Computer Science and Information Systems 14(2):447–466 DOI: 10.2298/CSIS161022009R

REST API Example Generation Using Javadoc

Petri Rantanen[1]

[1] Pori Department, Tampere University of Technology, Pori, Finland,
PL 300, 28101 Pori, Finland
petri.rantanen@tut.fi

Abstract. Formatting and editing documentation can be a tedious process regardless of how well your documentation templates are made. Especially, keeping the code examples up-to-date can be time-consuming and error-prone. The research presented in this article describes a Javadoc extension that can be used to produce example data in combination with automatically generated API method call examples, and explains how the APIs in our implementation are organized to further ease the automatic documentation process. The primary goal is to make generating method call examples for (RESTful) web services easier. The method has been used in the implementation of a media content analysis service, and the experiences, advantages of using the described approach are discussed in this article. The method allows easier validation and maintenance for the documentation of method usage examples with a downside of an increased workload in the implementation of software components required for the automatic documentation process.

Keywords: documentation, programming techniques, REST, examples.

1. Introduction

The background for the studies presented in this article lies in a research program, which had the goal of implementing a complex content analysis service [1][2][3]. The service could be used to categorize and classify users' multimedia content, such as videos and photos, using automated content analysis technologies developed by the program partners. Our task was to design and implement Application Programming Interfaces (APIs) for a front-end service accessed by clients and content analysis back ends. The preparation of manuscripts which are to be reproduced by photo-offset requires special care. Papers submitted in a technically unsuitable form will be returned for retyping, or canceled if the volume cannot otherwise be finished on time.

The problem was not only with the design of the APIs, but also how to keep the specifications up-to-date, easily accessible and how to create usable examples. In our case, we had a relatively small development team (less than ten developers) working with the specification and implementation of the core service. The development work was primarily done in a university environment, which also meant that we had very limited resources available for quality control. Still, developers from other companies and universities participating in our research project used our APIs and modified their systems based on our specifications, which required us to keep the provided examples up-to-date. In practice, this turned out to be a very tedious and error-prone process. Our APIs had a total of about 50 representational state transfer (REST) and remote procedure call (RPC) methods with many of them utilizing long and complex Extensible Markup Language (XML) payloads. The payloads and method call examples could not be guaranteed to be correct when made by hand and errors in the documentation were repeatedly reported by the API users. Naturally, a larger company might have more resources at their disposal, but still, our approach should provide assistance or a starting point for other that are struggling with their documentation process or are planning to improve their development process in general.

The reasoning behind the approach presented in this paper is in the fact that if an (online) interface exists, it must in itself require requests and produce responses, which could be directly used as example cases in the documentation. And by nature, these request and responses provide correct example code, because they are generated by the actual API methods. Utilizing the extension capabilities of the standard Javadoc tool, it is possible to modify the process of documentation generation in such way that these interfaces are directly called when the documentation is generated and request-response pairs are embedded within the surrounding automatically generated documentation.

The primary contribution of this article is the description of how the RESTlet extension for the Javadoc tool is used to generate documentation – and more specifically, to automatically generate web API call examples, which have traditionally been made by hand by the application developers. Additionally, an example generation API is described, which may be required in some cases. Implementing the example API may provide other additional benefits for developers even without the use of the RESTlet, and these benefits (creation of test data for developers and the utilization of the API in data format documentation) as well as possible downsides (increased work in the initial design and implementation of the APIs) are discussed in the following sections. A link to a minimal implementation of the RESTlet extension is provided in the references [4] though the purpose of this paper is not to provide a full-featured tool for documentation, but to illustrate how the standard Javadoc tool can be easily utilized for the generation of method call examples.

The structure of this article is as follows. The section 2 gives the background of automatic documentation generation in general and an overview of related studies. The section 3 presents an example use case, which illustrates what is the goal of the approach presented in this article. The example is loosely based on our real use case of content analysis system, but the data formats and request and response examples are simplified and parts unnecessary for the understanding of the presented method are removed. The sections 4 and 5 describe the interfaces designed to realize the documentation process, and the process itself is explained in the section 6. An example of the generated HyperText Markup Language (HTML) documentation can be seen in the section 7. The chosen approach does have certain limitation and disadvantages, and these are discussed in the section 8. The section 9 provides potential future directions for the research. Finally, the section 10 concludes this article.

2. Background

The first versions of our interface specifications were Word and Portable Document Format (PDF) files, which contained the API methods, use cases, and examples of how to use the API. This approach, in combination with HTML web pages, is a very common one for the documentation of web APIs [5], such as RESTful web services. The approach does have a couple of obvious issues. Firstly, there is the constant task of formatting and editing the documentation, which always seems to be a tedious process regardless of how well your documentation templates are made. The second issue is keeping the example code up-to-date, in this case, the REST API method call examples. The second problem is a very common one, and research generally confirms that documentation is often lacking or outdated [6][7[8]. Examples are also one of the most important sources of information and that missing examples can be a serious obstacle to learning how to use a specific API [5][9][10][11]. Naturally, it follows that automatically generated documentation in combination with examples would make things much easier, and studies show that majority of web APIs provide examples for developers [12]. In many cases the usage examples with example code, required data formats and recommended parameters are the most usable support for developers, but often these are also the most tedious parts of the documentation to keep updated. If only method declarations and parameters are to be documented without usage examples, this can be easily enough achieved with standard Javadoc, but it can be challenging to find tools for automatically generating usable examples. The research also suggests that successful frameworks often use automated builds to guarantee more consistent and better maintained documentation [12]. The solutions should have low overhead [13], be simple enough to remain useful and preferably be something that the programmer can implement whilst writing the source code. Additionally, research suggests that for maintenance, the source code is the most important artifact [14], so it makes sense to assume that whatever the solution is, it should be implemented on the source code level, and not somewhere externally.

Recent studies have explored how to link examples and method descriptions found on the Internet to API documentation that does not by itself contain example code [8][15][16], though in our case and in the case of many smaller and less known developers, it can be very difficult to find pre-made examples, which could be used with documentation.

Methods exist for using specific markups inside the source code alone or in combination with the use of external models to enable the generation process. Examples of these are the Javadoc tool [17] provided in the Java Software Development Kit, the Swagger framework [18], and the tools for documenting method descriptions in the Spring framework (such as wsdoc [19], RESTdoclet [20] and SpringDoclet [21). Our RESTlet implementation also takes advantage of code markup - in our case, the comment tagging mechanism of the Javadoc tool.

