A Case Study of API Management Using Aspects in a Brazilian Organization

Completed Research

Samuel L. Santana
Universidade Federal do Pará
ifpa.sntana@gmail.com

Rodrigo Q. Reis
Universidade Federal do Pará
quites@quites.net.br

Cleidson R. B. de Souza
Universidade Federal do Pará
cleidson.desouza@acm.org

Introduction

A new business context emerged as a result of the connection of internal parts of an organization with the external public (Chesbrough 2003). This context allows the creation of a collaborative environment between organizations and their external public. The collaboration can occur in a variety of ways, but it often occurs through a set of software solutions that provide support and automation of interoperable activities and transactions, both by the organizations themselves and the external members. This set of solutions is translated into a set of Application Programming Interfaces (APIs, for short). According to Blosh (2006), APIs can be the biggest assets of an organization because they can help to capture clients due to the clients' investment on buying, writing and learning them. Examples include Google, Twitter, and Facebook, to name a few (De Brajesh 2017).

As the use of an API increases and the development community that consumes this API expands, its management becomes increasingly more important (De Brajesh 2017; Oracle 2017). That is, API management is important to ensure that the IT services provided through the APIs are delivered faster, and with better quality. For instance, API management can provide insights into the use of APIs, including: geographical access patterns, types of devices used, latencies, key users, etc. Information on the use of APIs facilitates the decision making about product and service strategies of the underlying company (Zhenchang and Eleni 2007; Vásquez, 2014; Polak and Holubova 2015), as illustrated by organizations such as 3Scale (2017) and Apigee (2017). However, this management should be performed carefully to avoid problems in the application whose API is being monitored (Cibrán and Verheecke 2003; Bramantya 2015).

Given the importance of APIs in the modern world and the need to manage APIs, this paper describes a case study of API management in a Brazilian organization. By API management, we mean the process of enabling one to create, analyze, and publish APIs in a secure, scalable environment through flexible communication, life cycle management, developer attraction, and usage analysis (De Brajesh 2017). This paper describes the API management module that we developed and a case study evaluating it. This module has two goals: (i) to manage Restful Java APIs, and as a byproduct, (ii) to automatically create documentation of RESTful Java APIs. Our module was implemented using aspect-oriented programming (AOP) (Kiczales 1997), which means it was located in an "intermediate transparent layer" between the API and the client allowing the definition of the target resources to be monitored found in the API using Web Services Management Layer (WSML) (Verheecke, 2004). By using AOP we were able to create and execute our management module without affecting the organization and its APIs to be monitored. Once the target resources were defined, the designed aspects monitored HTTP requests directed to the API, extracted information from the input, output, and header parameters of the respective request, and stored this information in a database for later analysis. In addition to the cited parameters, it was possible to identify other useful information for API management including the source IP of the request, the execution time, the exceptions thrown at runtime-level including class and method that caused of each exception (De Brajesh, 2017). Furthermore, our approach also allows the automatic creation of API documentation following the recommendations provided by SPYREST (Sohan, 2015). Therefore, our

approach deals with two important aspects: API management and API documentation. Our module that implements our approach is called Sharingan.

We described a case study of Sharingan in a governmental institution in the state of Pará, Brazil. This organization is responsible for collecting taxes and performing financial management for the entire state and counts with a team of more than 100 IT professionals distributed in different teams. Sharingan was deployed in two APIs from December 2017 to March 2019. The results indicate the benefits provided by Sharingan – including helping the organization to easily identify information that could lead to U$ 2 million loss – and the interest of the organization in extending its usage.

The rest of this paper is organized as follows. The next section describes Sharingan, while the next one reports the case study. The following section describes the discussion of the case study results. Later, we describe the learned lessons and finally, the conclusions are presented in last section.

Sharingan

Sharingan is an API management module developed in Java using aspect-oriented programming, (AOP). The goal is to create a WSML (Bramantya, 2015) to monitor HTTP requests to the monitored APIs, as well as their responses to the clients. We also extract the information composed in the request "bodies". All this information is stored in a database and later can be displayed in a dashboard helping software engineers to make decisions about the managed APIs. Our module also addresses the difficulty of WSDL documentation, using SPYRest's approach (Sohan, 2015).

