A Dataset For API Usage

Anand Ashok Sawant
SORCERERS @ SERG
Delft University of Technology
a.a.sawant@student.tudelft.nl

Alberto Bacchelli
SORCERERS @ SERG
Delft University of Technology
a.bacchelli@tudelft.nl

***Abstract*—An Application Programming Interface (API) provides a specific set of functionalities to a developer. The main aim of an API is to encourage the reuse of already existing functionality. There has been some work done into API popularity trends, API evolution and API usage. For all the aforementioned research avenues there has been a need to mine the usage of an API in order to perform any kind of analysis. Each one of the approaches that has been employed in the past involved a certain degree of inaccuracy as there was no type check that takes place. We introduce an approach that takes type information into account while mining API method invocations and annotation usages. This approach accurately makes a connection between a method invocation and the class of the API to which the method belongs to. We try collecting as many usages of an API as possible, this is achieved by targeting projects hosted on GitHub. Additionally, we look at the history of every project to collect the usage of an API from earliest version onwards. By making such a large and rich dataset public, we hope to stimulate some more research in the field of APIs with the aid of accurate API usage samples.**

I. Introduction

An Application Programming Interface (API) is a set of functionalities provided by a third-party component (*e.g.*, library and framework) that is made available to software developers. APIs are extremely popular as they promote reuse of existing software systems [1].

Researchers have investigated APIs under different angles, such as API popularity trends [2], API evolution [3], and API usage [4]. This research has lead to valuable insights on what the research community currently knows about APIs and how practitioners use them. For example, Xie *et al.* have developed a system called MAPO wherein they have attempted to mine API usage for the purpose of providing developers API usage patterns [4]. Based on a developers need MAPO can recommend various code snippets that have been mined from other open source projects. This is one of the first systems wherein API usage recommendation leveraged open source projects to provide code samples.

One of the major drawbacks of the current approaches to investigating APIs is that what it is measured as "usage"—and it used to derive popularity, evolution, and utilization patterns—is the information that can be gathered from file imports (*e.g.*, `import` in Java) and the occurrence of method names in files. This information can be unreliable as there is no type checking to verify that a method invocation truly does belong to the API in question and that imported packages are used. Another important limitation of previous work is that the case studies generally include a small number of examples.

With the current work, we try to overcome these issues: We provide an extensive dataset with detailed and type-checked API method invocation information. Our dataset includes information on 5 APIs and how their public methods are used over the course of their entire lifetime by other 20,263 projects.

To achieve this, we collect data from the open source software (OSS) repositories on GitHub. GitHub in recent years has become the most popular platform for OSS developers, as it offers distributed version control, a pull-based development model, and features similar to social networks [5]. We consider Java projects hosted on GitHub that offer APIs and quantify their popularity among other projects hosted on the same platform. We select 5 representative projects (from now on, we call them only *API*s to avoid confusion with client projects) and analyze their entire history to collect information on their usage. In particular, we get fine-grained information about method calls using a custom type resolution that does not require to compile the projects.

We share our dataset and describe our data collection methods with the hope not only to trigger further research based on finer-grained and vast information, but also make it easier to replicate studies and share analyses.

II. Dataset Construction

The dataset has been built through two steps: (1) We collect data on the usage at project level of APIs across projects within GitHub and we use it to rank APIs according to their popularity; (2) we devise a method to gather fine-grained type-based information on API usages and we collect historical usage data traversing the history of each file of each API client.

A. Coarse-grained API usage: The most popular APIs

GitHub stores more than 10 million repositories [6] written in different languages and using a diverse set of build automation tools and library management systems. To create the current dataset, we focus our effort on one programming language, *i.e.*, Java, and we one specific build automation tool, *i.e.*, Maven.[1] This made the data collection and processing more manageable, yet relevant. Maven employs the use of a Project Object Model (POM) files to describe all the dependencies and targets of a certain project. POM files contain artifact ID and version of each project's dependency, thus allowing us to know exactly which APIs (and version) a project uses. The following is an example of an entry for a POM file:

[1]Maven is one of the most popular Java build tools [7].

```
<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.8.2</version>
</dependency>
```

To determine the popularity of APIs on a coarse-grained level (*i.e.*, project level), we parsed POM files for all GitHub projects using Maven (ca. 42,000). Figure 1 shows a partial view of the results with the 20 most popular APIs in terms of how many GitHub projects depend on them.

