RestMule: Enabling Resilient Clients for Remote APIs

Beatriz A. Sanchez
Department of Computer Science
University of York
York, UK
basp500@york.ac.uk

Konstantinos Barmpis
Department of Computer Science
University of York
York, UK
konstantinos.barmpis@york.ac.uk

Patrick Neubauer
Department of Computer Science
University of York
York, UK
patrick.neubauer@york.ac.uk

Richard F. Paige
Department of Computer Science
University of York
York, UK
richard.paige@york.ac.uk

Dimitrios S. Kolovos
Department of Computer Science
University of York
York, UK
dimitris.kolovos@york.ac.uk

ABSTRACT

Mining data from remote repositories, such as GitHub and StackExchange, involves the execution of requests that can easily reach the limitations imposed by the respective APIs to shield their services from overload and abuse. Therefore, data mining clients are left alone to deal with such protective service policies which usually involves an extensive amount of manual implementation effort. In this work we present RestMule, a framework for handling various service policies, such as limited number of requests within a period of time and multi-page responses, by generating resilient clients that are able to handle request rate limits, network failures, response caching, and paging in a graceful and transparent manner. As a result, RestMule clients generated from OpenAPI specifications (i.e. standardized REST API descriptors), are suitable for intensive data-fetching scenarios. We evaluate our framework by reproducing an existing repository mining use case and comparing the results produced by employing a popular hand-written client and a RestMule client.

CCS CONCEPTS

• **Information systems** → **RESTful web services**; • **Software and its engineering** → *Reusability*; *Error handling and recovery*;

KEYWORDS

Resilience, OpenAPI Specification, HTTP API Clients

ACM Reference Format:
Beatriz A. Sanchez, Konstantinos Barmpis, Patrick Neubauer, Richard F. Paige, and Dimitrios S. Kolovos. 2018. RestMule: Enabling Resilient Clients for Remote APIs. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3196398.3196405

MSR'18, May 2018, Gothenburg, Sweden

1 INTRODUCTION

Research in the area of Mining Software Repositories (MSR) [6] investigates challenges associated with handling information that originate from public software development data sets, source code repositories, Q&A knowledge bases, requirements and issue tracking systems, and are commonly offered by service providers in form of remote APIs.

Service providers commonly constrain the access to public data sets in term of service policies to protect their services from saturation and abuse. For example, such protective measures may be implemented by imposing a limit to the number and rate of client requests, returning multi-page responses, and black-listing abusive clients. However, there are a number of research applications, such as evaluating the popularity of software technologies [7, 9], that require the execution of a number of requests to remote APIs that easily exceed rate limits imposed by service provides. Consequently, as a result of requiring clients to handle service provider policies during the execution of their collection and analysis tasks, their implementation is considered to be a cumbersome and time-consuming activity [5].

Although some service providers offer client libraries for their services in a variety of programming languages and thus greatly reduce implementation effort by providing an abstraction from low-level remote API behavior, such as informal client-server interaction contracts over the native HTTP protocol, such a library usually (*i*) does not address server-side protection mechanisms, such as request throttling, (*ii*) requires extensive manual implementation effort during its development and maintenance, and (*iii*) is limited by the functionality and behavior exposed through its operations.

Research in the area of MSR has brought to light a set of reusable frameworks, such as BOA [4], as well as replications of public repositories, such as GHTorrent [5], the Stack Exchange's Data Dump [2] and the Maven Repository Dataset [12], that enable bypassing service provider policies. However, employing data sets mirroring large public repositories exposes several challenges, like the requirement of (*i*) maintaining a large infrastructure to import, store, maintain, and provide access to mirrored repository information, (*ii*) handling inconsistencies in replicated data sets [5], and (*iii*) dealing with outdated information.