The tagging approach of the apiDoc [22] documentation generator is similar to ours. It uses source code tags to define requests and responses though these are essentially pre-defined “macro expansions” that are used to populate the final documentation, and they cannot be used to generate the example code dynamically. The documentation itself is created by utilizing an external application as opposed to using the Javadoc tool.

The basic idea behind the SpyREST tool [23] shares certain similarities with our approach. The primary component of the tool is the Hypertext Transfer Protocol (HTTP) proxy server, which is used to intercept actual API calls performed by the API users, developers or by automated scripts. The collected data is then used in the generation of the API documentation. In our use case, we propose creating and collecting the example requests and responses when the documentation is created with the standard Javadoc tool removing the need to use a separate proxy server.

Another approach is to perform code analysis to decipher the logical structure of the source code automatically and use this information for example generation [24] or for the creation of method summaries [25][26][27]. This kind of analysis is not trivial to implement, and discovering the higher level information and semantics for complex systems might not be realizable in practice [28]. Additionally, for web-based systems this can be problematic because the entire source code or the internal structures of the application are not known, or the API usage patterns cannot be easily detected by analyzing the service interfaces. Especially in RESTful web services, the use of Hypertext Transfer Protocol (HTTP) enables greater freedom for the implementation of server and client components, but it also makes it more difficult to detect the method call hierarchies and patterns automatically, and to generate examples that illustrate how the information exchange between the two participants works. This makes it challenging to utilize existing solutions in the web based world, and thus, alternative approaches are required.

There seems to be an apparent lack of research that target *example generation* for web-based services even though web development and the utilization of public APIs have become a de facto means of designing and implementing end-user services. Furthermore, based on the research it seems that most of the existing tools rely on: the utilization of external applications; manual definition of request and response examples by the developers; or require extensive frameworks, which may or may not be compatible with the preferred development environment of the application developers. Our approach attempts to provide a method, which does not interfere with other development tools and is relatively simple to implement yet provides guaranteed correctness of the generated examples.

3. Example Case: Photo Analysis

The case of photo analysis is chosen in this paper to illustrate how the presented approach can be utilized in generation of example code. The example assumes that Java or Java-compatible programming language is used to implement the service, which is a very common use case in today’s web environment. The example is loosely based on our real use case of content analysis system, but the data formats and request and response examples are simplified and parts unnecessary for the understanding of the presented method are removed. Similarly, the method and object names used in this example case (and in the following method call examples) are simply for illustration purposes and the approach is not limited for the API method names or object types presented in the example.

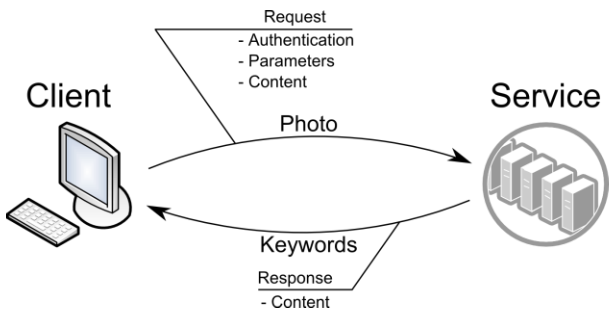
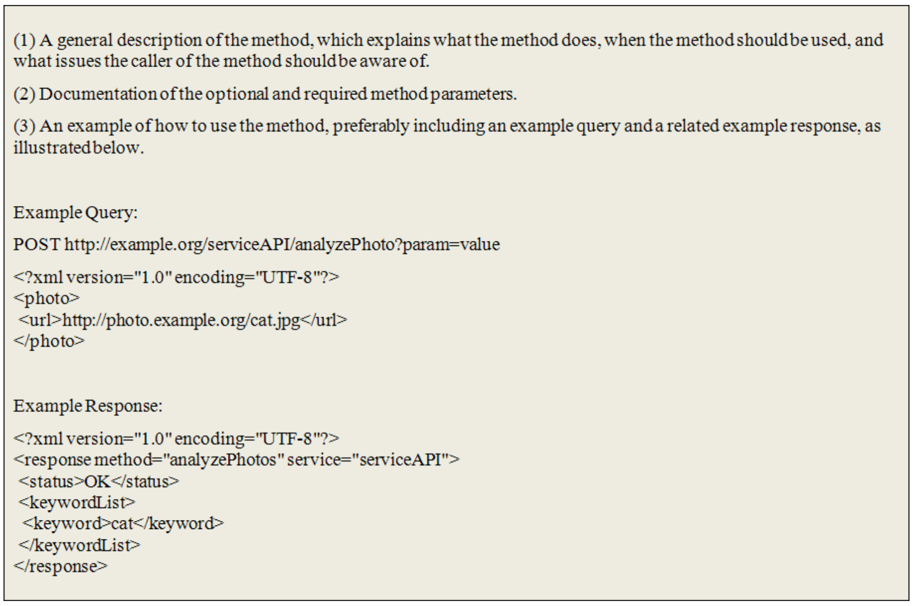


Fig. 1. Illustration of a basic HTTP request

In this case, the photo analysis service implements a single interface, which can be used to extract and generate keywords based on the provided photo. The Fig. 1 shows this relatively simple HTTP operation with the crucial components included. The request, in general, contains: user details (or lack of it, for anonymous access) in appropriate authentication scheme (e.g. HTTP basic, tokens, certificates); the method parameters, in this case, as part of the request Uniform Resource Identifier (URI); and the actual content in the HTTP body consisting of a list of URLs in XML format for accessing the photos. The response in turn contains the list of generated keywords also in XML, and a status code.


(1) A general description of the method, which explains what the method does, when the method should be used, and what issues the caller of the method should be aware of.

(2) Documentation of the optional and required method parameters.

(3) An example of how to use the method, preferably including an example query and a related example response, as illustrated below.

