Exploiting Dependency Management Systems to Learn about API Usage: A Lucene Case Study

Manuel Leuenberger
leuenberger@inf.unibe.ch

Software Composition Group
University of Bern
Switzerland
http://scg.unibe.ch

Abstract
Understanding API usage is important for API developers to manage the evolution of their projects. They need to know who their clients are and which API versions and features they use. Many API usage studies analyze either a few hand-selected projects or whole package repositories. Those approaches do not suit the needs of an API developer, who is only interested in the ecosystem formed around his API. We present an approach to find clients of a specific API by exploiting dependency management systems. As a validation we perform a case study on the ecosystem around Apache Lucene, an open-source full-text search engine. We inspect who uses Lucene and how it is used. We implement our approach in Kowalski for Maven, but the same approach could also be applied on other systems, for example NPM.

1 Introduction
API developers need to understand how their API is used in order to manage its evolution. To estimate the impact of changes, developers need to identify popular, missing, or unused features [HLSN14]. The answers to these questions are buried in the clients of an API. While it may be easy to collect the relevant clients for a company-internal API, the problem is harder in the open-source environment as API developers do not necessarily know their client projects.

In this paper we present an approach to automatically collect clients of an open-source API by exploiting dependency management systems. Given the name of the API, our implementation, called Kowalski, relies on Maven services to find and download client binaries.

We evaluate the performance of our approach with a case study of the usage of Apache Lucene. We show that the analysis of the clients collected by Kowalski can give useful insights to API developers about the usage of the API. With Kowalski we find 2'548 versions of 148 clients that are calling Lucene methods. Kowalski can only find clients that are published in the public Maven Central repository, which are mostly libraries, not standalone applications. Among these clients we find search platforms, databases, ORMs, RDF frameworks, wikis, and a vulnerability checker. Tracking the major releases of Lucene, we find that only a few clients adopt a new release rapidly. It takes about two years for a release to reach its peak usage. Some clients stick to a Lucene version for over five years. These findings are in concordance with earlier studies on library adoption [KGI+17]. The most popular methods of Lucene are used to access an index and search within it. A breaking change in those methods has a big impact on Lucene clients. Statistics about the index are rarely used. Changing related statistics methods has therefore a limited impact. We identify the conversion from models specific to the domain of the client to Lucene queries to be a non-trivial common use case. This is a potential for new Lucene functionality that supports this use case.

In the remainder of the paper we explain how Kowalski finds clients of an API. Then we present more detailed insights of our Lucene case study.
2 Collecting API Clients

Before we can analyze Lucene clients to see how they use Lucene, we need to find them. Collecting clients of a specific API means that we need to extract a sub-graph of the dependency graph spanned by all projects, as shown in Figure 1. First, we need to find the APIs for which we want to collect clients (a). Those APIs are different versions of Lucene. Second, we need to find clients of the matched APIs (b). Third, we need to download the API client artifacts we want to analyze (c). These artifacts can be sources, binaries, or documentation.

Many Java projects use Maven as a dependency management system, which we can exploit for our purpose. Maven projects declare their dependencies in a meta-data file. For example, one version of Neo4j declares the artifact descriptor `org.neo4j:neo4j-lucene-index`, version 3.1.1, and a dependency to Lucene version 5.5.0. Neo4j developers use Maven to automatically collect the required Lucene dependency from a package repository. Just as Lucene, Neo4j itself is published to this package repository. Therefore, both the API and its client are stored in the same repository and linked through the declared dependency.

In Figure 2 we show how the dependency sub-graph extraction is implemented in KOWALSKI. The query to search for Lucene is passed to Maven Central Search (a) to receive a list of all versions of Lucene. Clients of an artifact are found by scraping the mvnrepository website (b). In the last step (c) we download the binaries of a client including its dependencies using Maven itself. The intermediate results of this process are stored in three pools. The descriptors of the Lucene versions identified (a) end up in the first pool. Multiple workers consume the Lucene descriptors from the first pool, identify the descriptors of client projects (b), and store them in the second pool. Another group of workers then downloads the client binaries (c) and stores the classpaths in the third pool. The clients in the third pool are then processed by our static analysis worker to extract their call graphs. As all workers are independent of each other, they are executed in parallel. Within two hours KOWALSKI collects 7'123 versions of 294 artifacts belonging to 174 groups as potential Lucene clients.

