university of
groningen

faculty of mathematics
and natural sciences

Automated test client for testing RESTful APIs

Bachelor's thesis

July 2013

Student: Hessel van Apeldoorn (s1881140)

Primary supervisor: Prof. Dr. A.C. Telea

Secondary supervisor: Dr. A. Lazovik

Abstract

To test a RESTful API one must make requests to all resources of an API and check whether the provided output is correct. Verifying the output is done by checking whether the request type was allowed on this resource as well as checking what data the request returned. A universal test client should be able to do this automatically. The only entry point that it should need is the base URL of the API that needs to be tested. In this thesis it is shown to what extent a test client like this can be automated and still be universally suitable for RESTful APIs. Furthermore, a test client has been implemented to support the findings. It has been used on a number of real-world APIs and the outcomes of this test work is discussed in this thesis.

Contents

1 Introduction

Request tests, tests performed by communicating with the Application Programming Interface (API) over HTTP, form an important part of testing RESTful [1] web APIs. A test needs to be made for every single request and request type in a RESTful API. A request is tested by sending the API input parameters and checking if the request returns the expected output. For manually testing web APIs people often use applications such as Groovy [2] and SoapUI [3]. For some automation the Linux command 'curl' [4] can be used.

Since each request requires a test, it would take quite some time to write tests for the entire API. For writing tests for requests it is a lot faster if a test client could recognize what input and output a certain request expects. Combine that with a client that can find its way through all the requests of an API and there would not need to be a test method for every single request. This "finding its way" through the resources of an API is called discovery. This discovery is one of the constraints of a true RESTful API and is called the Hypermedia As The Engine Of Application State (HATEOAS) protocol [5].

The purpose of this project is to make a test client that can determine what input and output is expected for a request and also be able to find its way through all the requests of the API. In this thesis is discussed to what extent this is possible.

First off, in chapter 2 a short explanation is given of the software that is used in this thesis. A short revision of what a RESTful API does and what it is used for follows. The test client tool itself also deserves some explanation. In chapter 3 an elaboration on the data flow of the test client is given. The next chapter, chapter 4, contains the testing results of the test client. Chapter 5 contains the questions and findings from my research. In chapter 6 conclusions are drawn and a recommendation is made in chapter 7 for the future work that can be done on this subject. Chapter 8 contains the acknowledgements. To wrap it up, chapter 9 contains the bibliography and chapter 10 contains the definitions for specific often used words in this thesis.

2 Context

This section provides an overview of the techniques and tools used to answer the questions of this thesis. Furthermore, a list of requirements for the test client is given.

2.1 RESTful APIs

A company often has a database with information about its products and other company related information. Whenever a company wants to make this data available such that developers can incorporate this data in their own applications/websites, an API or iframe [6] will most often be the technique of choice. A developer can, by using an API, format and use this data in any way he or she would want, contrary to using an iframe in which a developer certainly does not have full control of this data. An iframe allows to have some web page data (usually some data from a database is contained in this page) inside a web page located elsewhere. This means a developer cannot modify the way the data is presented as well as how it is modified. This poses no problem if the data is presented exactly as desired, but to gain full control over it a developer thus needs to use an API, if available.

Furthermore, an API can be split into a SOAP (Simple Object Access Protocol) [7] API and a REST (REpresentational State Transfer) API. Both have their own advantages and disadvantages [8]. The list of advantages and disadvantages of REST and SOAP is nearly endless. The, for this thesis relevant, differences are listed in table 1.

Furthermore, REST also seems to be gaining the upper hand the last years [9]. This is especially due

SOAP	**REST**
well-known/old	relatively new
large marketshare	small marketshare, but growing
large amount of frameworks	frameworks are largely still in development
one or more HTTP URIS	standardized URI
no importance to HTTP verbs	importance to HTTP verbs
hard to develop	simple to develop

Table 1: Comparison of REST and SOAP

to the simplicity of REST. REST also has quite strict rules, which make it far easier to write automated tests for than for a SOAP API. It is thus near impossible to make tests that can be used for all SOAP APIs.

A RESTful API can be defined as a type of web server that enables a client, either user-operated or automated, to access resources that model a system's data and functions [10]. The list of constraints for RESTful APIs is very large and this list is also quite different in each book on this subject. There are four constraints however, that are important for making an automated test client and that also contribute to the strictness of a RESTful web API [10].