Example Query:

POST http://example.org/serviceAPI/analyzePhoto?param=value

```
<?xml version="1.0" encoding="UTF-8"?>
<photo>
 <url>http://photo.example.org/cat.jpg</url>
</photo>
```

Example Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response method="analyzePhotos" service="serviceAPI">
 <status>OK</status>
 <keywordList>
  <keyword>cat</keyword>
 </keywordList>
</response>
```

Fig. 2. A simplified example of method documentation

In well-made method documentation, each of the components of the operation is explained, with the exception being the authentication scheme, which is often global for the entire service and might be documented separately (which is also the assumption in this simplified example). Based on the rest of the components a very basic documentation template can be created consisting of three parts, which can be seen in the Fig. 2 above.

The generation of parts (1) and (2) for the documentation can be relatively simple. Assuming that the programmer has properly documented the code, these can be extracted from method comments, and the standard Javadoc tags can be used to define the first two parts. In principle, as the program code includes the data types (Java classes) for the method input parameters as well as for the method return value, this information could be used to create the request and response examples. In practice though, it can be very difficult to decide, based on the analysis of the code structure, how the objects should be initialized and what kind of data they should be populated with in the scope of a specific method.

Fortunately, there is a simpler method for (partially) generating the part (3) of the documentation example: by *calling the API method*. The service implementation can process the method call and will provide response, which can be directly embedded into the generated documentation as a response example. To realize this kind of functionality, three primary components must exist: an actual or mockup implementation of the service API, called *Reference API* in the context of this paper; an interface that can generate the example data utilized in the method call to the Reference API, called *Example API*; and a method of embedding the example request and response pair into the automatically generated documentation, implemented in our approach as an extension to the standard Javadoc tool. In the following sections, the use and implementation of the Reference API, the Example API and the Javadoc extension called RESTlet are explained.

4. Reference API

The generation of valid responses is basically the only requirement for a functioning Reference API, thus, it is possible to directly use the existing Service APIs, if these can be organized so that they produce usable responses when the documentation is generated. In fact, creating a test configuration of the Service API for the generation of the documentation is the preferred solution. Using the Service API would, in general, require less work in the implementation of the Reference API, and it would guarantee that the Reference API (testing version) and the Service API (production version) do not differ in their functionality and implementation. In any case, this means that there should be a running version of the system available when the documentation is generated.

In some cases, it may not be possible to use the actual service APIs for the generation of the response examples. It is possible that the production version of the API is not available, when the documentation is generated, or the use of the API requires data (such as real user details) that cannot be accessed in a test environment, or the test environment for whatever reason does not produce examples in clean and usable format. It is also possible that the execution of the calls to the service would take a long time considerably increasing the total time required to produce the documentation. For example, in the example use case it could take a long time to analyze a certain set of photos.

In our actual implementation of the content analysis service, the Reference API is a dummy implementation of the specification for testing purposes. It is comparable to the "code playgrounds" provided by many companies [29][30] and organizations [31] although we do not provide any premade web page for testing, but only the API endpoint. In our case, the implementation is entirely handmade to conform to the specification, but if there is already a fully functional implementation, it is also possible to use one of the many existing libraries (such as Easymock [32, jMock [33], and Mockito [34] to produce a mockup implementation of the API to work as a Reference API. Automatically generating the mockup methods also helps to keep the Reference API up-to-date with the Service API implementation.

Naturally, the mockup implementations only provide structurally valid responses. For example, querying a Reference API method *analyzePhoto* with a photo of a cat would return a valid list of keywords, but it does not guarantee that the keywords actually relate to cats. Similarly, calling the same method twice (for example, HTTP DELETE on a resource) would return an identical response for both calls as long as the syntax and format of the request is valid regardless of the provided resource identifiers. Also, certain parameters such as paging and sorting may not make logical sense when observing the responses. Generally, this is not an issue, as the examples are meant to illustrate the syntax, usage and functionality of the API and these aspects can be seen in the examples regardless of the actual content. If a dummy implementation is used instead, the API can also be manually hard-coded to provide responses better suited for the example in question. Hard-coding might increase the amount of manual work, but in general, the definitions made on program code level are easier to test, making the process less error-prone than documenting the examples by hand. If "logical correctness" is required, the test environment used for the generation of the documentation should be populated with data that produces the required responses when the API is called.

In our case the designed service is RESTful, but this does not mean that the method could not be used in other web-based services, for example, for the documentation of more RPC-like services. The approach can be used to document both asynchronous and synchronous methods, but the documentation for asynchronous methods would be split into two different places within the documentation: to the place where the primary method to be called is located; and to the place where the callback method is documented. This means that manual documentation work is required to refer one method from another in the documentation.

5. Example API

The Example API serves three use cases. Firstly, it can be used to generate object examples, which can be directly used when testing applications. The users and developers of the service API could read the documentation or study related XML schemas, but in practice, it makes understanding the data formats easier if pre-made examples are available. The Example API makes it possible to generate structurally valid example data to be used in the application development process. Additionally, this offers a solution for data formats that do not provide standardized means of validation (e.g. JSON), or the methods to generate structured data examples from schemas.

Secondly, the examples can be embedded into the data format documentation using the RESTlet extension. Thirdly, and most importantly in the context of this paper, the Javadoc doclet expansion (called RESTlet) uses it to produce the workloads required to perform the HTTP requests for example generation – that is, to produce the HTTP body of the requests targeted to the Reference API.

```
GET http://example.org/exampleAPI/photo

<?xml version="1.0" encoding="UTF-8"?>
<response method="photo" service="exampleAPI">
 <status>OK</status>
 <example>
  <photo>
   <url>http://photo.example.org/cat.jpg</url>
  </photo>
 </example>
</response>
```

Fig. 3. Simplified Example API request

Fig. 3 shows a simplified example request to the API in XML format. The URI path uses the “standard” REST convention as described by Fielding et al. [35][36]. The last part (*photo*) is the root element name of an object (or the target resource, as also specified by the URI), which could be replaced with any valid object name defined by our specification. In practice, the *photo* could be replaced with, for example, *video* for retrieving an object representing a single video file. The example in Fig. 3 does not contain any additional parameters, but these could be utilized for filtering the elements present in the output, or for changing the output format to JSON.

All methods in the Example API are HTTP GET methods. There are three reasons for this. Firstly, and most importantly, it makes it much easier to use the web browser to generate example code, and secondly, it helps to keep the RESTlet implementation simpler. The third reason is that all of the methods in the Example API are stateless and read-only, making it logical to use only GET. Thus, the Example API is implemented as any other RESTful service and the actual implementation could be made using any framework or solution the developers prefer.

The returned objects are pseudo-randomly generated based on the Java classes, and the same classes can be used in other testing outside the context of the Example API, such as in unit tests. The randomization (and serialization) process, in our case, takes advantage of class annotations (JAXB [37] for XML and GSON [38] for JSON) that define which member variables should be included in the (response) example. The process itself uses a simple randomization based solely on the Java class member type. For example, in Fig. 3, the *url* element of *photo* could be defined as a string. In that case, every call to the API method would return structurally identical responses, but the value of *url* would be a different string each time. Variations in the values cause the side-effect of slightly changing the generated example documentation each time the documentation is generated. In our use case this is irrelevant, and it would be trivial to modify the interface to produce identical objects on each time either by adding a parameter, which would control the randomization process upon a method call, or by hard-coding the API to always return the same data. In some cases, it might also be preferable for testing purposes to generate examples with variations in the data.

6. The RESTlet Extension

RESTlet is a Java program that uses the doclet API to specify the content and format of the output of the Javadoc tool [39]. The RESTlet tag implements a Javadoc interface called Taglet. In the source code comments (and also in the example, in Fig. 4 below) it is referred to by the name “doc.restlet”. The RESTlet tag is meant to be used as an inline tag, so the tag has to be enclosed inside curly brackets – this also neatly separates the tag from the surrounding comment block. We chose to use the XML attribute styled scheme for passing on parameters, and Fig. 4 illustrates an example of the RESTlet tag usage. In practice, depending on the framework or library utilized in the service implementation, other parameters or annotations might be present in the Java method declaration in combination with other standard Javadoc tags, but none of these have any effect on the functionality of the RESTlet extension and are not shown in the example below. In essence, the RESTlet is a simple application, which implements a text parser for processing the attributes defined in the doc.restlet tag and also implements an HTTP client – in our case, the Apache’s HTTP Client for Java [40] – for accessing remote URIs.