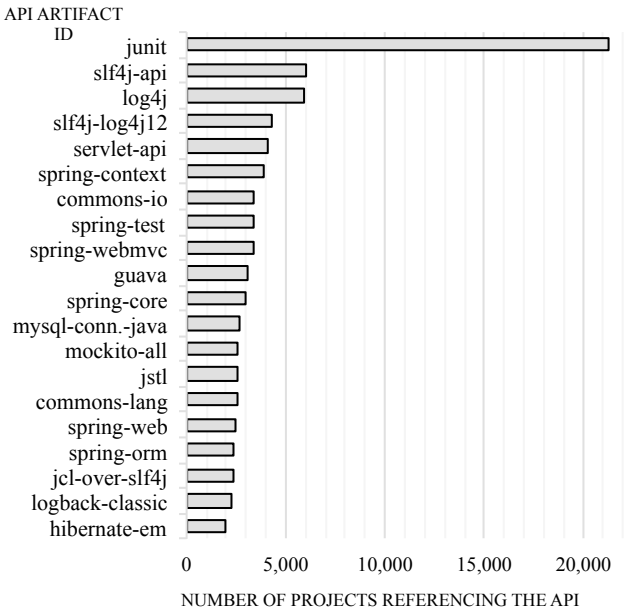

Figure 1. Popularity of APIs referenced on Github

This is in-line with a previous analysis of this type published by Primat as a blog post [8]. Interestingly, we note that our results show that JUnit is by far the most popular and Primat's results report that JUnit is just as popular as SLF4J. This discrepancy can be caused by the differences in sample sizes (he sampled 10,000 projects, while we sampled about 42,000).

B. Fine-grained API usage

We used our coarse-grained analysis of popularity as a first step to select API projects to populate our database. To ensure that the selected API projects offer rich information on API usage and its evolution, rather than just sporadic use by a small number of projects, we consider projects with the following feature: (1) have a broad popularity for their public APIs (*i.e.*, they are in the top 1% of projects by the number of client projects), (2) have an established and reasonably large code base (*i.e.*, they have at least 150 classes in their history), (3) and are evolved and maintained (*i.e.*, they have at least 10 commits per week in their lifetime). Based on these characteristics, we eventually select the five APIs summarized in Table I, namely Spring, Hibernate, Guava, and Guice and Easymock. We decide to remove, for example, JUnit as it is an outlier in popularity and its code base does not respect our requirements. We decided to keep Easymock, despite its small number of classes and relatively low amount of activity in it's repository (ca. 4 commits per week), to add variety to our sample. The chosen APIs are to be used by clients in different ways: *e.g.*, Guice clients use it through annotations, while Guava clients instantiate an instance of a Guava class and then interact with it through method invocations.

Table I
SUBJECT APIS

API & GitHub repo	Inception	Releases	Unique Entities	
			Classes	Methods
Guava google/guava	Apr 2010	18	379	2,010
Guice google/guice	Jun 2007	8	192	463
Spring spring-framework	Feb 2007	40	431	1,161
Hibernate hibernate/hibernate-orm	Nov 2008	77	585	1,963
EasyMock easymock/easymock	Feb 2006	14	10	86