1. Identification of resources. Each resource should have a unique URI.
2. Manipulation of resources through representations. A browser might see HTML while an automated program sees JSON.
3. Self-descriptive messages. A resource should give information, possibly in the header of a request, about the state, format, size of a resource.

4. The API must be hypertext driven. That is, just by following links a client should be able to explore the REST API. Also known as HATEOAS.

2.2 Test client tool

In order to explore which parts of API testing could be automated and which parts cannot, a test client is implemented. This is purely a conceptual software tool and not a fully functional piece of software. And as such, a small explanation here on how the tool is used is included, instead of posting many pages of code.

2.2.1 Requirements

The following section summarizes the requirements for the test client tool. Firstly, the non-functional requirements of the test client:

1. Insensitive to resource changes (adding/deleting/editing resources)
2. Fit on all RESTful APIs
3. Easy to deploy/use, just one line of code to deploy it for a certain API
4. Cross-platform
5. Language in which the API is written does not matter

Secondly, the functional requirements for the test client are the following:

1. A test, made by the test client, consists of sending a request to the API, receiving the response and checking whether the response is correct
2. Each resource and each operation on a resource is tested once
3. The requested data type from an API is JSON (available in nearly all APIs) to ease parsing
4. Base URL is manually inserted, nearly everything else functions automatically
5. If necessary, some form of authentication can be added for each request
6. Errors in tests are written to a file

There needs to be some assurance that this test client works on most RESTful APIs and that it also adheres to the previously mentioned requirements. It would be undoable to test it on all RESTful APIs there are on the internet. Instead, in order to achieve some kind of assurance, a few APIs are used that adhere to most constraints of a pure RESTful API and also have the convenience of using OAuth as authentication. The test client was built such that it could at least check these APIs. The APIs that are eligible for this are the APIs of Github, Netflix, Paylogic (called the 'PTA', still under development though) and Sun Cloud. Netflix however does unfortunately not take in new developers to use their API as of February 2013. And since implementation of this tool started in March 2013, Netflix' API could not be used. The Sun cloud API unfortunately is not easily publicly accessible and thus this API cannot be used for the test client. The Github API and PTA are easy and free to use.

3 Data flow

The test client is somewhat conceptual, therefore this data flow is also conceptual at some points. The test client has to perform multiple actions. To give an idea of the main actions this test client performs, a high-level data flow diagram is given. It gives an overview of what the test client should do. All possible outcomes from a process are not displayed here. The diagram does neither contain error paths. If one of the blocks in the diagram fails, a test will simply be skipped.

The data flow starts with receiving a base url, e.g. http://example.com. From there on, it should be able to work its way through the whole API. When all resources of the API have been visited the test clients job is done.

3.1 Data flow elaboration

The previous section has given a short overview of the data flow of the test client. Each block in this data flow will be explained in more detail in this section. The input and output are based on the success path of that particular block. Alternative paths that arise when certain blocks fail are not discussed, since this would just result in a fail and a test would be skipped. An error code should just pop up and the currently checked resource should be skipped.