```
/**
 * This is an example of a method that takes a photo and returns a list of keywords.
 *
 *{@doc.restlet
 * service="referenceAPI"
 * method="analyzePhoto"
 * query="param=value"
 * body_uri="/exampleAPI/photo"
 * type="POST"}
 *
 * @param param You can pass a value for this parameter.
 * @param photo Details of the photo to be analyzed.
 * @return A list of keywords.
 */
public KeywordList analyzePhoto(String param, Photo photo) { … }
```

Fig. 4. Example of RESTlet tag usage in Java source code

The *service*, *method* and *body_uri* attributes refer to the relative paths within the service. The separation of *service* and *method* is mainly a style decision; they could just as well be combined into a single attribute, for example, *rest_uri*. The construction of the full URI is a simple string concatenation operation, and how the URI is divided into a base URI and a relative URI can be freely chosen based on the developer’s preference. The use of relative URIs is designed to make the code comments easier to modify and to keep them up-to-date, as well as to make the definitions look tidier in the source code files. Similarly, the *query* attribute, which defines the query parameters for the request could be directly appended to the value of the *method* attribute, separated by the commonly used question mark operator. The *body_uri* attribute, if present and contains a value, defines if a data from the given URI should be retrieved and used as the HTTP body for the final request. In this case a simple non-parameterized URI is used, but if more specific example object is required (and the Example API supports it) the request URI could include parameters for controlling the object construction.

The *type* attribute defines the HTTP operation type (e.g. GET, POST…). The proposed HTTP standard [41] dictates certain limitations for which HTTP methods can contain a body – for example, HTTP HEAD can never contain a body. Checking if *body_uri* can be present in combination with the given *type* attribute value is relatively simple and can be easily performed to prevent mistakes in the tag definitions. In the RESTlet implementation, GET is the only allowed type of method call for the *body_uri* and the request cannot contain body data. It would be possible to define attributes for the method type of the body request call, and for the definition of the body data, but this would make the implementation more complicated, and it would also make the taglet code inside the Java comments more complex and thus harder to read, maintain, and use. Also, in our case, the *Example API* only has methods that use GET.

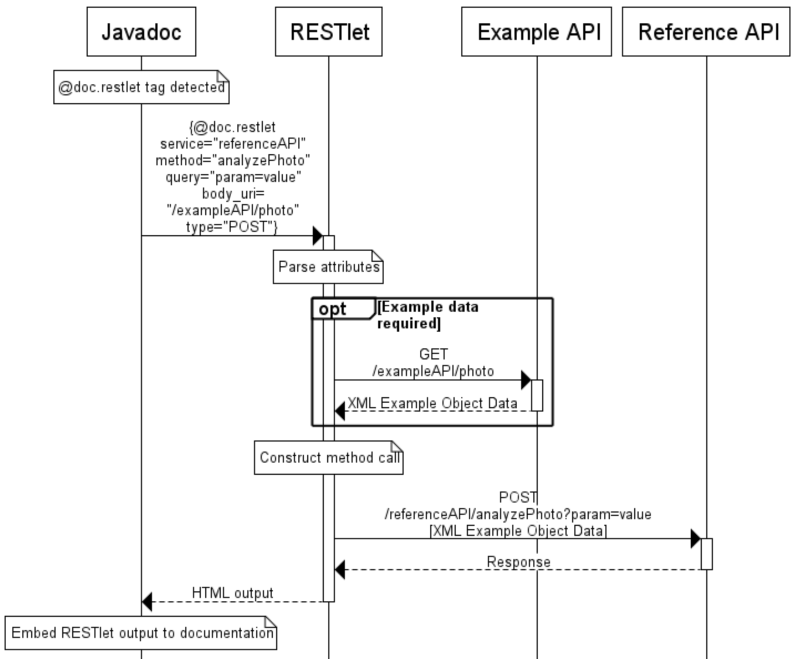

Javadoc
RESTlet
Example API
Reference API
@doc.restlet tag detected
{@doc.restlet service="referenceAPI" method="analyzePhoto" query="param=value" body_uri= "/exampleAPI/photo" type="POST"}
Parse attributes
opt [Example data required]
GET /exampleAPI/photo
XML Example Object Data
Construct method call
POST /referenceAPI/analyzePhoto?param=value [XML Example Object Data]
Response
HTML output
Embed RESTlet output to documentation

Fig. 5. Sequence diagram of Javadoc and RESTlet processing

Any number of other parameters could be defined within the RESTlet tag definition, though adding more would make the commented code files harder to read and could also complicate the use of RESTlet. Implementing the Java doclet itself and the code required to perform the operations described in this section (text parsing, HTTP operations), is a pretty straightforward task, and does not require extensive development time. A minimum implementation of the RESTlet taglet can be found in [4]

Fig. 5 illustrates the sequence of operations when Javadoc parses source code using the RESTlet tag seen in Fig. 4 as an example. When Javadoc encounters the tag *{@doc.restlet ...}* anywhere in the source code, it will instantiate the registered RESTlet class, and the block of text shown in the Fig. 4 is passed on to the RESTlet implementation.

The given text attributes are parsed by RESTlet. If the optional *body_uri* attribute is given, the HTTP Client will call the given URI to retrieve the *Example Object Data*, which could be a simple object as presented in the Fig. 4 or a more complicated payload. In many cases, the calls to RESTful services can be represented simply by utilizing URI query parameters, and in these cases, the "call *Example API"* step can be ignored, and thus, this step is marked as being optional in Fig. 5. The primary purpose of the call to the *Example API* is to facilitate cases where it is difficult or cumbersome to define the query data in the RESTlet attribute definitions or in source code comments in general. The *Example API* can be used, for example, when the content to be sent to the *Reference API* is too large for there to be any sense in adding it inside the source code files or when the content type is binary data, which cannot be added in the source code.