3 Analyzing Lucene Usage

From the collected potential Lucene clients we extract a call graph and filter the dataset. We find 2'548 versions of 148 artifacts belonging to 125 groups as independent clients that are calling Lucene methods. In 2'948 versions we cannot find a call to Lucene, which means that Lucene is either a transitive or unnecessary dependency. The remaining artifacts are parts of Lucene itself and not independent clients.

3.1 Who is using Lucene?

We lookup the homepages of the clients with recent and multiple releases. The constraint filters inactive or dead projects. We find the two rival search platforms using Lucene, Apache Solr and Elasticsearch. Both expose full-text search as a web service. The graph database Neo4j and the multi-model database OrientDB use Lucene as a full-text index, instead of rolling their own solutions. The ORM frameworks Hibernate and ActiveObjects support full-text search through Lucene. We also find RDF frameworks (RDFAj, Apache Jena) and Wikis (WikiMedia, Wiki-Brain). With Uberfire there is a framework to build workbench applications and OWASP checks project dependencies for known vulnerabilities.

Most detected clients are related to databases or services. Those are not standalone applications, but rather application components. We argue that standalone applications are rarely distributed through Maven as their primary intention is not reuse in other projects. Our results reflect the usage of Lucene in other libraries and frameworks. The usage in standalone applications might lead to different conclusions.

3.2 How are new Lucene releases adopted?

To measure the adaptation of different Lucene versions we count the number of client releases over time. We track the client’s date of publication and the used Lucene version, as shown in Figure 3. The used versions of Lucene are color-coded and the first release of the major Lucene versions is indicated.

For Lucene 1 we do not find any clients. The only version 1 we find in Maven Central was released at the same time as the first version 2. We see that the release cycle of major Lucene releases is speeding up.
Figure 2: Dataflow in Kowalski from the initial query to the classpath of downloaded binaries to analyze.

While the jump from version 2 to 3 takes more than three years, each following jump takes a few months less.

For version 2 we notice a long lifetime. Many clients released in 2011 and 2014 depend on it, although the last maintenance release was at the end of 2010. We find that the single project ActiveObjects accounts for 81% of these releases and is causing the irregular spikes. ActiveObjects sticks to a nearly five year old Lucene version instead of updating, although already two successors have been released. The documentation mentions limited support for Lucene.\(^2\) The low priority for supporting full-text search in ActiveObjects could explain the reluctance to upgrade to a newer version.

For Lucene 4 there are some clients published before the official Lucene release. Those clients depend on alpha- and beta-versions of the major release. We can see that as soon as a new version is released, its usage increases until a new version is released, then decreases slowly, probably as it is replaced by its successor. It takes nearly two years to reach that peak usage for versions 3 and 4. We interpret this as hesitation of developers to migrate. Be it due to the complexity of the migration, the hesitation to change a working system without necessity, or the lack of trust in the robustness of the newer version.

Version 5 usage seems to be decreasing rapidly after reaching its peak usage. We find only one project that skips version 5, while there are still many clients using Lucene 4 in their latest version. We notice that our analysis does not terminate for 14 newer Solr and 11 Elasticsearch versions, while it succeeds on older versions. The analysis fails to terminate for 161 out of the 7'123 artifacts, therefore this should not affect the general results.

\(^2\)bitbucket.org/activeobjects/ao/src/...
As we notice the increasing release rate for versions 5 and 6 we search for possible explanations in the nature of the releases. While browsing the documentation of the various releases we detect some @Deprecated tags, indicating the imminent end of life of some methods. A look at the deprecation timeline in Figure 4 reveals: Version 2 includes no deprecation tags, but 3 introduces many of them (320), while 4 removes half of them again. We interpret this as a mean to allow smoother migration to newer versions. While the fate of the project might not have been clear for version 2, it became clearer in version 3 and many methods were obsolete and marked as such to prepare for their removal. Many of them were removed in version 4. In 5 we notice no change in deprecation, but for the first time the size of the project decreases, while it was rapidly growing before. We suppose version 5 was mainly a refactoring to prepare for new features and another round of deprecation removal in version 6.