3.1.1 Base URL

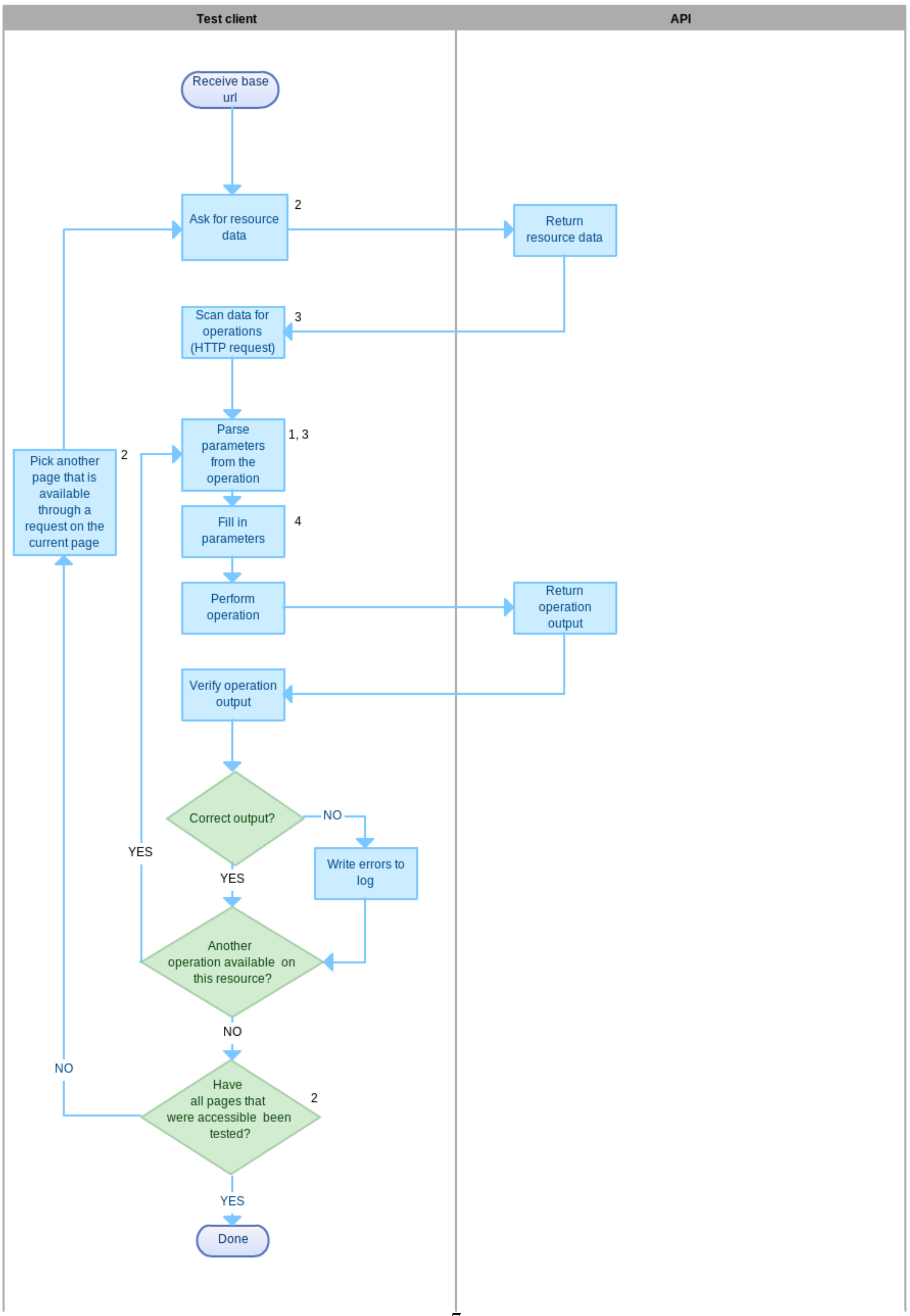
output: url
The first and probably the simplest block. This is also the only block which certainly has to be done manually. The user will have to provide this URL once. The base URL is simply an address, something like http://example.com/rest/. This URL is unique for each API. Hence if the URL is known to the user, the API is known. This URL is also the first resource from which other resources in the same API can be discovered.

3.1.2 Ask for resource data

input: url
output: request
The url that was received in the previous step, is used here to request data. The test client will request a connection with the web API here to receive data. This request concerns one URL, hence the data of one page is requested. This data is requested automatically. The test client will ask for data in the JSON format since this is supported by nearly all APIs.

Return resource data

input: request
output: resource data
This is one of the two steps where the API has to provide information. The API takes a URL and returns the data that is associated with that URL.



Test client
API
Receive base url
Ask for resource data
2
Return resource data
Scan data for operations (HTTP request)
3
Parse parameters from the operation
1, 3
Pick another page that is available through a request on the current page
2
Fill in parameters
4
Perform operation
Return operation output
Verify operation output
Correct output?
NO
Write errors to log
YES
YES
Another operation available on this resource?
NO
NO
Have all pages that were accessible been tested?
2
YES
Done

Figure 1: Data flow of the test client. The numbers refer to the the questions in Section 5.2 as each question refers to some parts of this data flow.