Based on the values of *service*, *method* and *query* the URI to be called is constructed, which in this case would be *referenceAPI/analyzePhoto?param=value*, and the data retrieved from the *Example API* would be used as the HTTP body of the request, which in this case would be an HTTP POST request. As RESTlet will call the API method, it is also possible to utilize this functionality for testing the API while generating documentation. In our use case, the Javadoc extension is primarily meant for example generation and HTML output, and as such it does not validate the API responses retrieved from the *Reference API*. Depending on the service in question it may or may not be a trivial task to validate the responses within the Javadoc extension, and it may be better to rely on more traditional testing frameworks. As discussed earlier, the call could also be directed to the *Service APIs*, if the APIs can produce valid responses to be used within the documentation based on the limited input data that can be defined in the source code and utilized by RESTlet – or provided by the *Example API* in the previous step.

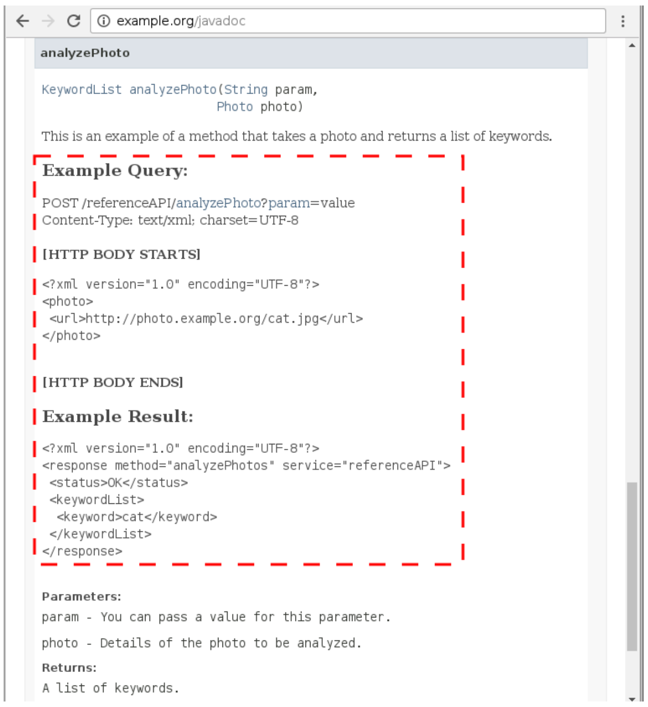
In our case we use a separate *Reference API*, which produces pretty-printed XML documents, and we can contain the responses in HTML <pre> tags to preserve the formatting. As long the methods produce output in a well-structured format, whether it to be XML or JSON, it is a simple operation to modify the output format with RESTlet. Also, because RESTlet has all the parameters used to perform the HTTP request it is a relatively easy task to create a template for printing the example request and the related response for creating a complete method call example. Similarly, only the request or the response could be printed and the remaining parts of the HTTP operation output could be discarded. Finally, the formatted responses are returned to Javadoc by RESTlet and Javadoc will embed the output within the surrounding (automatically) generated documentation.

The RESTlet tag can be located inside any comment normally processed by Javadoc, though at least in our case the most natural place is inside a method comment block accompanied by other textual descriptions related to the method usage. This feature also highlights an important concern in using the RESTlet tag (or any Javadoc tag): the modifications to the method source code do not automatically update the RESTlet tag parameters and the modifications to the tag do not update the method. The disadvantage is that the programmer must manually update the RESTlet tag if changes to the method signature are made. The advantage is that even though the most logical URI to call might be the one associated with the documented method (in the example of Fig. 4, the URI, which causes a call to the *analyzePhoto* method), the tag can also be defined to call *any* URI. The content returned by the call to the URI can be embedded into the documentation as long as the content type can be used within HTML markup. Defining an invalid URI will not be caught when the program is compiled, but it will be caught when the documentation is generated.

Additionally, if constant parameter declarations are available, these can also be used in the tag declaration if so desired. In the example presented in the Fig. 4 and Fig. 5, the parameter values are given as hard-coded plain text. The attribute parser would use these values as they are. However, by utilizing Java's reflection [42], it is also possible to retrieve values for variables declared elsewhere in the source code ignoring access modifiers, or by using the Java class loader to initialize variables based on the plain text found in the comments as long as the text denotes a valid variable path. For example, the method parameter (in Fig. 4) could be replaced with a more complex "method=[org.example.ReferenceService#METHOD_NAME_PHOTO]". This could be used to signal the parser that the value should be resolved from a field called "METHOD_NAME_PHOTO" declared in the Java class "org.example.ReferenceService" – that is, in a variable "public static final String METHOD_NAME_PHOTO" of the class ReferenceService. The package.class#VARIABLE is the standard style utilized, for example, with the default @value tag of Javadoc, and in this case, the inclusion of brackets, [ ], commands the extension to resolve the variable name instead of using the literal value. Utilizing the Java class loader or reflection is not required for the basic operation of the RESTlet extension, but shows an example of an advanced feature.

7. HTML Documentation

Fig. 6 shows an example of the generated documentation in a modern web browser (Google Chrome). The injected auto-generated XML example is highlighted by a dotted box. The rest of the HTML page is created using the standard Javadoc syntax and formatting in the source code, and the page has the look'n feel common to Javadoc documentation generally found on the Internet. An attempt to generate the documentation without the taglets (the RESTlet extension) results in warnings produced by the Javadoc tool in the generation phase and in the absence of examples in the final documentation. In this case, the XML example found inside the dotted box would simply be missing. This allows the documentation to be created in environments, which do not have access to the RESTlet extension.



analyzePhoto