The work presented in this paper offers an approach, which has been implemented in terms of the RestMule framework, that

addresses several repository mining challenges by providing a solution to semi-automatically generate resilient clients from standardized REST API specifications. In general, our approach generates executable clients for remote APIs that are formally defined in terms of OpenAPI [10] specifications, which has been proposed as a machine-readable format for describing the architecture of services offered as RESTful remote APIs [11]. Moreover, resilience is achieved by specifying service policies, such as rate limits and pagination, and handling them, as well as network failures and response caching, accordingly and in a graceful and transparent manner.

Roadmap. The rest of the paper is structured as follows. Section 2 presents our approach alongside its architecture and design within the RestMule framework. Section 3 demonstrates the evaluation of our framework by reproducing an existing repository mining use case based on GitHub and comparing the respective results produced by employing a prominent hand-coded client and a RestMule-generated client. Section 4 compares with existing literature and frameworks in the area of intense data collection and Mining Software Repositories, in general. Section 5 concludes our work by summarizing findings and highlighting future work.

2 APPROACH

This section presents our approach on semi-automated generation of resilient clients by focusing on the design and architectural components of its implementation within the RestMule framework[1].

2.1 Architecture and Design

The RestMule framework is designed on a layered architecture to reduce the coupling between the two main components, i.e., RestMule core and RestMule codegen, as well as to reduce the amount of generated code.

In general, the RestMule core component, c.f. circle 1 in Fig. 1, and the RestMule codegen component, c.f. circle 2 in Fig. 1, act as general purpose layer and API-specific layer, respectively. Moreover, the general purpose layer encapsulates the functionality that can be shared across different API-specific components. The RestMule API client component, c.f. circle 3 in Fig. 1, is responsible to bridge the gap between the functionalities offered by the RestMule core component and API-specific RestMule-generated clients. RestMule has been designed to return Data Access Objects (DAOs) that handle the different types of successful HTTP response payloads as well as provide insights to the data acquisition status. The entities responsible for submitting requests to the service provider are wrappers of plain API requests. Furthermore, these entities (*inner clients*) offer a list of services that encapsulate and handle various aspects of the system, such as API-specific request-limits and pagination. The RestMule framework defines a number of inner clients that is equal to the number of rate request limit policies implemented by the service provider. For example, GitHub implemented two different request limit policies for two different groups of HTTP endpoints, i.e., one for those starting with `/search/*` and one for all other endpoints, and thus requires defining two inner clients to handle both groups of endpoints appropriately.

[1]https://github.com/beatrizsanchez/RestMule

Inner clients are associated to user *sessions*, i.e., employed to authenticate HTTP requests, and may affect the request rate limit value of individual endpoint groups. For example, in GitHub the aforementioned `/search/*` endpoints (only) allow 10 request per minute to a public session and 30 requests per minute to an authenticated session. RestMule abstracts multiple inner clients by providing users with a single *entrypoint* that offers services and delegates their execution to any appropriate inner client. Moreover, in case a response is available as a valid cache entry, no request is issued to the service provider.

Resilient Client Generator. In general, the resilient client generator, c.f. circle 2 in Fig. 1, takes an OpenAPI Specifications (OAS) in JSON as well as a service policy description as input and produces a *RestMule model* that conforms to the *RestMule metamodel*. In more detail, the service policy description is represented by an Epsilon Object Language (EOL) [8] script and contains information, such as pagination and rate-limits. Finally, the *RestMule model*, which conforms to the *RestMule metamodel*, c.f. Fig. 2 for a simplified version, is consumed by the *M2T* transformation for the generation of the resilient Java client, c.f. circle 3 in Fig. 1. The generated code is structured as an Eclipse Plugin[2] and can be employed as Java library by third party-applications. In the following we describe functionalities and behaviors that are shared among RestMule-generated clients.

Data Access Objects (DAOs). In addition to simple types, RestMule API client can handle three types of HTTP response payloads: single objects, arrays of objects, and objects containing pointers to data needing to be fetched (wrappers). The latter two may contain information regarding limitations on the number of items to be provided or stored, specially if they are in disagreement with the total count returned by the service provider. For example, GitHub returns details of a maximum of 1000 items as well as 100 items per page but its response may indicate that a total of 2000 items have been found, i.e., addressed by the RestMule API client in terms of supporting capped results.