3.1.3 Scan data for operations (HTTP requests)

input: resource data
output: all operations possible in this particular resource
This is one of the more important steps. The data that was received from the previous step has to be parsed here. The resource output will consist of information for several operations which in turn will contain several parameters. The resource output has to be split into separate operations in this stage of the client. This means that the data has to be parsed in such a way that the correct parameters are also attached to the correct operations. To get a bit of an insight in how a single operation is build, an example of how a POST operation is constructed will follow. Different types of operations do vary somewhat in what data they possess. E.g. a DELETE operation usually requires less parameters than a POST. The following should nevertheless give some insight in how an operation is constructed.
First off, the test client constructs the URL for this operation, which is the same for all operations on the same resource. This can be parsed from the previously given data. This will be something along the lines of *www.api.com/{user}/tickets* . The part between braces will be filled in correctly in the next steps. Secondly, the test client determines what POST parameters have to be sent along with this URL. This information can also be retrieved from the resource data. In a POST operation these parameters usually resemble the information a user fills in in a form on a web page. Some parameters would be the name and date of the ticket for example. So the information of this single operation consists of 3 parts; *POST* type, *www.api.com/{user}/tickets* and the parameters *name* and *date*.

3.1.4 Parse parameters from the operation

input: A single operation
output: Operation that has been parsed into URL + parameters
This part of the data flow deals with a single operation. A parser is used to put parameters and the URL into separate variables. Python is quite easy to use with JSON hence the parsing into arguments and URL should be pretty straightforward.

If the parser fails to parse a certain URI then this operation can be marked as failed. There are probably some other resources reachable from this operation however. This means that the test client would first have to fix this parse error before the test client could check these other resources. Ultimately this means that the test client cannot detect all errors in an API in one single run. Making sure that URIs are correct in an API is thus very important.

3.1.5 Fill in parameters

input: parameters
output: filled-in parameters
Perhaps the most versatile block of the data flow. Filling in the parameters correctly is something that is perfectly done by humans. The idea of this test client however is to automate everything as far as possible. Some kind of artificial intelligence has to be applied here. There are different methods for filling in parameters. The ones that have been considered are explained here.

The most primitive way would be to just fill in random strings and random numbers. This tests whether or not a certain resource can be accessed, but does very little besides that. With each testing round there would be random passes and fails (incorrect filled parameters). This method can however be used if all else fails.

A completely reverse approach would be to let all parameters be filled in by a human. But then an important feature of this test client would be lost, namely doing testing automatically. With each new resource the user would have to fill in all parameters again.

An improvement/middle way would be to at least check the type of the parameter before filling it in. But this would for example still allow negative values where only positive values are expected and more of such problems will occur. The test client should therefore be learning. The only thing that certainly distinguishes one parameter from the other is its name. The client therefore keeps a database of variable names together with the expected value or value range of that variable. When the test client discovers a new parameter name it asks for a value or value range which this parameter should have. It will thus have a growing list of values it knows. In this way the test client tests the API without human interference after the values have been added to the database. Adding resources will also be quite easy since the client already knows most of the parameters because of the presence of these parameters in other resources in the API. Later on it is proven that this principle can only be applied to one API and not to, universally, all APIs.

3.1.6 Perform operation

input: operation
output: request
In this step the client commits the previously prepared operation. It sends a request to the API to respond to the given URL plus the generated parameters.

3.1.7 Return operation output

input:request
output:operation data
The API processes the incoming operation and returns the corresponding data output.

3.1.8 Verify operation output

input:operation data
output: Pass or fail
The test client checks whether the output from the API is the output the client expects. When the output is correct, it just continues to the next one. If it is wrong, a notification is made about the wrong output. The outputted data is verified with the resource data from a few steps ago. This resource data does actually contain some information about what the outputted data should look like, at least in a proper RESTful API. Hence when the outputted data and resource data match, the test passes.

3.1.9 Cycling through the data flow

There are a few loops in this data flow. The section from “Parse parameters from the operation” till ”verify operation output“ is repeated for every single operation. Also, the entire part of ”ask for resource data“ till ”verify operation output” is repeated for every resource. These cycles ensure that every part of the API is tested properly and completely.

4 Results

To give some kind of assurance that the test client works, it is used on some APIs. Some APIs did a lot better than others. table 2 shows which parts of an API could and which parts could not be tested.

Table 2 shows that there is no single API that works perfectly. The Github API and the PTA were

APIs	**Reachable resources**	**Test the base URL**	**Discover entire API**	**Perform/ Discover all kind of operations (POST /PUT /GET /DELETE)**
Flickr	✗	✗	✗	✗
Imgur	✔	✔	✗	✗
Rdio	✔	✔	✗	✗
Twitter	✔	✔	✗	✗
Github	✔	✔	✔	✗
PTA(Paylogic)	✔	✔	✔	✗

Table 2: Results of testing APIs with the test client

the only two that yielded some decent results. Other APIs would kill the test client after only testing the base URL. In order to make sure that a full test run could be done on the Github API and PTA, some small modifications had to be made to the test client when one of these APIs were tested. For the Github API the assumption needed to be made that only GET and OPTIONS are valid operations (other operation methods were simply not available or specified). For the PTA, some information about parameters had to be inserted manually.