```
KeywordList analyzePhoto(String param,
                         Photo photo)
```

This is an example of a method that takes a photo and returns a list of keywords.

Example Query:

POST /referenceAPI/analyzePhoto?param=value
Content-Type: text/xml; charset=UTF-8

[HTTP BODY STARTS]

```
<?xml version="1.0" encoding="UTF-8"?>
<photo>
 <url>http://photo.example.org/cat.jpg</url>
</photo>
```

[HTTP BODY ENDS]

Example Result:

```
<?xml version="1.0" encoding="UTF-8"?>
<response method="analyzePhotos" service="referenceAPI">
 <status>OK</status>
 <keywordList>
  <keyword>cat</keyword>
 </keywordList>
</response>
```

Parameters:

param - You can pass a value for this parameter.

photo - Details of the photo to be analyzed.

Returns:

A list of keywords.

Fig. 6. HTML documentation example

The documentation can be further customized by utilizing the standard methods available for Javadoc, such as CSS definitions, or by implementing additional taglets. The custom taglets do not generally interfere with each other or with the default Javadoc tags though one should keep in mind that redefining a pre-existing tag name will override the previous tag. Javadoc passes only the parts of the code comments annotated with our RESTlet tag (@doc.restlet) to our extension, and the remaining code is processed by the default Javadoc implementation and is embedded into the documentation unaffected by our additions.

8. Limitations and Implementation

One way of looking at the limitations of the approach is to consider the work required in implementing the supporting APIs, documenting the Service API methods and creating the tasks responsible for the generation of the documentation. As the implementation work is the most significant downside of the presented method, the following subsections discuss the possibilities for implementing the required functionalities, and describe how the requirements have been fulfilled in our service implementation.

8.1. Implementation of the Reference and Example APIs

In our case, the *Example* and *Reference APIs* are implemented as entirely separate APIs (as illustrated in the Fig. 7, below), and they contain mockup copies of all relevant interfaces – of "our" interfaces, and of remote interfaces required for asynchronous communication by our integration specification. The services that implement our remote interfaces are developed simultaneously with our core service (system) by different teams requiring us to run mockup implementations of interfaces, which are not yet fully implemented or that require data that is not always available. Thus, in our case, the *Reference API* is a collection of these mockup APIs, provided for the remote teams for testing purposes – in a way, the mockup interfaces work as a reference implementation of our system, which others (primarily developers, as opposed to the end-user clients, which use the *Service APIs*) can use to test their own applications without having an access to the running "stable" version of the system. Similarly, in addition to our internal testing purposes, the *Example API* can be utilized by the external teams to generate test objects for their benchmarks and test cases. In our case the implementation of the Example API is more complete than would be required for the generation of the documentation alone to allow developers access to pre-made examples of all data structures and formats utilized in our Service API implementation.

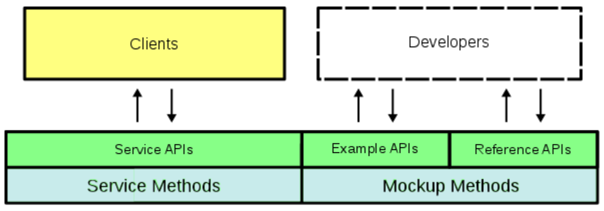


Fig. 7. API model for automatic documentation generation

In some cases it might be possible to use the *Service API* to mimic the *Reference API*. On option is to run a separate development or testing environment with the running system and use the actual *Service APIs* with test data to produce the required mockup responses used for the documentation generation. Unfortunately, the use of actual APIs can create a couple of challenges. When a mockup Reference API is used, it is irrelevant what data is provided in the method calls as long as the method parameters and the payload is structurally valid, but when a testing configuration of the Service API is used as the Reference API, it is possible that the only option for the implementation of the Example API is to hard-code the examples to guarantee that the generated data is usable with the API.

Additionally, the use of a testing configuration may require the use of test user accounts. Depending on the authentication scheme used, it can range from trivial to time-consuming to implement authentication for the RESTlet. In our implementation, the Reference API accepts both unauthorized and authorized users in order to simplify testing. A related issue is the possibility to abuse the APIs for malicious purposes, such as denial-of-service attacks or brute-forcing system access if authentication is implemented, though this is a real risk for any online service. For these reasons, it might be required to implement a proper authentication scheme in any case, or at the very least implement usage limits for the Reference and Example APIs.

The increase in the work required to implement the method presented in this paper, starting from the design of the RESTlet extension to the possible need to implement additional Example API or Reference API is obvious. On the other hand, once implemented, it should guarantee the correctness of the example documentation in the future. If Service APIs are directly used in the generation process, the changes in the APIs are automatically reflected in the generated documentation, if a separate Reference API (mockup) implementation is used, it needs to be kept up-to-date with changes to the Service API. For this reason, the direct use of Service APIs might be preferred to reduce the implementation work. The Example API, in general, does not need to be modified unless new data types or formats are added – that is, the same example data can be used for any number of Reference API calls.

Moreover, it allows the modifications to the examples to be made on the program code level by the programmers themselves without the need to manually update the application documentation (for example, the HTML documentation). If there is no need for either the *Example* or the *Reference API*, then it might not be worth the extra effort to implement them only for the sake of documentation.

In practice there could be services that cannot be easily converted into mockup or test implementations, or cannot be organized in such fashion that in the documentation generation phase there would a working (online) API that produces data usable for the documentation generation. The services, which cannot adhere to aforementioned limitations, are not feasible for use with the model shown in Fig. 7 – or usable with the approach described in this paper in general.

8.2. Documentation of the Methods

There can be excessive work required to re-document the methods, especially in the case of an existing public API, which has already been documented using alternative means. The size of the service also plays an important role in whether the approach described in this paper should be used or not. The size issue is basically twofold. On one hand, if the service only has a handful of methods, manually creating the example codes is doable though perhaps not a pleasant task. One the other hand, a smaller service can be easier to modify to support the methods described in this paper even at a later phase of development. To reduce the complications arising from the example generation process and from automatic documentation in general, the method of documentation should be decided when the system is designed – and not after the system has been implemented.

The design of the RESTlet extension tag is similar to the standard Javadoc convention, which should make it easier for developers accustomed to writing code in Java to start using it in their own applications. Additionally, the RESTlet extension can be modified by the developers to take advantage of alternative tag syntax or to extend the tags to include more advanced features. The ability of the Javadoc tool (and its extensions) to execute any Java code during the processing of the source code files is an often overlooked feature in the design of the application deployment and documentation process.