Requirements

Based on a brief literature review we conducted about API management, it was possible to identify that an API management tool, or module, should help organizations obtain insights into the use of its APIs, including access patterns, client device types, latencies, key users, and other relevant elements (De Brajesh, 2017). We report this as Sharingan's initial requirements that are detailed in Table 1.

Requirements	Description
R01	Extract content of the request and response, path of the resource used, source IP and API name
R02	Identify class and method that belongs to the resource
R03	Calculate resource processing time.
R04	Identify class, method, and cause of an error thrown at runtime.
R05	Create documentation in RESTful API context.

Table 1. Sharingan's requirements from the literature review

Implementation Details

Execution Flow

Sharingan's execution flow is based on the activity diagram shown in Figure 1. Simply put, the request information is extracted first, then it is observed if there is an exception (omitted from the diagram for simplification), and then the feedback information is extracted. Finally, all information is unified to be stored in a database and displayed for later analysis in the dashboard. During the requests, the monitoring module checks if the IP client is blocked through a table before storing any data.

Technology

Sharingan was developed in Java using Spring MVC 4 and Maven 3.5 and has two main modules. The first one, the API Manager, implements the actual management by extracting information from the monitored

API as well as its answer to API clients. The other module, called Dashboard, is responsible for displaying the collected management information using graphs, tables and JavaScript Object Notation (Json).

Sharingan testing was based on the Test-Driven-Development (TDD) approach to achieve consistency and cohesion of the implemented code. Through TDD it was possible to identify errors before the application was used in the case study. To automate the tests, we used the JUnit framework for performing unit tests.

Aspect Oriented Programming (AOP)

The main insight of our work was to use AOP, since this programming paradigm allows the separation of transversal interests of an application by introducing a new unit of interest, the so-called aspect (Kiczales, 1997). Aspects are first order elements to design transversal functional capsules in a clear way. Each aspect crosses over a specific functionality, so the main classes are not overloaded with secondary interests. The ability to allow the construction of transverse functionalities was essential for API management, i.e., the ability of not to interfere in the main flow of the application that uses the API. In other words, aspects allow us to avoid problems of modularization in the application whose API is being monitored (Bramantya, 2015; Cibrán and Verheecke 2003). As we will detail later, this was fundamental for the success of our case study because it allowed the usage of Sharingan without changes in the application to be monitored.

Some definitions are necessary at this point. Join points are the identifiable points of the AOP; the point cuts allow the gathering of several join points; advices execute point cuts; and, finally, aspects gather all the mentioned items. In Sharingan, an aspect that refers all targets through point cuts, join points and advices was created. Targets are all public methods of API interfaces that are under the @RestController annotation and have a HttpServletRequest object as a parameter. From the moment that a request for the API occurs, i.e., when a target is called, Sharingan monitors the behavior of the mapped resources (targets) and dynamically extracts the information inserted and generated by these target resources, so that at the end of the process this information is persisted. Finally, the method of the API executes the core flow of its business rule and returns a response to the client. Before the response reaches the client, Sharingan extracts information from the response, unifies this information with the request information, and persists all these pieces of information in the database.