The data returned by the DAOs implements the Observer interface (from RxJava[3]). Both types of aforementioned DAOs return ReplaySubjects, which keep a copy of the data pushed to its subscribers such that all observers can get the same data upon request.

Pagination and Wrappers. To manage paged responses from the remote sources, two components are used: page traversal and response body wrapping. For the page traversal strategy, RestMule core is used by RestMule API client, with the API-specific pagination parameters provided and relevant methods defined. Response body wrappers wrap appropriate responses to provide common accessors to be used within the page traversal methods. As various sources may define different collection schemas, they are API-specific. For example the one for GitHub provides the mapping to the specific JSON fields that GitHub provides.

As data is retrieved through callbacks, the handling of these responses (successful or unsuccessful – of asynchronously-sent requests) for different pages that are associated to the same result

[2]https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

[3]https://github.com/ReactiveX/RxJava

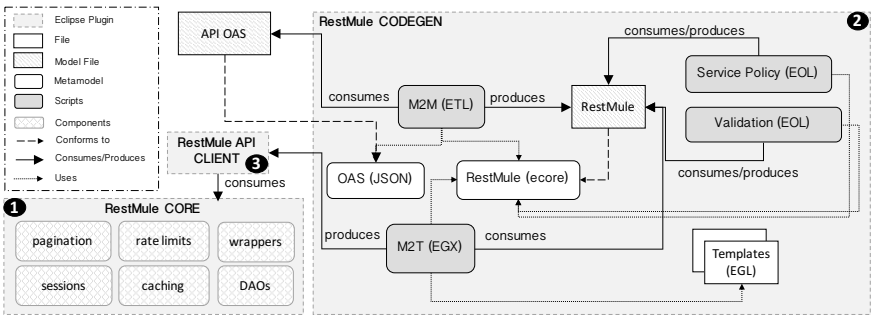

Figure 1: RestMule Architecture

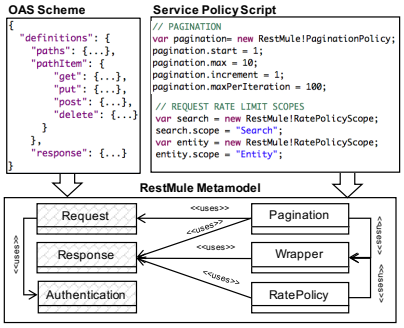

Figure 2: Simplified *RestMule Metamodel*

data set is needed. RestMule uses the abstractions provided by the OkHttp library[4], which deals with asynchronous response types.

Sessions. They are currently used within the HTTP interceptors context to update the request rates available to the user every time a new response is received from the Internet; this verifies that a given session has a non empty request allowance before dispatching them to the network. The relevant generated API-specific classes provide methods to create sessions based on the supported authentication schemes offered by the service provider. These methods return a session interface which can be used to access its request limits.

Inner Clients and Main Entrypoint. The RestMule API client layer exposes API calls that hide pagination and request limit handling to the end-user and return Data Access Objects. This is achieved by delegating the API calls to inner clients that use different request dispatcher engines to deal with specific request-limit policies regardless of the authentication scheme used to identify the user. Based on the pagination policy employed by the service-provider (e.g. page numbered, limit-offset), the inner clients know how to traverse the pages of a multi-part response from the server. The request dispatcher engine will monitor the request allowance based on the user session and await for allowances to be refilled before sending further requests to the network.

The main entrypoint for the user is a facade to the exposed methods of the multiple inner clients; it is responsible for instantiating the inner clients based on the configuration that the user passes to its builder (e.g. allow caching, user session).

Caching. This is defined in the core layer; API-specific extensions are generated which define how to load and put index entries that represent HTTP responses. Since the OkHttp library used by RestMule offers reliable caching capabilities, these are the ones currently used by the system. If more fine-grained control of the cache is desired though, RestMule allows for the incorporation of a custom manager instead.

