apiwave: Keeping Track of API Popularity and Migration

Conference Paper · September 2015

DOI: 10.1109/ICSM.2015.7332478

CITATIONS
12

READS
48

2 authors:

Andre Hora
Universidade Federal de Mato Grosso do Sul
47 PUBLICATIONS **272** CITATIONS

SEE PROFILE

Marco Tulio Valente
Federal University of Minas Gerais
165 PUBLICATIONS **1,238** CITATIONS

SEE PROFILE

apiwave: Keeping Track of API Popularity and Migration

André Hora, Marco Tulio Valente
ASERG Group, Department of Computer Science, Federal University of Minas Gerais, Brazil
{hora, mtov}@dcc.ufmg.br

***Abstract*—Every day new frameworks and libraries are created and existing ones evolve. To benefit from such newer or improved APIs, client developers should update their applications. In practice, this process presents some challenges: APIs are commonly backward-incompatible (causing client applications to fail when updating) and multiple APIs are available (making it difficult to decide which one to use). To address these challenges, we propose apiwave, a tool that keeps track of API popularity and migration of major frameworks/libraries. The current version includes data about the evolution of top 650 GitHub Java projects, from which 320K APIs were extracted. We also report an experience using apiwave on real-world scenarios.**

I. Introduction

Nowadays, software repositories make it simpler to store source code and collaborate when creating small or large systems. GitHub, the most popular software repository, has more than 24 million projects and 10 million developers. Such projects continue to evolve daily with the addition of new features, bug-fixes and code refactoring. This generates a large amount of information that is hard to keep track.

Many of these projects are frameworks and libraries, which are commonly used worldwide in software development to increase productivity. As these projects evolve, their Application Programming Interfaces (APIs) are likely to change. As a side effect, client developers should often update their applications to benefit from newer or improved APIs. In practice, these activities involve some challenge due to:

- **API Popularity:** APIs are evolving at fast pace and multiple libraries that provide similar services are available [1]. This makes it difficult for client developers to decide which API to use. For example, developers should decide whether they should use a bleeding edge library or an old-stable one.
- **API Migration:** APIs are commonly backward-incompatible, causing client applications to fail [2], [3]. Thus, when updating their code, client developers may need to find replacements for removed/renamed/updated APIs to keep their applications working correctly.

To address these challenges, we propose the tool apiwave. Its goal is to keep track of API popularity and migration of major open-source frameworks/libraries, aiming to help both API client and developers. The current apiwave version includes data about top 650 GitHub Java projects and is available at **http://apiwave.com**.

II. apiwave in a Nutshell

apiwave is proposed to fill a gap existing between the literature and real-world tools in the context of API evolution. By mining hundreds of frameworks and libraries, we aim to better understand how APIs are evolving over time and make such large-scale data easily accessible. apiwave is implemented as a web application and it is responsive for multiple devices as shown in Figure 1.

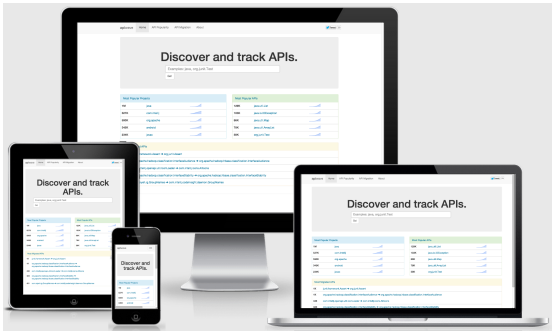

Fig. 1. apiwave home page.

We focus on two aspects: API popularity and migration. While it is easy to know how "popular" is a project in number of contributors or fans, it is quite hard to know how many clients are actually using it. Thus, apiwave covers such aspect and characterizes how popular is a framework/library, package or interface in number of clients. For example, one can detect what are the most used interfaces in Android or whether Facebook packages are decreasing/increasing usage. apiwave also supports the replacement of interfaces in order to help framework/library migration. Moreover, APIs are always presented in the light of source code examples.

A. Architecture

Figure 2 presents an overview of apiwave architecture, which includes two modules: Preprocessing and Frontend.

Preprocessing Module. This module receives as input the history of a project stored in a source code repository (*e.g.,* a GitHub project). It provides as output a database of APIs including information about API popularity and migration as well as a database of source code examples about such APIs.