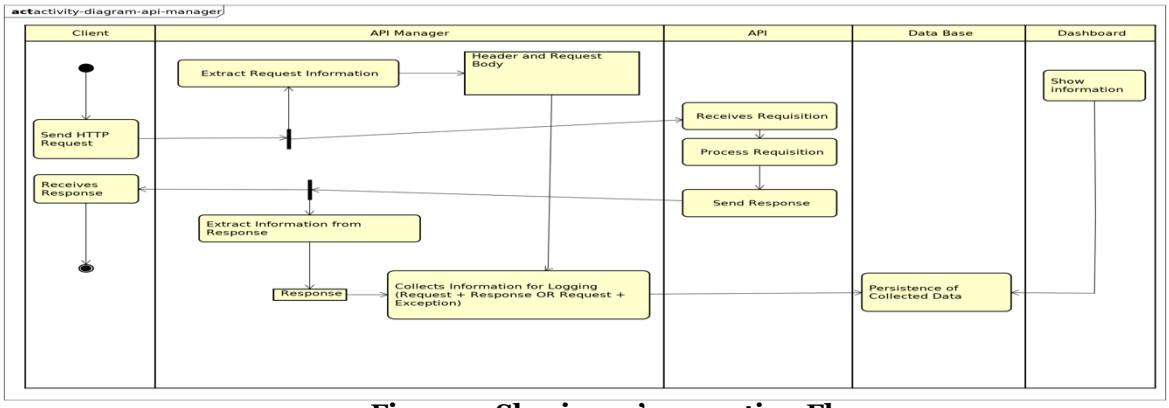

Figure 1. Sharingan's execution Flow

The Dashboard

After HTTP requests are monitored and stored into the database, the stored data will be available for study and analysis. The analysis can be realized through a dashboard. This dashboard graphically represents on a single screen different types of information and metrics. It allows one to centrally monitor the health of the API for data-driven decision-making. The dashboard helps both managers and technical team as it displays information related to the documentation of the monitored services, the errors that occurred during their execution, the summary of the metrics that need to be monitored and other information so that the API is always stable and available.

Figure 2 presents the daily access indicator since the implantation of Sharingan in the production environment of the case study we report. Details about this will be presented in the next section. This

graph visually assists the team to check the growth, or decrease, in the number of accesses of the API during a certain period, allowing for quick analysis of the API.

Figure 3 represents the number of accesses per each API method (target). This assists the team in the analysis of the access parameters regarding the importance of the method for its clients and may allow the discussion about the impact, or even monetization, of the most consumed API methods. Figure 4 presents the clients, and the largest client, who access the API. By combining information presented in Figures 3 and 4, a company can discover which resources are more, or less, used by a client, therefore inferring characteristics about the behavior of each client with the service (API method). In fact, in our case study we observed that the Brazilian organization did use this information.