3 EVALUATION

In order to gain confidence that RestMule is able to offer its intended capabilities, a two-way evaluation has been performed between RestMule and a popular Java-based GitHub API client[5], i.e., referred to as GAJ in the sequel, using the methodology of a published case-study. This evaluation aims at reproducing the experiment methodology published in previous work that assessed the use of MDE technologies [9], using RestMule. In that work, hand-written imperative code in a Javascript-like language was used to obtain data from GitHub regarding the use of 18 different model-driven engineering technologies. The resulting data comprised (*i*) the number of repositories using these technologies, (*ii*) the number of files in those repositories that contained code written in the relevant MDE language, and (*iii*) the commits and authors of those files.

Method. To evaluate the functionality of RestMule, we generated a client for GitHub's HTTP API v3[6] from an unofficial OpenAPI Specification[7] in JSON. The resilient Java client is generated with the aid of the *RestMule* Metamodel as well as a service policy script that captures restrictions imposed on requests and pagination.

To reproduce the analysis workflow used to assess MDE techologies ([9]), an initial search on GitHub is performed looking for all files that, for a given MDE technology, contain a specific keyword and have a specific file extension that identifies a technology. This query is repeated for each of the 18 technologies in question. Since GitHub imposes a limit of 1000 results regardless of the type of query, and since such queries may return more than this number, the following workaround was used in that study: For each of the files returned, a new search is performed to access the repository of the file, then a new query is used to retrieve all files that contain the keyword and file extension on that repository. This is in hope that more relevant files can be retrieved than in the initial limited search query. From this new set of files, other information is extracted like (1) the number of commits a file has been involved in and (*ii*) the number of authors that have written these commits. With this information it is possible to generate statistics like total number of repositories, total number of files, estimated number of developers (based on commit authors), etc.

This evaluation considers (*i*) asynchronous page traversal, i.e., single multi-paged requests, (*ii*) variable request policies, i.e., for groups of API endpoints, (*iii*) response awaiting when blocked, i.e., consecutive queries that exceed request limits, (*iv*) response caching, i.e., for repeated queries, and (*v*) network failure, i.e., await for reconnection.

Results. For both tools, the experiment ran on a quad core i5-4670k CPU @ 3.40 GHz, with 32GB of RAM and an SSD hard-disk. The JVM was provided with up to 5GB of memory and ran Java 8 on

[4]https://github.com/square/okhttp
[5]https://github.com/kohsuke/github-api
[6]https://developer.github.com/v3/
[7]https://api.apis.guru/v2/specs/github.com/v3/swagger.json

JDK 1.8.0_92. Since running all 18 technologies would take in the order of weeks to execute, we decided to replicate the experiment for a subset of these technologies, more specifically for those used to create graphical model editors – Eugenia, GMF and Sirius.

```
IDataSet<SearchCode> searchCode =
  gitHubAPI.getSearchCode("asc", query, "indexed");
```

Listing 1: Query snippet for RestMule

```
try {
  key = Cache.key(gitHub.queryCode(query));
  if (Cache.contains(key)) { return cache entry; }
  else {
    response = gitHub.queryCode(query);
    Cache.put(key, response);
  }
  for (page : response.pages){
    // request additional pages and consider caching
  }
} catch (NetworkException e1){ handle(e1);
} catch (AuthenticationException e2){ handle(e2);
} catch (ServerException e3){
  handle(e3); // for similar responses
  waitAndRetry(); // for rate limits
}
```

Listing 2: Algorithm for hand-written resilient Java code

The code required to run the experiment for these technologies was in the order of 100 lines of code for both RestMule and GAJ (88 and 139 LOC repsectively), much lower than the orignal code of 550 LOC. As seen in Listings 1 and 2, a query expressed in a single line in RestMule will require a much more verbose algorithm when written in a generic non-resilient manner. It is worth noting that extending the experiment to run on all 18 technologies in both cases would not take more than a couple of extra LOC.