In a first step, in *Diff Processor*, the project history is downloaded from the repository and organized in way that eases the comparison between two versions of a source code file. This structured data is then used in the second step to facilitate

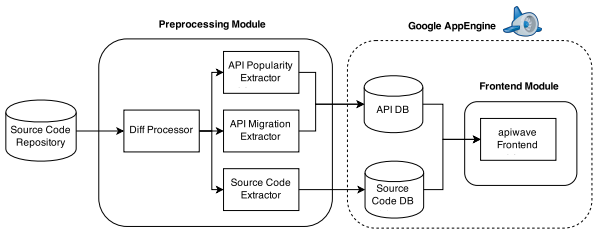

Fig. 2. apiwave architecture.

the extraction of data on API popularity, API migration and source code examples. *API Popularity Extractor* mines from the diff the notion of popularity while *API Migration Extractor* mines the notion of migration; the results of such mining is stored in a database. For example, consider the diff shown in Figure 3 in which a refactoring was done between two versions of a Java class. *API Popularity Extractor* will detect that a dependency to java.util.Vector has been removed and that a dependency to java.util.List has been added, *i.e.,* java.util.List has gained one user while java.util.Vector has lost one user. This is done by mining import statements. In addition, *API Migration Extractor* will infer that the class java.util.Vector has been replaced by java.util.List. This is done by ensuring that only one dependency is removed and only one is added. Finally, *Source Code Extractor* will extract the whole diff as an example to document the refactoring.

```
 import android.widget.TextView;

-import java.util.Vector;
+import java.util.List;

 public final class BenchmarkActivity extends Activity {

@@ -76,7 +76,7 @@ public void handleMessage(Message message) {
   };

   private void handleBenchmarkDone(Message message) {
-    Vector<BenchmarkItem> items = (Vector<BenchmarkItem>) message.obj;
+    List<BenchmarkItem> items = (List<BenchmarkItem>) message.obj;
     int count = 0;
```

Fig. 3. Diff between two versions of a source code file.

Frontend Module. This module represents the web application interface of apiwave. It receives as input two databases: one including information about API popularity and API migration, and one with source code examples. It provides as output the view for the end user, which is implemented in Python. This module and the two databases are stored on the cloud through Google AppEngine platform. This allows us to focus more on code development than on database administration, and to scale using Google infrastructure.

B. Main Functionalities

In apiwave it is possible to navigate in three levels of granularity: (1) frameworks/libraries, (2) packages or (3) interfaces. Each framework/library is composed of a set of packages, and each package is composed of a set of interfaces.

Figure 4 presents a typical page for framework/library (or package) granularity with respective functionalities. First, the

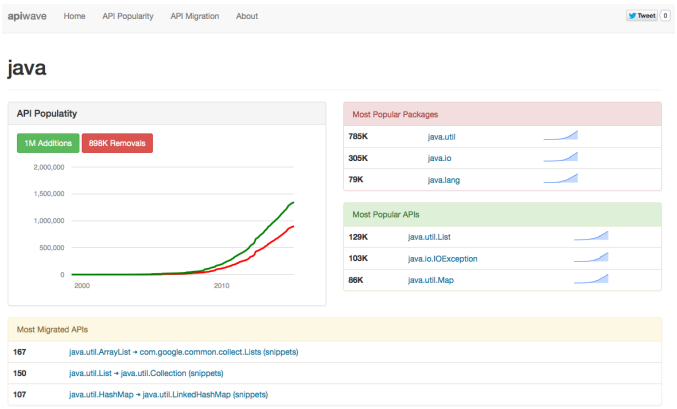

Fig. 4. Framework/library example page.

API Popularity Graph (top-left) presents the evolution of the number of users for a given framework/library. The green line represents the gain of dependencies while the red line represents the loss of dependencies over time. Second, the *API Popularity Ranks* (top-right) show the most popular packages and interfaces of the analyzed framework/library. Finally, the *API Migration Rank* (bottom) displays the most migrated interfaces. In the case an interface is selected, the page is contextually updated including source code examples.

The current version of apiwave includes data about the evolution of top 650 GitHub Java projects. From such data, 320K frameworks/libraries, packages, and interfaces were extracted. To summarize such information, we propose two rankings: Most Popular APIs[1] and Most Migrated APIs[2].

C. Potential Users

Framework/library developers can use apiwave to verify how their APIs are being used by clients. Client developers can discover, compare and migrate APIs. In addition, we present a complementary view of our users obtained from our Google Analytics data. We present the data collected from such service covering over two months, from April 5th, 2015 to June 16th, 2015. During this time frame, we had 32,063 unique users and 50,080 pageviews. Such users came from different countries and were classified by this service as related to distinct interests, as shown in Table I.

TABLE I
TOP-5 USERS BY COUNTRY AND INTEREST.

Country	Unique Users	Interest	% of Users
United States	7,039 (22%)	Java	9.05%
India	3,196 (10%)	Mobile Phones	3.34%
Germany	1,817 (5.7%)	Web Development	3.06%
South Korea	1,642 (5,1%)	Data Management	2.85%
France	1,255 (3.9%)	Linux and Unix	2.85%

[1]http://apiwave.com/most_popular_projects
[2]http://apiwave.com/most_migrated_apis

III. Case Studies

A. Following API Popularity

In this case study, we focus on exploring four distinct API popularity trends: (1) fast growth, (2) constant growth, (3) peak growth, and (4) dead growth. To detect these trends, we analyze the interface popularity of four widely adopted Java libraries: Android, JSON, Apache Hadoop, and Minecraft.