The most common approaches to automatically mine heterogeneous datasets for API usage are *text-matching-based*, *build-based* and *partial-programming-analysis-based*. The first, used for example by Mileva *et al.* [2], extracts API usage by matching explicit imports and corresponding method invocations directly in the text of source code files; the second, used for example by Lämmel *et al.* [9], extracts API usage by first compiling the code and then parsing generated files for precise type-correct method usage information; the third, uses the partial programming analysis (PPA) tool developed by Dagenais *et al.* [10], which can parse incomplete code snippets and give accurate type information on the code snippet. At first we tried all three approaches to generate our usage dataset, but results were not satisfactory to our aim. The text-matching-based approach proved problematic, for example, in the case of imported API classes that share method names, because we could not disambiguate the method invocations without type-information. Although some analysis tools used in dynamic languages [11] handle these cases through the notion of candidate classes, we considered this approach suboptimal for typed languages. The build-based approach was also problematic. Although it provided precise information, to use it we had to discard ca. 3,000 projects (only considering Guava clients) and many more revisions since they could not be compiled. The PPA approach was flawed as the tool that would perform the analysis works only in the eclipse plugin environment. This would require all projects that are to be analyzed to be imported into eclipse. Such a task proved to be infeasible due to the number of projects that were to be analyzed.

Our final solution was to build a tool based on the JDT Java AST Parser [12], *i.e.*, the parser used in the Eclipse IDE for continuous compilation in background. This parser handles partial compilation: When it receives in input a source code file and a jar file with possibly imported libraries, it is capable of resolving the type of methods invocation and annotations

of everything defined in the code file or in the provided jar.

With our tool based on JDT, we gather the entire history of usage of API artifacts over different versions. In practice, we manually downloaded all the jar files corresponding to all the releases of the five considered API projects, then we used Git to obtain the history of each client project and we run the JDT parser on each source code file providing the jar with the version of the API that the client project declares in Maven. The result are accurate type-resolved method invocation references for the considered projects through their whole history.

C. Results

Table II shows an introductory view on the information about the collected usage data. We see for example that in the case of Guava, even though version 18 is the latest (see Table I), version 14.0.1 is the most popular amongst clients. APIs such as Spring, Hibernate and Guice predominantly expose their APIs as annotations, however we see also a large use of methods from their APIs. The earliest usages of Easymock and Guice are outliers as GitHub as a platform was launched in 2008, thus the repositories that refer to these APIs were moved to GitHub as existing projects.

Table II
INTRODUCTORY USAGE STATISTICS

API	Most popular release	Usage across history	
		Invocations	Annotations
Guava	14.0.1	1,148,412	—
Guice	3.0	59,097	48,945
Spring	3.1.1	19,894	40,525
Hibernate	3.6	196,169	16,259
EasyMock	3.0	38,523	—

III. DATA ORGANIZATION

We stored all the data that was collected from all the client GitHub projects and API projects in a relational database, precisely PostresSQL. We have chosen this type of database because the usage information that we collect can be naturally expressed in forms of relations among the entities. Also we can immediately leverage SQL functionalities to perform some initial analysis and data pruning.

Figure 2 shows the database schema for our dataset. On the one hand we have information for each client project: The PROJECTS table is the starting point and stores a project's name and its unique ID. Connected to this we have PROJECTDEPENDENCY table, which stores information collected from the Maven POM files about the project's dependencies. We use a DATE_COMMIT attribute to trace when a project starts including a certain dependency in its history. The CLASSES table contains one row per each uniquely named class in the project; in the table CLASS_HISTORY we store the different versions of a class (including its textual content, ACTUAL_FILE) and connect it to the tables METHOD_INVOCATION and ANNOTATION where information about API usages are stored. On the other hand, the database stores information about API projects, in the tables prefixed with API. The starting point is the table API that stores the project name and it is connected to all its versions (table API_VERSION, which also stores the date of creation), which are in turn connected classes (API_CLASS) and their methods (API_METHOD) that also store information about deprecation. Note that in the case of annotations we do not really collect them in a separate table as annotations are defined as classes in Java.

A coarse-grained connection between a client and an API is done with a SQL query on the tables PROJECTDEPENDENCY, API and API_VERSION. The finer-grained connection is obtained by also joining METHOD_INVOCATION/ANNOTATION and API_CLASS & API_METHOD.