Both tools produced the same results for Eugenia and GMF, whilst GAJ did not terminate for Sirius in our tests[8]). As both tools offer similar capabilities and we obtained the same results after running this case-study, we have gained confidence that the code generated by RestMule (for GitHub) is as good as the hand-written code used by GAJ, and they are both superior to the original experiment's code that was tailored to a specific use-case and much more verbose. Since RestMule is a generic resilient client generation tool designed for any repository with an available OpenAPI Specification, these results are promising.

Threats to Validity. Although the generated client has been built to enable resilience against request restrictions and network failure, a more extensive set of unfavorable scenarios, e.g. contemplating remote API exceptions, has to be considered.

Several remote APIs, such as GitHub, StackExchange, and Bugzilla, have been explored to identify their commonality, yet the results of our evaluation are limited to generating and using the GitHub remote API. Further investigation is required to gain confidence that RestMule can successfully generate clients for other technologies and offer the same resilience as the GitHub client.

4 RELATED WORK

This section presents related work within the MSR research community. Although research using data from collaborative development environments such as (*i*) source control management systems (e.g. GitHub, SourceForge), (*ii*) bug tracking systems (e.g. JIRA, Bugzilla), and (*iii*) archived project communications (e.g. Eclipse Forums, StackOverflow) is relevant to MSR research, we focus on existing work that employs them during the construction of data sets and analysis tools and, in particular, highlight differences to RestMule.

Massive Datasets. Intense data collection tools usually involve mirroring the server contents into massive public datasets. These datasets are built either in a non-evolving fashion, such as the Maven Repository Dataset, or an evolving fashion, such as Stack Exchange's Data Dump and GHTorrent. Non evolving datasets hold information based on a single snapshot of a software repository while evolving datasets usually follow a schedule to make their updated versions publicly available. Consequently, at the time their information is queried, both evolving and non-evolving datasets may be inconsistent when compared to their origin.

GHTorrent and the Maven Repository Dataset use complex infrastructures to retrieve, store and share their data. Their public availability relies on the benevolence of its maintainers [2], which is associated with risk from data stagnation and corruption. For example, both datasets reported data corruption issues that may be a result of data processing tasks, such as cleansing, abstraction, transformation, or a collection thereof. As opposed to dealing with the entire contents of remote software repositories, RestMule processes near real-time data required to satisfy a given user query.

The restriction imposed on the number of requests that can be issued within a particular timespan to a remote API, such as offered by GitHub, has been presented as a major challenge for data collection in the MSR research community and often motivates the use of mirroring datasets. To our knowledge, this restriction has only been addressed by GHTorrent and our framework. GHTorrent manages this restriction by using their collection of user access-tokens[9] and dispatching requests with different user accounts, which are persisted in a shared database, and consequently achieve an increased request rate limit. Currently, RestMule handles request limitations based on a single account. However, our intention is to provide support for collaborative and distributed queries in the future.

Analysis Frameworks. In general, existing frameworks for mining information from software repositories differ based on their orientation, either metric-oriented or workflow-oriented. Metric-oriented frameworks, such as OSSMETER [1] and RepoGrams [13], focus on collecting data for producing metrics, such as software quality, static source code, and changes. Workflow-oriented frameworks like SmartSHARK [14], CODEMINE [3], and BOA [4], aim to provide a shared environment for data analysis purposes. RestMule can act as a complementary component, which can help with the activity of data collection from remote APIs in such systems.

5 CONCLUSIONS AND FUTURE WORK

Massive data collection has proven to be a challenge in the MSR community, although not exclusively. In order to mine data from remote software repositories, applications may perform significant numbers of requests, commonly to public HTTP APIs. Client applications can be blocked by strategies implemented by API providers to protect their servers from saturation. Mining applications tend

[8]The raw data obtained from these experiments can be found at: https://doi.org/10.6084/m9.figshare.5840991

[9]Built with voluntarily offered access-tokens from GitHub user accounts

to re-implement from scratch data collection infrastructures due to the lack of reusable frameworks able to deal with these restrictions.