4.1 API shortcomings

Nearly all (popular) APIs do not fully adhere to the previously mentioned constraints for RESTful APIs. A REST API needs to adhere to these constraints to be testable by an automated test client. A good way of grading the “RESTfulness“ of a RESTful API is to use the Richardson Maturity Model [11]. This model grades APIs from level 0 to level 3. Table 3 gives an overview of how “RESTful” some of the currently most popular APIs really are. The difference with table 2 is that table 2 concerns to what extent the test client could test an API whereas table 3 gives an overview how ’RESTful’ an API is according to the Richardson Maturity Model.

- **level 0:** This level means that this particular API is utter garbage in terms of REST. Resources cannot be distinguished and only one type of request (usually POST). XML Remote Procedure Calls and SOAP APIs belong to this level.
- **level 1:** In this level each URI leads to one and only one resource of an API. There is a distinction between, for example, book 1 (api.com/book/1) and book 2 (api.com/book/2). There is still just 1 type of request however. I have yet to come across an API which does implement resources, but does not implement different kinds of operations.

Level 0	**level 1**	**level 2**	**level 3**
SOAP APIs		Twitter	Github
XML RPC's		Imgur	PTA (Paylogic)
Flickr		Amazon S3	Netflix
		Facebook	
		Twilio	

Table 3: API division according to Richardson Maturity Model

- **level 2:** Most of the mature APIs can be found in this level. All rules from previous levels apply plus the fact that different types of requests will be used. That is, a DELETE for deleting a resource, GET for requesting a resource, PUT for adding a resource and so on.
- **level 3:** This level is the one nearly no API reaches. It states that a client should be able to discover all the resources in an API on its own. This means that the API should provide information to the client as to where a client can set its next step in the API. This property is also known as HATEOAS.

Most popular APIs nowadays do implement separation of resources and provide support for different request types. Hence the large number of APIs in level 2. The fact that an API is under level 3 in the Richardson Maturity Model makes it unsuitable for automatic testing. A client in an API with a maturity level lower than 3 can not find its way in resources. One remedy to this would be to statically tell the client what resources are available. Adding, removing or editing a resource would still break testing then. Which means after each edit of the API, the test client will break, losing the whole advantage of automated testing. This is also a problem for non automated clients, no discovery still means that editing/removing a resource will require all clients to rebuild their systems.

A problem that is also recurring in many APIs is that an API does not tell what to do on a certain page. Most APIs have huge documentation pages (at a different URL than base API URL). When a client asks the API what to do (options call), the API most often responds with either an error or a huge chunk of documentation. This surely makes it somewhat readable for a human, but an automated program cannot do anything with it.

The reason that so many APIs do not meet all standards of a truly RESTful API is not that API developers do not know how to build an API. Rather, it is the easy way out to just document everything instead of providing proper discovery and options in an API. Another problem is that developers often use a file extension in the URL. If the client wants an API to return JSON, then something like this has to be called: *api.example.com/user.json* . This is again the easy way. The proper way would be to include what type of response the user expects in the header.

Even APIs that made it to level 3 still have some flaws. The Github API for example does not implement an OPTIONS request to see what can be done at a particular URL. There is no way in checking what requests (POST,PUT,GET,DELETE) can be made. Instead, the user has to resort to reading the documentation once again. While this is useful for people who use the API and just have to check the documentation once, it is a disaster for automated clients since they simply cannot read this documentation.

4.2 test client shortcomings

Mainly due to the fact that APIs show some shortcomings, the test client unfortunately also shows some practical shortcomings (theoretical shortcomings will be discussed in the next section since the theoretical shortcomings of the test client is actually the research scope of this thesis).

The most important shortcoming is that the test client could not fully verify that all type of operations work correctly in an API. This is especially the case with POST calls. POST calls nearly always require quite some parameters. Ideally, these parameters and accompanying values are documented well in the OPTIONS call. Looking at table 2, it can be seen however that this is not the case with any API that the test client tested.