Once the RESTlet tags have been added to the comments, they need to be modified only if the method signature changes. As the tags are located in the method comments, they can be easily modified when the method is modified by the programmer – as one would modify any method comment when making changes to the method. If the change affects only the HTTP body content required for the request, it is enough to update the Example API, and the changes will automatically be used by the RESTlet. Or, if the Example API uses the Java classes directly for the generation of the example data, the class modifications will be automatically visible in the documentation.

8.3. Running the Automatic Documentation Generation

As mentioned before, our approach requires that at the very least a test configuration of the Service APIs is available during the creation of the documentation. The process of generating the documentation is very similar to any other test case that might be run to validate the overall system functionality.

Unlike in the implementation and documentation of the APIs, which require repeated work throughout the API evolution, the actual process of creating the documentation can be prepared once and run whenever needed. The required test environment can be created using scripts or automated tools, and many of the commonly used build tools include pre-made tasks (such as tasks for Apache Ant [43] and Apache Maven [44]) for utilizing the standard Javadoc tool making it relatively easy to integrate the document generation in the application deployment cycle enabling up-to-date documents to be present at all time. The HTML documentation can be generated using the default Javadoc tool found on JavaSDK releases from Oracle or from OpenJDK. The process does not require any external plug-ins and the required taglets (the RESTlet) can be added using command line arguments. There also exists a possibility to generate PDF documents from Javadoc [45], but whether this is a viable option or not depends largely on the use case.

Additionally, as the generation process in our approach directly uses the responses produced by the API methods, errors in the API implementations are directly reflected in the generated documentation. Thus, it makes sense to include the generation of the documentation as part of the normal testing or deployment process to guarantee the correctness of the documentation.

9. Future Directions

One problem with our approach is that integrated development environments (IDE) do not necessarily support refactoring for custom comment tags, which may cause problems depending on how the method call paths (URIs) and parameter names were defined in the comments. The paths and method signatures of public APIs should not change often, but if they do, it might be difficult to track the changes. The Doclet extension detects invalid paths in the documentation generation phase, but these errors are only caught when the Javadoc tool is executed, and not whilst making changes to the code. One solution would be to implement a custom module for the IDE used, which handles refactoring, if that is possible with developers' preferred IDE choice. Another solution proposed would be to replace the Javadoc tags with Java annotations [28], but this would create an additional code dependency – that is, the annotation classes used for documentation would be needed to compile the application. Additionally, this would require the use of an external application for processing the annotations because the default Java compiler (or the Javadoc tool) cannot be easily modified to run arbitrary code when custom tag is found in the source code. Similarly, other documentation generators could be used (e.g. Doxygen [46]) to process the RESTlet tag. This would enable the use of the method in programming languages and environments, which do not support Doclets. It could be challenging to port the RESTlet implementation to another platform as such, but the core functionality of the RESTlet (creating HTTP requests and returning formatted responses) is simple enough to be doable on other platforms.

One issue mentioned in studies related to code documentation [10][47] and often faced in everyday programming tasks and further validated by the popularity of the "question and answer sites" (such as Stack Overflow [48] and Rosetta Code [49]) and blogs [50] found on the Internet is the difficulty of finding usable documentation. Unfortunately, the approach chosen here does not help in this matter. Even though using Javadoc gives the documentation a more "standard" look and feel, it requires the reader to have some understanding how the Javadoc documentation structure is organized. Also, when it comes to APIs, one still need to have a "gut feeling" on where something is located, and which API should be used to perform what task regardless of how logical the method naming conventions in the API are designed to be.

The use of standard Javadoc commenting style does provide one possible solution for the difficulties related to the use of the generated documentation. There have been research studies on how to format and organize the documentation to be more usable [51], and there are studies on how to process existing API documentation to be more usable [8][52][53]. As our approach uses standard Javadoc, other extensions could be used to further improve documentation generation, though this has not been tested in practice. From a technical point of view, the use of Javadoc tags allows to use the chosen approach with any source code or IDE because the tags do not have dependencies on the code level. Additionally, since the tags are utilized within the code comments, they are normally ignored by compilers removing any compile-time issues.

10. Conclusion

Often the original specification for the service is not in a format that can be easily converted into online documentation. Especially, if both the original version and the online version are needed, the documentation process requires constant maintenance keeping the documentation up-to-date. Additionally, errors in the created documentation can be difficult to detect.

In our case, an initial draft of the service and data format specification is created, but afterwards we do not provide any traditional document-based specification. Instead, the specification is created in-code through the implementation of the Java classes. The use of the Example and Reference API, in combination with class and method comments, allows us to generate the specification (including examples). Of course, the programmer could write the entire documentation in the code comments, including the XML examples, but this can be cumbersome and more time-consuming than to simply type a single tag, which directs the document generator to create the example automatically.

Also, in our experience, regardless of the work required in implementing the Example and Reference APIs, the examples tend to be kept more up-to-date when they are defined directly in the code files as opposed to maintaining a separate online or document-based specification. More importantly, the approach automates perhaps the most tedious part of the process: the creation of the final documentation, which can be fully performed through automated scripts or by utilizing deployment tools.

In our research projects, the Doclet-based implementation has replaced the traditional document-based approach when it comes to interface specifications. Our test case has shown that the approach can be utilized to document web-based service, though additional feedback would be required from a wider range of developer teams (and use cases) to fully discover if application developers are willing to use the RESTlet based approach.

References

1. Digile: Data to Intelligence Program.[Online] Available: http://www.datatointelligence.fi (current October 2016)
2. Rantanen P, Sillberg P, Soini J.: Content Analysis System for Images. In Proceedings of the 16th International Multiconference Information Society (IS 2013), Volume A, Josef Stefan Institut, Ljubljana, Slovenia, 241-244. (2013)