The full dataset is available as a PostgreSQL data dump on FigShare [13], under the CC-BY license. For platform limitations on the file size the dump has been split in various tar.gz compressed files, for a total download size of 51.7 GB. The dataset uncompressed requires 62.3 GB of disk space.

IV. LIMITATIONS

Mining API usages on such a large scale and to this degree of accuracy is not a trivial task. We report consequent limitations to our dataset. First, to analyze as many projects as possible on GitHub, we needed to checkout the correct/latest version of the project on GitHub. GitHub uses git as a versioning system which employs branches, thus makes the task of automatically checking out the right version of the client challenging. We consider that the latest version of a given project would be labeled as the 'master' branch. Although this is a common convention, by restricting ourself to only the master branch there is a non-negligible chance that some projects are dropped. Second, we target only projects based on a specific build automation tool on GitHub. This results in data from just a subset of Java projects on GitHub and not all the projects. This may in particular affect the representativeness of the sample of projects. We try to mitigate this effect by considering one of the most popular building tools in Java: Maven. Third, even though GitHub is a very popular repository for open source software projects, this sole focus on GitHub leads to the oversight of projects that are on other open source platforms such as Sourceforge and Bitbucket. Moreover, no studies have yet compared the representativeness of GitHub projects with respect to industrial ones.

V. RESEARCH OPPORTUNITIES

The data we collected can be used for a number of applications. We outline some in the following.

Knowing which parts of an API are popular (*i.e.*, used by most projects) can give an indication as to what kind of features of an API are most in demand and give API developers an indication about features of their API that should be changed with more care to limit the introduction of breaking changes. In addition, API methods used by more clients could be considered as "better tested" than methods sporadically used, this can give a quality indication to someone deciding which library to adopt or whether to update to a new version. Moreover, the popularity of APIs can be used to support program comprehension of the libraries themselves:

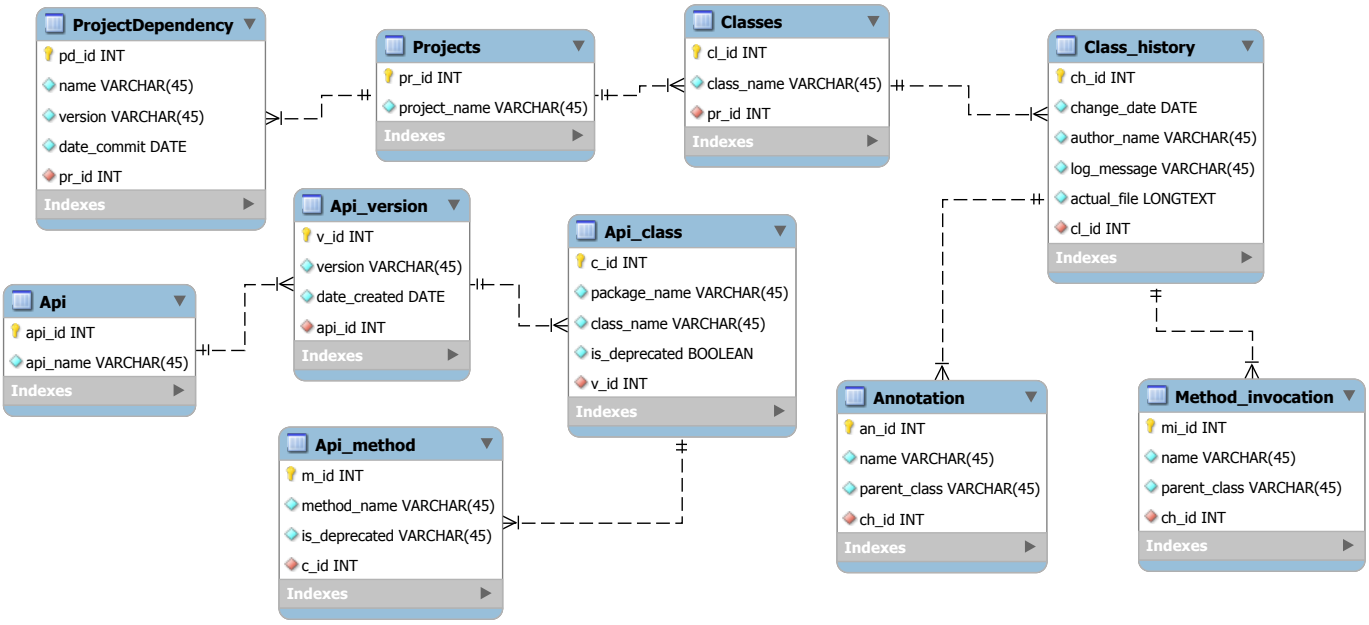

Figure 2. Database Schema For The Fine-grained API Usage Dataset

For example, by considering popular API one could find entry points in a system, as it has been done with emails [14].

The evolution of the features of the API can be mapped to give an indication as to what has made the API popular. This can be used to design and carry out studies on understanding what precisely makes a certain API more popular than other APIs that offer a similar service. Moreover API evolution information gives an indication as to exactly at what point of time the API became popular, thus it can be studied in coordination with other events occurring to the project.

A large set of API usage examples is a solid base for recommendation systems: One of the most effective ways to learn about an API is by seeing samples [15] of the code in actual use. By having a set of accurate API usages at ones' disposal, this task can be simplified and useful recommendations can be made to the developer; similarly to what has been done, for example, with Stack Overflow posts [16].

VI. Conclusion

We have presented a rich and detailed dataset to allow researchers and developers alike get insights into trends related to APIs. A conscious attempt has been made to harvest all the API usage accurately. A total of 20,263 projects and accumulated a grand total of 1,482,726 method invocations and 85,098 annotation usages related to 5 APIs have been mined. It is our hope that our large database of API method invocations and annotation usages will trigger even more precise and reproducible work in relation to software APIs.

References

[1] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.

[2] Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining API popularity," in *Testing–Practice and Research Techniques*. Springer, 2010, pp. 173–180.

[3] D. Dig and R. Johnson, "How do APIs evolve? a story of refactoring," *Journal of software maintenance and evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

[4] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in *Proceedings of MSR 2006*, pp. 54–57.

[5] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu, "Cohesive and isolated development with branches," in *Fundamental Approaches to Software Engineering*. Springer, 2012, pp. 316–331.

[6] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean GHTorrent: Github data on demand," in *Proceedings of MSR 2014*, pp. 384–387.

[7] V. Massol and T. M. O'Brien, *Maven: A Developer's Notebook: A Developer's Notebook*. O'Reilly, 2005.

[8] O. Primat, "Github's 10,000 most popular java projects – here are the top libraries they use," http://blog.takipi.com/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/, nov 2013.

[9] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source java projects," in *Proceedings of ACM SAC 2011*, pp. 1317–1324.

[10] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *ACM Sigplan Notices*, vol. 43, no. 10. ACM, 2008, pp. 313–328.

[11] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an extensible language-independent environment for reengineering object-oriented systems," in *Proceedings of CoSET 2000*.

[12] L. Vogel, "Eclipse JDT-Abstract Syntax Tree (AST) and the java model-tutorial," http://www.vogella.com/tutorials/EclipseJDT/article.html.

[13] A. Sawant and A. Bacchelli, "API Usage Databases," 03 2015. [Online]. Available: http://dx.doi.org/10.6084/m9.figshare.1320591

[14] A. Bacchelli, M. Lanza, and V. Humpa, "RTFM (Read The Factual Mails) –augmenting program comprehension with remail," in *Proceedings of CSMR 2011 (15th IEEE European Conference on Software Maintenance and Reengineering)*, 2011, pp. 15–24.

[15] M. P. Robillard and R. DeLine, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[16] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack overflow in the ide," in *Proceedings of ICSE 2013 (35th International Conference on Software Engineering, Tool Demo Track)*. IEEE CS Press, 2013, pp. 1295–1298.