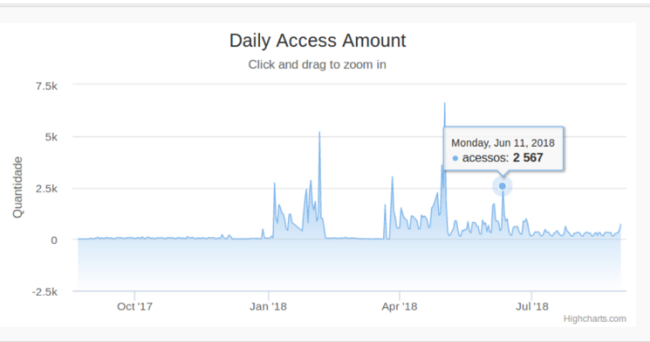

Figure 2. Daily Access indicator

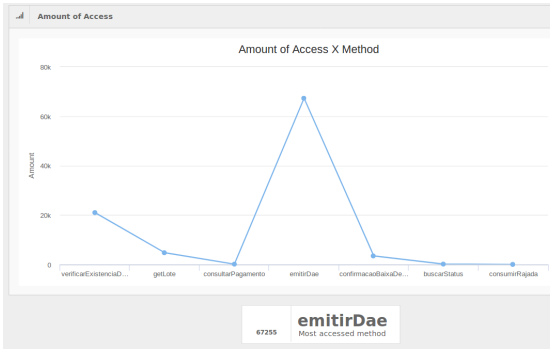

Figure 3. Amount of Access by Target Method

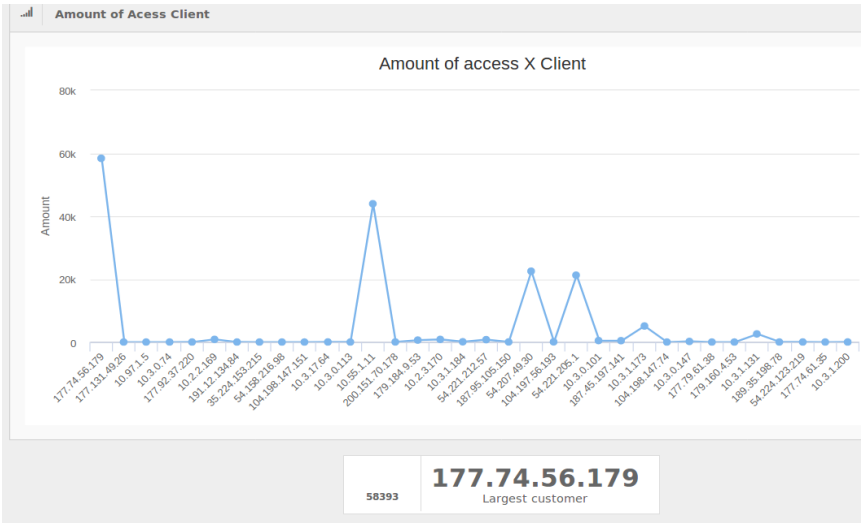

Figure 4. Number of Access by Client

Figure 5 describes a list of errors that occurred during the execution of the monitored API method, user behavior, or interaction with other systems. This information from the dashboard might help the technical team, for example, in quickly identifying the cause of an error, therefore avoiding the wasted effort in searching for information in log files, etc. The table basically describes the date at which the event was registered, an identification of the event, information about the cause of the event and the target method, and the method class.

Finally, Figure 6 shows the service documentation that is created by Sharingan and based on SpyREST (Sohan, 2015). In this case, the dashboard displays the resource name, the input data required by the resource, the expected output data, and the request header. The goal is to avoid constructing the documentation manually and to minimize the effort required for writing this documentation, avoiding both creation and updating effort (Sohan, 2015).

The Case Study

The Context

Sharingan is being used in a governmental institution in the state of Pará, Brazil. This organization is responsible for collecting taxes and performing financial management for a state government composed by 144 cities. The organization counts on a team of more than 100 IT professionals distributed in different teams, namely: database administrators, developers, business analysts, and technical staff related to networking, monitoring, infrastructure, configuration, support, project management, architecture, designers and IT services in general. Three application deployment environments are available to teams: (i) development is used by developers during the construction phase and also for unit testing applications; (ii) staging is responsible for integration tests and the validation of business rules by the stakeholders; and, finally, (iii) production is the environment in which applications are deployed and made available to end users.

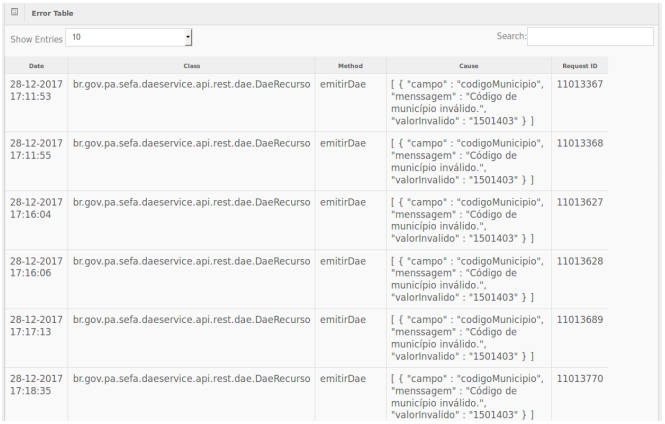

Figure 5. Error Table

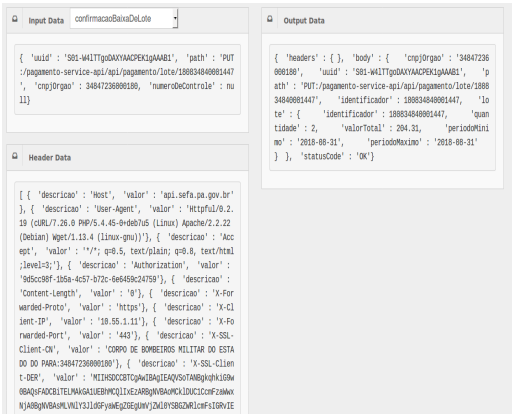

Figure 6. API Documentation

The goal of the monitored API in this case study is to generate the State Revenue Statement (or simply, DAE in Portuguese acronym) and check its payment status. The DAE is a PDF document that allows the payment of Brazilian taxes, revenues, taxable and non-taxable income; in short, it allows the collection of State taxes. The monitored API provides its resources (methods) to a number of government and private institutions that can issue DAEs. In the rest of this paper this API will be called the DAE-SERVICE-API.

The Methodology

In this section we described the methodology and the steps we performed to conduct the case study.