3.3 How are the clients using Lucene?

From the names of the client packages, classes and methods we extract a wordcloud as a high-level view of the client model. In the package names in Figure 5 (a) we find the most occurrences for search, index and.lucene, indicating the purpose of the module: Searching, indexing documents or wrapping the Lucene API for the use in the domain of the client. A different picture emerges from the wordcloud of the class names in Figure 5 (b). The word search is no longer prominent, but query, field, and filter take its place. For method names in Figure 5 (c) it is now query and index in pole position. As many methods are dedicated to queries, we inspect these methods and find that most of them are building Lucene queries based on the clients’ domain specific models: either by parsing some DSL or by mapping model states to query clauses. We also find that these query-generating methods are the most frequent factory methods for Lucene objects in clients. Lucene provides builders for some types of queries, but many clients do not use them. Here we see potential for Lucene to better support this use case with new features.

We count the number of invocations per method to identify popular and unpopular methods. The five most used methods are shown in Table 1, the least used ones in Table 2. We see thousands of calls related to the IndexReader, which is used to search in an index. Amongst the least used methods we find methods related to statistics. But results are less indicative here, as many of the rarely used methods are short-lived and either deprecated or replaced by methods with a similar signature.

4 Related Work

Many API usage studies focus on the needs of downstream developers. For example they find usage examples of an API by analyzing a few hand-selected projects [GRPK17, BW12]. Others collect usage patterns by analyzing a large corpus of projects and select those patterns with the highest support [RLP13, LPS11]. Raemaekers et al. extract metrics and call graphs from seven years of release history in Maven Central [RvDV14]. We focus on the needs of upstream developers in this study. Our data collection approach is targeted, automated, and requires only the name of the API of interest as an input. As an output we get relevant API client projects we can pipe into the analysis.

Kula et al. visualize dependency relations extracted from Maven repository meta-data [KRG+14]. In a later study they analyze library aging by parsing project meta-data in version control systems [KGF+17]. They track library usage as “a population count of client systems that use a certain library at a specific point in time”, which differs from our measure of release counts. However, they note similar adoption rates for popular APIs. With KOWALSKI we not only process project meta-data, but we collect the
associated binaries with their dependencies. This enables us to analyze how an API is used, not just if it is used.

API usage can also be tracked by analyzing source code in version control systems. For example Google BigQuery\(^3\) or Boa [DNRN13] index the history and source code of GitHub projects. Package repositories, the origin of our dataset, track the evolution in a coarser grained manner, as only the meta-data of published versions is indexed. Additionally, not all open-source Java projects on GitHub are published on Maven and therefore are undiscovered by our approach. On the other hand, the analysis of binaries as in our case is type-precise. This level of precision is hard to achieve when analyzing raw source code.

5 Conclusions and Future Work

In this paper we show how to collect clients of an API by exploiting dependency management systems. The case study on Apache Lucene shows that our approach provides an extensive source to answer questions of API developers related to API usage.

We show that an API usage dataset must not necessarily be composed of some hand-picked, the most popular, or all available projects. We can exploit dependency relations between projects to cut an ecosystem around a specific API out of a huge dataset.

For many languages there exist commonly used dependency management systems with underlying central package repositories. For example NPM for JavaScript, PyPI for Python, and CRAN for R. We intend to implement our API client collection process for NPM in order to enable API client collection for JavaScript APIs.

Acknowledgments

The authors gratefully acknowledge the funding of the Swiss National Science Foundations for the project “Agile Software Analysis” (SNF project No. 200020_162352, Jan 1, 2016 - Dec. 30, 2018).\(^4\)

References


\(^3\)https://cloud.google.com/bigquery/public-data/github

\(^4\)http://p3.snf.ch/Project-162352