The API will test any URL that has the base URL as prefix. This means it might also test something that is not a resource, e.g. an image that is stored somewhere in the API. A good practice would be to only have resources start with the base URL. All other media should be stored at a separate URL.

5 Discussion

The following section contains the questions posed regarding automated testing. It also contains the conclusions that can be drawn from the test client tool. Most points have turned out as expected when I started working on this thesis. A few points have turned out differently than expected. Especially the filling of parameters turned out to be different.

5.1 Research Question Revisited

The main goal of this project is to make an automated test client that can test RESTful web APIs. The main question would then logically be: ”how can automated testing be achieved?”. However, the very first thing to ask is: Is this feasible? A 100% automated test client means that the client will have to make decisions that only humans can make. For example, a human decides that a parameter called “salary” has to contain a positive number, likely somewhere between 100 and 100.000 . A test client on its turn could pick a negative number for the field “salary” which would be plain wrong. Making a test client 100% automated is therefore deemed to be not possible. In this thesis the most important questions are not how to automate the client, but rather if it is possible to automate. The main question to be solved is thus: What parts of a test client can be automated?

This is a quite broad question. Before this question can be answered, a few subquestions should first be answered. A test client performs multiple actions to test a request. The main question can therefore be split into subquestions. Each subquestion concerns a different part of the test client.

These subquestions can again be split into even smaller subquestions. This process continues until these questions concern such a small area that a direct answer or explanation can be given. This thesis has thus a bottom-up approach; After all subquestions have been answered, the umbrella question can be answered.

5.2 Findings from testing tool

There are a number of sub areas which need to be answered to answer the broader question “Can a test client be fully automated?”. The main question can therefore be divided into the following sub questions.

- Can the parsing of parameters for a request be automated?
- Can a test client work its way through the entire API on its own?
- Is it possible to see what type of request the client is dealing with?

- Can a client automatically fill in correct values for parameters?
- Can this client be implemented in such a way that it is easy to use in different web APIs?
- How should the client be fit into a complete software project?

In each of the following sections each one of these subquestions is discussed.

5.2.1 Automating the parsing of parameters for a request

Making an OPTIONS call to a certain resource in an API gives back a JSON object. This JSON object contains some kind of description of the resource and also contains the fields (id, name, user, etcetera) that need to be filled in to create an operation on this particular resource. Since this is all JSON, it is quite easy to parse. Most modern programming languages can just use a single library to parse this, including Python, in which the test client was written. The response that is received from the API after an options call can be loaded into an object with just one single line of code. For convenience, these values are stored in a structure like a hashmap such that keys and values can easily be retrieved. Later on these keys are used to fill in their respective values. These values need to be included in an operation. This automatically filling of parameters is discussed a few sections later.

5.2.2 Work your way through the entire API on your own

HATEOAS. It is the only thing that an API needs to implement for client discovery. A client can find its way through the entire API easily if at each resource is stated what resources can be entered next. The only problem is, as stated before, nearly no API implements this properly. The Github API and PTA were specifically chosen since they do implement this feature.

Each resource starts with the base URL. For example, *api.example.com/baskets* and *api.example.com/tickets/1*. To check what resources can be entered next, the test client can simply scan the resource for strings starting with the base URL. Each of these strings provides an entrypoint for the next resource. This string may however include some parts that need to be filled in. For example, *api.example.com/baskets/{user}* . This part needs to be parsed out of the string first and given a correct value later.

"Working your way through" a resource can be seen as walking through a graph. And as such the test client has to be sure that every node is visited once. As every resource is visited that connects to the current URL, with certainty can be said that every resource is visited at least once. But the test client does need to look out for cycles which will cause it to be stuck in an infinite loop. A list is maintained with all URLs that have previously been visited. As such, every URL is surely visited once and only once. An API may however contain some URL that has the base URL as a prefix but that URL might not be a resource. In this case the test client will still view this as a resource and the test will just report a fail.

When testing an API, the tester may not have full control over everything that is present in the backing database of the API. There can be thousands of instances of just 1 resource. The Github API (figure 2) for example has a resource called "user" which includes all users that are using Github. Github has a grand total of 2.8 million users [12]. As such, testing may take an extensive amount of time. All entries of the entire Github API could be tested, but after 7 hours of running the test client it seems wise to abort running. One trick that was applied to severely reduce testing time on these big APIs, is to only test 1 instance/URL per resource. That is, When the test client tests *api.example.com/tickets/3* , it does not test *api.example.com/tickets/2* or *api.example.com/tickets/5* . This reduces testing to under half an hour, which is not ideal, but acceptable for such large APIs. The best option would still be that the tester has full control over the API and can limit the amount of objects that are present in the database.