Figure 5 shows trends detected in interfaces of these libraries. Fast growth is seen in android.content.Context, confirming the overall popularity of Android nowadays. Constant growth is presented in org.json.JSONArray; this alerts library developers that even though they are gaining clients over time, it occurs at a constant pace, so there is space for improvements. Peak growth is shown in org.apache.hadoop.fs.Path; such peaks may confirm (or not) whether an improvement or a promotion resulted in more clients. Finally, dead growth is seen in net.minecraft.src.ItemStack; this states that clients are not using the interface anymore either because it is buggy or because it is replaced by another one.

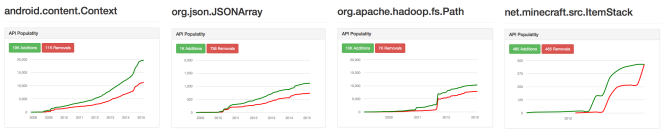

Fig. 5. Exploring API popularity trends in widely adopted Java libraries.

B. Supporting API Migration

In this preliminary case study, we analyze real-world needs in the context of API migration and assess whether apiwave is able to handle such needs. To cover that, we present in the following lines four real-world questions about interface migration extracted from StackOverflow:[3]

1) **Replacement for obsolete Hashtable class in Java** (12.3K views): "*When I tried to use the Hashtable class, Netbeans gave me an error. Any replacement suggestions?*"
2) **Assert in junit.framework has been deprecated - what use next?** (8.6K views): "*I bump version of JUnit to 4.11 and get a warning. How and to what migrate?*
3) **Replacing com.google.inject with javax.inject** (5K views): "*Is it true that javax.inject annotations can function as direct replacements for com.google.inject?*
4) **Replacement for java.net.URL** (200 views): "*I am looking for replacement for java.net.URL. The problem with current java.net.URL is that it can't model protocols other than [http, https, ftp, file, and jar]*"

These questions highlight that there exist a demand for API migration. By using apiwave we are able to automatically answer all the aforementioned questions. For the first question, apiwave proposes the replacement of java.util.Hashtable by java.util.HashMap.[4] For the second, the tool proposes the replacement of junit.framework.Assert by org.junit.Assert.[5] For the third question, it confirms that com.google.inject.Inject can be replaced by javax.inject.Inject.[6] For the last one, apiwave proposes the replacement of java.net.URL by java.net.URI.[7] Moreover, apiwave provides more than one suggestion for some interfaces. The replacement of junit.framework.Assert, for example, is complemented with two other suggestions: org.testng.Assert and org.junit.Assert.assertEquals, which are alternative solutions. All the migrations suggested by apiwave matched the best human answer to these questions in StackOverflow, confirming the correctness of our answers.

[3]StackOverflow question pages: 1) http://goo.gl/9axMJl, 2) http://goo.gl/VJFI65, 3) http://goo.gl/3Uaz5S, 4) http://goo.gl/5OFJuP

[4]http://apiwave.com/java/api/java.util.Hashtable

[5]http://apiwave.com/java/api/junit.framework.Assert

[6]http://apiwave.com/java/api/com.google.inject.Inject

[7]http://apiwave.com/java/api/java.net.URL

IV. Related Work

The literature proposes several approaches to handle API migration. For example, this can be done with the support of modified IDEs [4] or by mining source code [2], [5], [6], [7], [8], [9] In the context of API popularity, one study was an inspiration for us, but it is restricted to medium-scale Apache-only analysis with no tool support [10]. Unfortunately, these approaches neither provide a tool to be used by developers nor analyze large-scale data to satisfy real-world needs. Moreover, none of them handle both API popularity and migration.

V. Conclusion

To the best of our knowledge, apiwave is the first large-scale tool that keeps track of API popularity and migration. Currently, it contains data from 650 GitHub Java projects, from which 320K APIs were extracted. We plan to extend apiwave with more projects and to cover other programming languages.

Acknowledgment

This research was supported by CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil.

References

[1] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *International Conference on Software Maintenance*, 2013.
[2] W. Wu, Y.-G. Gueheneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *International Conference on Software Engineering*, 2010.
[3] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? The case of a smalltalk ecosystem," in *International Symposium on the Foundations of Software Engineering*, 2012.
[4] J. Henkel and A. Diwan, "Catchup!: Capturing and replaying refactorings to support API evolution," in *International Conference on Software Engineering*, 2005.
[5] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.
[6] A. Hora, N. Anquetil, S. Ducasse, and S. Allier, "Domain Specific Warnings: Are They Any Better?" in *International Conference on Software Maintenance*, 2012.
[7] A. Hora, N. Anquetil, S. Ducasse, and M. T. Valente, "Mining System Specific Rules from Change Patterns," in *Working Conference on Reverse Engineering*, 2013.
[8] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "APIEvolutionMiner: Keeping API Evolution under Control," in *Software Evolution Week (European Conference on Software Maintenance and Working Conference on Reverse Engineering)*, 2014.
[9] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How Do Developers React to API Evolution? The Pharo Ecosystem Case," in *International Conference on Software Maintenance and Evolution*, 2015.
[10] Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining API Popularity," in *International Academic and Industrial Conference on Testing - Practice and Research Techniques*, 2010.