Defining the API to be monitored

The first step of our case study was an initial presentation of our prototype to the Brazilian organization in a group meeting. This meeting took place in January 2017 and included a technical team consisting of a general manager, a project manager, two software architects, a senior analyst, two full analysts and one database administrator. The presentation focused initially in the Sharingam management module. During the presentation, it was necessary to briefly introduce the concept of AOP to explain the context in which our API management approach fits in. Existing business solutions for managing APIs, including Oracle's solutions and Google Analytics, were also discussed at the meeting. Therefore, it was possible to verify the differences between these approaches and our tool. At the end of the meeting, the technical team realized that they were in fact interested in a solution for monitoring web applications and APIs. They concluded that our solution meets many important requirements of such a system including the collection of information about source IP addresses, classes and methods executed, execution time, quantity of accesses, content of the arguments of the body of the requests, content of the header arguments, raised exceptions and resource consumption. Given this scenario, the technical team agreed that the prototype could be the first module of the organization's audit solution set, with the proviso that new monitoring

parameters would be added if new API points needed to be monitored. Finally, it was decided that the case study would target DAE-SERVICE-API. The entire meeting lasted 2.5 hours with the participation of the first author.

Monitoring the DAE-SERVICE-API

After the initial meeting with the technical team, the prototype evolved to address a few additional requirements provided by this team. Then, Sharingan was incorporated into the project, the DAE-SERVICE-API as follows.

Initial tests with Sharingan

In addition to the unit tests described before, additional tests were performed in two application deployment environments. The first environment was the development deployment environment. In this case, database-related errors such as privileges in tables and procedures, and programming logic such as sign reversal, absence of attributes and Null Pointer Exceptions were identified. The second application deployment environment was the staging environment, where client applications use the API to test their functionality and, at the same time, the organization business analysts validate the expected result of the (i) API, and the (ii) client application. In short, we used this environment to validate Sharingan.

Sharingan's New Requirements

In the staging environment the stakeholders were able to evaluate the first results obtained through the monitoring of the DAE-SERVICE-API. Through this evaluation, new requirements emerged as described in Table 2 below. After the first evaluation the project returned to the development environment for the implementation of these new requirements. This was done in 15 days by the first author. After that, Sharingan returned to the staging environment. Details about this are presented in the next section.

Requirements	Description
R06	Identify which clients access the monitored API more frequently
R07	Display a dashboard with information about the classes, methods, and causes of the exceptions thrown.
R08	Extract information from the request header.
R09	Identify the most accessed method.

Table 2. Requirements based on the staging environment

The benefits that the Sharingan brought to the network, development, and support teams were highlighted in informal interviews and meetings with different members of these teams. These interviews were conducted by the first author at the Brazilian organization. Sharingan's strengths included low cost, modularity, flexibility in evolutionary maintenance and reduction of problem fix time, i.e., the time required by the teams to solve problems in the application whose API was being monitored. This last advantage occurred because the errors were quickly identified, since Sharingan registers the date and time of the occurrence of the error, the class that caused the error, the name of the method responsible for the error, the cause that led the application to post an error, and information about the request. According to the participants in the meeting, Sharingan helped all teams to fix bugs. Another advantage highlighted was the ease to provide documentation that Sharingan allowed. After Sharingan's deployment, it was possible to easily discover paths to the available API resources (methods). In addition, it was also possible to identify the input parameters needed to make a request, obtain the structure of a real response of the resource as well as the structure of the request header.

One point that was also mentioned by our informants was the impact of the monitoring module on the response time of the monitored application: it was observed that our module did not significantly increase the API processing time. Unfortunately, we were not able to collect information about the API processing time before our module was installed so that we could actually measure our overhead.

The final reported benefit by the informants was the identification of the most consumed resources (methods) by the API clients. This led the internal teams to be more cautious about changing these resources. In other words, the teams were aware of the potential impact (in number of clients) of changing these resources. It has also become easier to identify and elect a possible customer to perform the homologation of a given resource, since by using Sharingan it was possible to view the clients of the DAE-SERVICE-API that most consumed this resource. This is available in Sharingan's Dashboard through a chart that represents the number of accesses per client of the API (see Figure 4). In short, this chart allows one to identify the best customers for validating a new feature in the API.

Regarding the disadvantages, it was observed the need to replicate changes in Sharingan's code for all the projects interested in using it. That is, if Sharingan evolves, it is necessary to apply the changes in each project that uses it.