3. Sillberg P, Rantanen P, Soini J.: A Content Based Tool For Searching, Connecting and Combining Digital Information - Case: Smart Photo Service. In Proceedings of the 16th International Multiconference Information Society (IS 2013), Volume A, Josef Stefan Institut, Ljubljana, Slovenia, 249-252. (2013)
4. Javadocer/RESTlet Javadoc documentation. [Online] Available: http://visuallabel.github.io/javadoc/javadocer (current October 2016)
5. Espinha T, Zaidman A, Gross HG.: Web API growing pains: Stories from client developers and their code. In Proceeding of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week, Antwerp, Belgium. (2014)
6. Forward A, Lethbridge TC.: The relevance of software documentation, tools and technologies: a survey. In Proceedings of the 2002 ACM symposium on Document engineering (DocEng '02), Virginia, USA, 26-33. (2002)
7. Lethbridge TC, Singer J, Forward A.: How Software Engineers Use Documentation: The State of the Practice. IEEE Software, Vol. 20, No. 6, 35-39. (2003)
8. Subramanian, S., Inozemtseva, L., Holmes, R.: Live API documentation. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), Hyderabad, India, 643-652. (2014)
9. Stylos J, Faulring A, Yang Z, Myers BA.: Improving API documentation using API usage information. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2009), Corvallis, Oregon, USA. (2009)
10. Robillard M, DeLine R:. A field study of API learning obstacles. Empirical Software Engineering, Vol. 16, No. 6, 703-732. (2011)
11. Robillard M.: What Makes APIs Hard to Learn? Answers from Developers. IEEE Software, Vol. 26, No. 6, 27-34. (2009)

12. Maleshkova, M., Pedrinaci, C., Domingue, J.: Investigating Web APIs on the World Wide Web. In Proceedings of the 2010 Eighth IEEE European Conference on Web Services (ECOWS '10), Ayia Napa, Cyprus, 107-114. (2010)
13. Spinellis D.: Code Documentation. IEEE Software, Vol. 27, No. 4, 18-19. (2010)
14. de Souza SCB, Anquetil N, de Oliveira KM.: A study of the documentation essential to software maintenance. In Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information (SIGDOC '05), Coventry, UK. (2005)
15. Vassallo C., Panichella, S., Di Penta, M., Canfora, G.: CODES: mining sourCe cOde Descriptions from developErs diScussions. In Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014), Hyderabad, India, 106-109. (2014)
16. Treude, C. Robillard, M.P.: Augmenting API Documentation with Insights from Stack Overflow. In Proceedings of the 38th International Conference on Software Engineering (ICSE '16), Austin, Texas, USA, 392-403. (2016)
17. Oracle. Javadoc Tool Home Page. [Online] Available: http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html (current October 2016)
18. Swagger, The World's Most Popular Framework for APIs. [Online] Available: http://swagger.io (current October 2016)
19. wsdoc. Documentation generator for Spring MVC REST services. [Online] Available: https://github.com/versly/wsdoc (current October 2016)
20. RESTdoclet. [Online] Available: http://ig-group.github.io/RESTdoclet (current October 2016)
21. SpringDoclet. [Online] Available: http://scottfrederick.github.io/springdoclet (current October 2016)
22. apiDoc. Inline Documentation for RESTful web APIs. [Online] Available: http://apidocjs.com (current October 2016)
23. Sohan, S.M., Anslow, C., Maurer, F.: SpyREST in Action: An Automated RESTful API Documentation Tool. In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15), Lincoln, Nebraska, USA, 271-276. (2015)
24. Buse RPL, Weimer W.: Synthesizing API usage examples. In Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland. (2012)
25. McBurney, P.W., McMillan, C.: Automatic documentation generation via source code summarization of method context. In Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014), Hyderabad, India, 279-290. (2014)
26. Sridhara, G., Hill, E., Muppaneni, D., Pollock, L, Vijay-Shanker, K.: Towards automatically generating summary comments for Java methods. In Proceedings of the 25th IEEE/ACM international conference on Automated software engineering (ASE '10), Antwerp, Belgium, 43-52. (2010)
27. Moreno, L., Marcus A., Pollock L., Vijay-Shanker, K.: JSummarizer: An Automatic Generator of Natural Language Summaries for Java Classes. In Proceedings of the IEEE 21st International Conference on Program Comprehension (ICPC), San Francisco, California, USA, 230-232. (2013)
28. Nosál, M., Porubän, J.: Reusable software documentation with phrase annotations. Central European Journal of Computer Science, Vol. 4, No. 4, 242-258. (2014)
29. Facecbook. Tools & Support. [Online] Available: https://developers.facebook.com/tools-and-support (current October 2016)
30. Twitter. API Console Tool. [Online] Available: https://dev.twitter.com/rest/tools/console (current October 2016)
31. W3Schools (2016). JavaScript test page. [Online] Available: http://www.w3schools.com/js/tryit.asp?filename=tryjs_myfirst (current October 2016)
32. EasyMock. [Online] Available: http://easymock.org (current October 2016)

33. jMock. [Online] Available: http://www.jmock.org (current October 2016)
34. mockito. [Online] Available: http://mockito.org (current October 2016)
35. Fielding RT, Richard N.: Principled design of the modern Web architecture. ACM Transactions on Internet Technology, Vol. 2, No. 2, 115-150. (2002)
36. Fielding RT.: Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, USA, 2000.
37. Oracle. Java Architecture for XML Binding (JAXB). http://www.oracle.com/technetwork/articles/javase/index-140168.html [17 October 2016]
38. Gson. google/gson: A Java serialization library that can convert Java Objects into JSON and back. [Online] Available: https://github.com/google/gson (current October 2016)
39. Oracle. Doclet Overview. [Online] Available: https://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/doclet/overview.html (current October 2016)
40. Apache Software Foundation. Apache HttpComponents - HttpComponents HttpClient Overview. [Online] Available: https://hc.apache.org/httpcomponents-client-4.5.x (current October 2016)
41. IETF, Internet Engineering Task Force (2014). RFC 7230, Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. [Online] Available: https://tools.ietf.org/html/rfc7230 (current October 2016)
42. Oracle Trail: The Reflection API. [Online] Available: https://docs.oracle.com/javase/tutorial/reflect (current October 2016)
43. Apache Software Foundation. Ant, Javadoc/Javadoc2 Task. [Online] Available: https://ant.apache.org/manual/Tasks/javadoc.html (current October 2016)
44. Apache Software Foundation. [Online] Available: Apache Maven Javadoc Plugin. https://maven.apache.org/plugins/maven-javadoc-plugin (current October 2016)
45. Oracle. Javadoc FAQ. [Online] Available: http://www.oracle.com/technetwork/java/javase/documentation/index-137483.html#print (current October 2016)
46. Doxygen. [Online] Available: http://www.doxygen.org (current October 2016)
47. Nasehi S.M., Maurer F.: Unit tests as API usage examples. In Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM), Timișoara, Romania, 1-10. (2010)
48. stackoverflow. [Online] Available: http://stackoverflow.com (current October 2016)
49. Rosetta Code. [Online] Available: http://rosettacode.org (current October 2016)
50. Parnin C, Treude C.: Measuring API documentation on the web. In Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering (Web2SE '11), Honolulu, Hawaii, USA. (2011)
51. Watson RB.: Development and application of a heuristic to assess trends in API documentation. In Proceedings of the 30th ACM international conference on Design of communication (SIGDOC '12), Seattle, Washington, USA, 295-302. (2012)
52. Stylos J.: Making apis more usable with improved api designs, documentation and tools. Doctoral dissertation, Carnegie Mellon University Pittsburgh, Pennsylvania, USA, 2009.
53. Dekel U., Herbsleb J.D.; Improving API documentation usability with knowledge pushing. In Proceedings of the IEEE 31st International Conference on Software Engineering (ICSE 2009), Vancouver, Canada. (2009)

Petri Rantanen received his degree of Master of Science from Tampere University of Technology (TUT), Finland, in 2009. He is currently working as a researcher in the TUT Pori Department in Pori, Finland. His research interests include API and architecture design, programming languages, and software engineering in general.

Received: October 22, 2016; Accepted: January 27, 2017