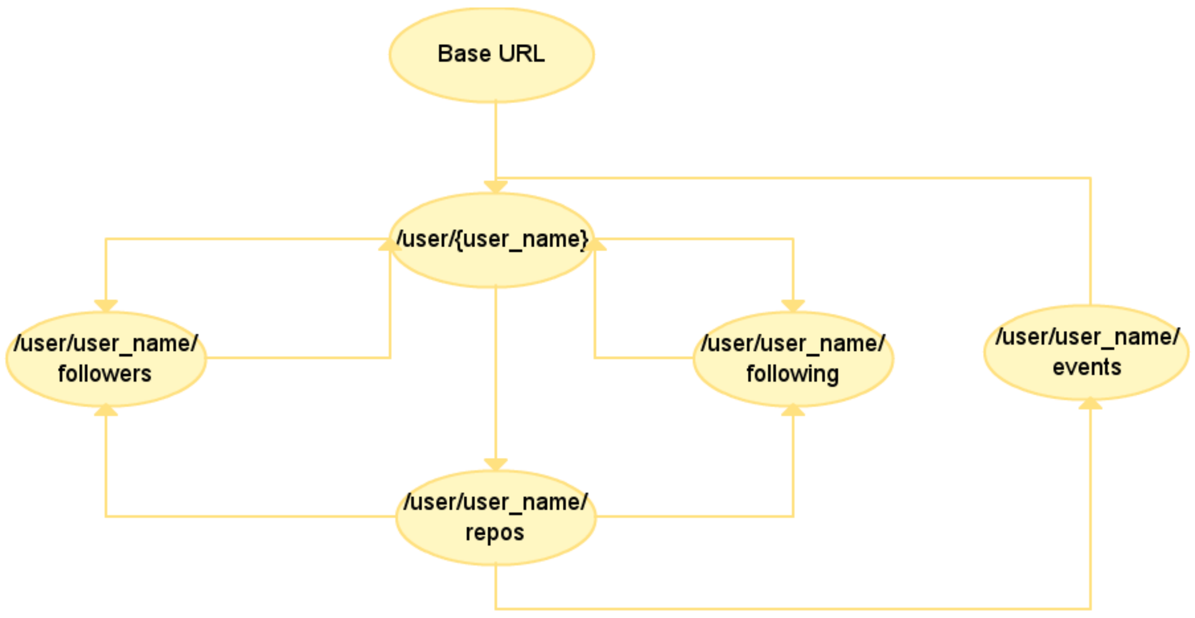


Figure 2: A small part of the resources of the Github API

5.2.3 Detect what type of request the client is dealing with

It appeared to be not feasible to do a lot of detection, rather the test client just builds up a list with keywords manually. As demonstrated by the previous section, this area does not deserve much attention. This is one of the few points that turned out differently than expected at forehand.

5.2.4 Automatically filling in correct values in parameters of a request

When the test client is given an URL, as specified in previous section, it may still contain parts that need to be filled in. In the chapter about the data flow, 3 options were mentioned: Random filling, filling by looking into a dictionary and manually filling. Random filling was already not seen as a good option. Testing is about reliability, randomness cannot be used here.

Filling in keywords by looking into a dictionary appears to be the way to go. A side note is however that APIs differ too much from each other to actually maintain one universal dictionary. What one API may consider to be a valid API is complete garbage to another API. As with random filling, the outcomes is too random.

For one API however, a list of keywords can be kept. Keywords are often reused within one API, not to mention that the developer only needs to specify them the first time the test client encounters them. After that the dictionary knows what values to fill in for certain keywords and a test session runs without any help from a human. This is a combation of the aspects of dictionary filling and manual filling.

5.2.5 Generalizing the test client to be easy to use by different web APIs

A test client can be generalized to some extent. If an API adheres to certain constraints, then the parts that are affected by these constraints can usually be automated. Authentication is one thing that is nearly the same in every good (level 3 Richardson Maturity Model) RESTful API. Nearly every API uses OAuth and as such, the tokens in OAuth are stored in the test client such that all subsequent requests are authenticated automatically. The test client is quite easy to keep general. The test client does not know anything about an API in advance and is still able to use it. And as such, it does not really matter what kind of RESTful API is passed to the test client. It is all about the RESTful API itself obeying to the rules.