Sharingan in the Production Environment

The result of the meeting with our informants indicated that Sharingan was successful in managing the HTTP requests directed to the DAE-SERVICE-API and in generating information through the collected data for identification of errors, documentation, reports; decision-making in general. In short, Sharingan was considered mature enough to be migrated to the production environment and, therefore, start managing clients' APIs through requests from real users. So, Sharing went into production on December 28, 2017.

Just as in the staging environment, informal interviews with developers of the organization reported that the impact of Sharingan in the time necessary to process a request remains insignificant. These meetings were again conducted by the first author at the Brazilian organization. The documentation became useful for the creation of customer integration manuals with the service provided, through real examples of expected input and return data. In other words, Sharing did support the documentation of the API, therefore facilitating developers' work.

Until August 29, 2018 Sharingan monitored APIs calls from 24 different real customers. During this period, Sharingan was able to register 323,564 requests, in which 303,764 were successful, 117,114 presented errors on the client side and 2,490 presented errors on the API side. Furthermore, during this evaluation period there was no record that indicated bugs in our module.

In particular, there is one example that suggest the success of our approach. A particular customer generated the DAEs, paid them, but was not able to find information associated with the payment. Since the organization was monitoring the API calls using Sharingan in both the staging and production environments, it was possible to quickly identify that the problem was on the customer side: they were generating the DAEs in the staging environment instead of the production environment. With that information, they could contact the customer and address the issue. This was a huge help for the organization since these "lost" payments were in the order of about U$ 2 million!

In general, according to the developers we interviewed, using Sharingam allows the organization to create API that are consumable and secure. In addition, Sharingam allows users to monitor API usage, load, transaction logs, historical data, and other metrics that better inform the status as well as the success of the monitored APIs.

Discussion

As mentioned and illustrated by the example from the previous sections, Sharing has brought many important benefits. Because of that, it is currently being deployed in the production environment to monitor one specific API (the DAE-SERVICE-API, which is arguably one of the most important APIs from the organization) and is in the staging environment monitoring a total of four other APIs.

We believe the benefits provided by Sharingan are only possible because we used aspect-oriented programming to develop it. The adoption of AOP prevented the application to be monitored from being changed, since, using aspects allowed us to create and execute our management module without affecting APIs to be monitored, and more importantly, without disturbing the organization, its employees and end

users[1]. In other words, it was possible to perform the monitoring process orthogonally to the business flow of the API being monitored. That is, the main application code already existed and the monitoring module was added without interfering with the code that was already developed and stable. In fact, according to our informants, that was one of the main reasons why Sharingan was regarded as successful by the organization where the case study took place.

Our approach allows capturing information about API requests. However, in addition to these requests, using AOP it is also possible to capture programming exceptions and errors that occurred in the API at runtime. In fact, during the case study, we observed the importance of catching errors at class level during the runtime. This functionality is directly responsible for reducing the bug fixing time at the organization, since it is possible to easily identify the cause of the error through Sharingan. This allows organization's developers to worry only about fixing the bug instead of wasting time searching for log files in different parts of the server. Such benefit has become a hallmark of Sharingan's API monitoring approach since it has not been observed in other studied solutions.

Finally, as we illustrated, we are also able to identify the actual source of the request; and to capture the execution time of resources; the most used resources; customers who use the API more often; and the number of accesses per day.

De Brajesh's (2017) discusses the idea of API management as the process of enabling one to create, analyze, and publish APIs in a secure, scalable environment through flexible communication, life cycle management, developer attraction, and usage analysis. In special, API life cycle management approaches API creation with tests on an environment before the API is published for production. In addition, it provides tools that can be used to migrate APIs from less important environments to production environments while providing the capability to manage multiple versions of an API and its eventual retirement. Regarding API usage analysis, the goal is to provide information to make future decisions about the API, for example: activity logging, user auditing, business value reports and service level monitoring. In our case, Sharingan provides support for API usage analysis, but not for the entire life cycle management. From an industrial point of view, API management can be observed in solutions using gateways and enterprise buses for script injection that allow monitoring API requests. As expected, these solutions are mostly closed and very expensive.