We have presented RestMule, a framework that comprises a set of reusable facilities that handle request rate quota, network failures, caching and asynchronous paging in a transparent manner; providing a code generator that produces API-specific Java libraries from OpenAPI specifications, that employ those facilities. RestMule allows developers to produce resilient remote-API clients enabling them to focus on their core mining and analysis logic.

Initial evaluation demonstrates that such resilient clients can be used to express queries in a much more concise manner, whilst providing the resilience that would be necessary to execute them, which would otherwise have to be manually implemented. As such, we encourage HTTP service-providers to publish OpenAPI specifications to ease service consumption and enable client code generation.

Future Work. In terms of future work, we plan to develop more sophisticated approaches to handle request rate policies. These approaches include adding support for collaborative clients (multi-user workflows) —in a similar fashion to GHTorrent [5]— and enabling (shared) distributed analysis workflows.

Furthermore we plan to enable the extension of the resilience strategies with user-defined policies.

ACKNOWLEDGMENTS

The work in this paper was supported by the Mexican National Council for Science and Technology (CONACyT) under Grant No.: 602430/440678 and the European Commission via the CROSSMINER Project (732223).

REFERENCES

[1] Bruno Almeida, Sophia Ananiadou, Alessandra Bagnato, Alberto Berreteaga Barbero, Juri Di Rocco, Davide Di Ruscio, Dimitrios S Kolovos, Ioannis Korkontzelos, Scott Hansen, Pedro Maló, et al. 2015. OSSMETER: Automated Measurement and Analysis of Open Source Software.. In *STAF Projects Showcase*. 36–43.

[2] Alberto Bacchelli, Luca Ponzanelli, and Michele Lanza. 2012. Harnessing stack overflow for the ide. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*. IEEE Press, 26–30.

[3] Jacek Czerwonka, Nachiappan Nagappan, Wolfram Schulte, and Brendan Murphy. 2013. Codemine: Building a software development data analytics platform at microsoft. *IEEE software* 30, 4 (2013), 64–71.

[4] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2015. Boa: Ultra-large-scale software repository and source-code mining. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 7.

[5] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: GitHub's data from a firehose. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 12–21.

[6] Huzefa H. Kagdi, Michael L. Collard, and Jonathan I. Maletic. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance* 19, 2 (2007), 77–131. https://doi.org/10.1002/smr.344

[7] Nafiseh Kahani, Mojtaba Bagherzadeh, Juergen Dingel, and James R Cordy. 2016. The problems with Eclipse modeling tools: a topic analysis of Eclipse forums. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 227–237.

[8] Dimitris Kolovos, Louis Rose, Antonio García-Domínguez, and Richard Paige. 2016. The epsilon book. (2016).

[9] Dimitrios S Kolovos, Nicholas Drivalos Matragkas, Ioannis Korkontzelos, Sophia Ananiadou, and Richard F Paige. 2015. Assessing the Use of Eclipse MDE Technologies in Open-Source Software Projects.. In *OSS4MDE@ MoDELS*. 20–29.

[10] openapi:online 2017. StackExchange API version update [Online]. (2017). Available at: https://github.com/APIs-guru/openapi-directory/issues/217 [Accessed: May 18, 2017].

[11] Cesare Pautasso. 2014. RESTful web services: principles, patterns, emerging technologies. In *Web Services Foundations*. Springer, 31–51.

[12] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2013. The maven repository dataset of metrics, changes, and dependencies. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 221–224.

[13] Daniel Rozenberg, Ivan Beschastnikh, Fabian Kosmale, Valerie Poser, Heiko Becker, Marc Palyart, and Gail C Murphy. 2016. Comparing repositories visually with repograms. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 109–120.

[14] Fabian Trautsch, Steffen Herbold, Philip Makedonski, and Jens Grabowski. 2016. Adressing problems with external validity of repository mining studies through a smart data platform. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 97–108.