float,
  fightIDs: [Int],
  killType: KillType,
  startTime: Float,
  translate: Boolean): JSON
- rankedCharacters: [Character]
- rankings(compare: RankingCompareType,
  difficulty: Int,
  encounterID: Int,
  fightIDs: [Int],
  playerMetric: ReportRankingMetricType,
  timeframe: RankingTimeframeType): JSON
- region: Region
```

- revision: Int!
- segments: Int!
- startTime: Float!

B.7.2 ReportPagination

B.8 UserData

B.9 WorldData

Bibliography

GraphQL. 2023. “graphql-dotnet.” https://github.com/graphql-dotnet/graphql-dotnet.

Warcraftlogs. 2023. “Docs.” https://www.warcraftlogs.com/api/docs.

WarcraftLogs and Graphql. 2023. “Graphql schema documentation.” https://www.warcraftlogs.com/v2-api-docs/warcraft/.