Regarding API documentation, SPYRest (Sohan, 2015) facilitates the process of API documentation by monitoring API requests and extracting information from the input, output, and header parameters of the respective request, as well as removing confidential data from the captured information. Sohan's proposal also publishes API readable API documentation for developers writing applications that consume data from this API (Uddin, 2012). Sharingan has a similar proposal to SPYRest: the documentation is constructed automatically, therefore it is possible to generate a comprehensive documentation in a simplified way. In addition, the documentation is always updated because, regardless of the API version, the documentation is always created in relation to the version of the API in use. This helps developers of client applications by providing real cases of API usage and facilitates testing for integration with API features. The difference between SPYRest and Sharingan is that our tool is not a proxy application and presented greater flexibility to select which resources would be monitored because AOP allow us to monitor methods, classes, and even attributes.

Finally, Sharingan use a WSML (web service manager layer) for monitoring RESTful calls using AOP techniques. In fact, this is a different approach for API management. This approach consists of an intermediate layer located between clients and (API) services. This layer manages the HTTP requests to the API services and extracts all information for the management in a way that it is independent of the code of the (API) service (Bramantya, 2015). However, this approach is limited regarding the documentation because WSML does not support the specification of documentation for web services.

In general, in addition to AOP, Sharingan combines approaches from SPYRest (Sohan, 2015) and WSML (Bramantya, 2015) to provide an independent layer for API management and automatic generation of API documentation.

[1] To be more precise, the only change to the main application was its pom.xml file, i.e., a configuration file that concentrates dependencies at the library level of the project.

Ii is important to discuss a possible alternative to AOP. In this case, the organization could monitor API calls through http(s) requests in the organization's routers / firewall. In fact, the organization reported that they have done that in the past. However, this approach does not detect run-time errors[2] since it does not have any associated semantics. In other words, because aspects are associated to particular parts of the code, monitoring can be even customized to understand and monitor the sequence of API calls (Silva 2014), among other aspects. We are discussing with the organization these possibilities.

Another limitation of our approach is the fact it is dynamic: it monitors and generates documentation based on the API calls, i.e., it is limited to the method activations that took place during these calls. If a target method is not called, our approach will not generate documentation associated to it. In contrast, a static approach that analyzes the source code (like VCC (Silva 2014)) takes into account all possible paths a method activation can take. Since our approach focuses more on API monitoring, we believe our approach is appropriate. In fact, one could even use both approaches: the static one to analyze and identify errors in the API and a dynamic (like we propose) to effectively monitor the API.

It is good to make clear, that unfortunately there was no opportunity to perform the measurement of processing time without the management module, due to the quality of the information that was emerging through the monitoring, until the moment this paper was written.

Conclusions and Future Work

This paper presented a monitoring module for RESTful APIs called Sharingan. Sharingan was built using aspect-oriented programming (AOP) to be less intrusive in the applications being monitored. Alongside the description of this module, we described a case study using Sharingan in a governmental institution in Brazil. This organization adopted Sharingan to monitor one of its mostly used APIs. After using this module in the development and staging deployment environments, many advantages were observed, including the identification of the most consumed resources by API clients. This allowed internal teams to be aware of the potential impact changes in the API would cause, allowing these teams to perform more appropriate planning, development, and testing of changes in the resources (methods) of this API. Due to its advantages, Sharingan has been used in the production deployment environment with some of the same benefits observed in the other environments. We have been in contact with this organization that plans to adopt it more widely. So, new case studies are being planned.

For future work, we have noticed that Sharingan quickly collects a large amount of information about how customers use an API. As of now, this information is useful for developers and managers only through the dashboard. We believe we can make Sharingan even more useful by applying machine learning algorithms to automatically identify trends and discover patterns. For instance, we are investigating the usage of the PLWAP algorithm (Ezeife. 2005) to identify a list of patterns (frequent method calls), accompanied by their support.

Acknowledgements

This research has been partially funded by the Brazilian National Council for Research and Development (CNPq), under research grants 420801/2016-2 and 311256/2018-0.

REFERENCES

Blosh, J.: How to Design a Good API and Why it Matters. Companion to the 21st ACM SIGPLAN OOPSLA, p506 -507 (2006).

Chesbrough, H. Open Innovation: The New Imperative for Creating and Profiting from Technology. Harvard business school press, pp. 227 (2003).

Fielding, T.: "Architectural styles and the design of network-based software architectures" Ph.D. dissertation, University of California (2000).

[2] As we discussed in the Case Study section, the organization found this a very important aspect provided by Sharingan.

Software Engineering Institute.: C4 Software Technology Reference Guide: A Prototype Available in https://resources.sei.cmu.edu/asset_files/handbook/1997_002_001_16523.pdf. Accessed on December 8, 2017.

3Scale.: What is an API? Your guide to the internet business (r)evolution. Available in https://www.3scale.net/wp-content/uploads/2012/06/What-is-an-API-1.0.pdf. Accessed on December 8, 2017.

Redwrite.: What APIs Are And Why They're Important. Available in em https://readwrite.com/2013/09/19/api-defined/. Accessed on December 8, 2017.

Santos, W.: Programmable Web API Directory Eclipses 17,000 as API Economy Continues Surge. Available in https://www.programmableweb.com/news/programmableweb-api-directory-eclipses-17000-api-economy-continues-surge/research/2017/03/13. Accessed on December 8, 2017.

Amer D.: CatchUp!: Capturing and Replaying Refactorings to support API evolution, Proceedings of the 27th ICSE, p15-21 (2005).

Zhenchang, X, Eleni S.: API-Evolution Support with Diff-CatchUp, IEEE Transactions on Software Engineering, p.818-836 (2007).

Vásquez, M. L.: Supporting evolution and maintenance of Android apps. Companion Proceedings of the 36th ICSE, p714-717 (2014).

Sohan, S M., et al.: SpyREST: Automated RESTful API Documentation using an HTTP Proxy Server. International Conference on Automated Software Engineering, p271-276 (2015).

Tilley, S. R., et al.: Documenting software systems with Views'. Proceedings of ACM sigdoc'92 p211-219 (1992).

Polak, M., Holubova I.: REST API Management and Evolution Using MDA. in Proceedings of the 8th International C* Conference on Computer Science & Software Engineering, p102-109 (2015).

Oracle: A Comprehensive Solution for API Management. Available in http://www.oracle.com/us/products/middleware/soa/api-management-wp-2490331.pdf. Accessed on December 8, 2017.

Kiczales, G., et al.: Aspect-Oriented Programming. 1º Edição. Springer Berlin Heidelberg. p220 – 242 (1997).

Laddad, R., Johnson R.: AspectJ in Action: Enterprise AOP with Spring Applications. 2º Edição. Manning Publications, p550 (2009).

Uddin, G., et al.: Temporal Analysis of API Usage Concepts. In Proceedings of the 34th ICSE, p804-814 (2012).

Espinha, T., et al.: Web API growing pains: Stories from client developers and their code. in Proc. of IEEE ICSM, p 84–93 (2014).

Apigee: API Manager. Available in https://pages.apigee.com/rs/351-WXY-166/images/apigee-ebook-api-mgmt-2015-07.pdf. Accessed on December 8, 2017.

Bramantya, W., Kusumo D. S., Munajat B.: Modularizing RESTful web service management with aspect oriented programming. 3rd International Conference on Information and Communication Technology (ICoICT), p358-363 (2015)

Cibrán, M. A. e Verheecke B.: Modularizing Web Services Management with AOP. In Proceedings of The 1st European Workshop on Object Orientation and Web Services (ECOOP'2003) held in conjunction with The 17th ECOOP, p14 (2003).

Tapscott, D., Williams A.: Wikinomics. Como a colaboração em massa pode mudar o seu negócio. Editora Nova Fronteira, p286 (2006).

Rebellabs: https://dl.zeroturnaround.com/?token=a00210107f1f238cf396b71472f260afadb8e06c . Accessed on December 8, 2017.

Verheecke B. Cibrán, M. A. Vanderperre W. Suvée D. Jonckers V.: AOP for Dynamic Configuration and Management of Web Services. In International Journal of Web Services Research, p 25-41 (2004).

De Brajesh. API Management: An Architect's Guide to Developing and Managing APIs for Your Organization. Apress, p195 (2017).

Silva Jr. L . L. N; Plastino, A.; Murta, L. What Should I Code now? Journal of Universal computer Science 20 (5): 797-821 (2014).

Ezeife, C. I., Lu, Y., Liu, Y.: "Plwap sequential mining: Open source code"; International Workshop on Open Source Data Mining: Frequent Pat- tern Mining Implementations. Chicago, USA; (2005), 26–35.