5.2.6 Fitting the test client into a complete software project

This turns out to be quite easy. This tool just accepts a base URL and some authentication and it can then find its way around the API.

6 Conclusion

At this very moment it has proven to be impossible to build a universal test client. This came to the surface during the making of the test client and the search for proper APIs. APIs are simply not constructed uniformly enough. There appears to be no RESTful API that fits all the constraints currently. This is also due to the fact that there are no frameworks to build RESTful APIs properly. REST is a technique that big companies are now slowly beginning to adopt [9], which is also one of the reasons that the current APIs are not completely flawless.

The test client has furthermore shown the impracticality of keeping a universal dictionary for all APIs. Which I did expect at first hand when I started with my thesis. Soon I found out that APIs just differ too much. To make a test client universal, developers need to make the APIs more generic. A few other minor points turned out differently than expected or were less important: 'detecting what type of request the client is dealing with' and 'fitting the test client into a complete software project'.

A note on the test client is however that I could not validate that everything in a proper API is covered by the test client. Simply because I haven't come across such proper APIs. When a developer does construct a good REST API with the REST rules in mind, then this client should be able to test it well.

7 Future work

When automating a test client, a few restrictions have to be dealt with. First off, it has proven to be undoable to keep one universal dictionary for all APIs. This means the developer will have to manually input keywords for each API. Since this problem is fundamental it is not something that can be solved by future work. Secondly, making random guesses, will yield random results and is therefore deemed not fit for testing. Not even if the test client can do some recognition such as the typing of a keyword. There needs to be 100% certainty that the value that is used for the keyword, is correct.

Another restriction is that the current APIs that are used often have quite some shortcomings. This is something that can be fixed. In the future, APIs should be build strictly according to the constraints [10] .

There are quite some frameworks that aid in building a RESTful API. One thing that greatly aids in making good APIs, is using a REST framework. This way not every API maker uses his own implementation of, for example, the OPTIONS call and discovery of resource. For Java, developers can use RESTlet [13]. For Python/Django the Django REST framework [14] as well as Tastypie [15] do the job. In the Tastypie framework developers are actually already working on providing a framework to fix some of the common shortcomings of APIs. An OPTIONS request is being implemented which returns what can be done at a certain URI and what input/parameters the API expects from the client. Developers may manage to improve these frameworks such that they produce truly RESTful APIs. This means a developer could just add an automated tester to his framework such that he would never have to look at HTTP testing his API ever again.

8 Acknowledgement

I would like to thank my supervisor Alexandru Telea for helping me tackle the issues of building an automated test client as well as for his extensive feedback. It really helped me to analyse the key problems and their solutions.

I'd also like to thank Paylogic for letting me use their API to build a good test client and helping me out with the implementation of the test client. Special thanks go to my external supervisor Òscar Vilaplana for proofreading some of my early analysis and diagrams as well as helping me to get a better understanding of RESTful APIs.

9 Bibliography

References

[1] Roy T. Fielding. Rest. `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`.

[2] Groovy. `groovy.codehaus.org`.

[3] Soapui. `http://www.soapui.org/`.

[4] curl. `http://curl.haxx.se/`.

[5] Joshua Thijssen. Hateoas. `http://restcookbook.com/Basics/hateoas/`.

[6] iframe. `http://www.w3schools.com/tags/tag_iframe.asp`.

[7] Soap. `http://www.w3schools.com/soap/`.

[8] In *Journal of Information Engineering and Applications*, volume 2, page 14, 2012.

[9] Adam DuVander. New job requirement: Experience building restful apis. `http://blog.programmableweb.com/2010/06/09/new-job-requirement-experience-building-restful-apis/`, June 2010.

[10] Mark Masse. Designing consistent restful web service interfaces. In *REST API Designer Rulebook*, page 14, 2011.

[11] Martin Fowler. Richardson maturity model. `http://martinfowler.com/articles/richardsonMaturityModel.html/`, March 2010.

[12] Github. `https://github.com/blog/1359-the-octoverse-in-2012`.

[13] Restlet framework. `http://www.restlet.org`.

[14] Django rest framework. `http://django-rest-framework.org/`.

[15] Tastypie. `http://django-tastypie.readthedocs.org/en/latest/`.

10 Definitions

- **client:** User of an API.
- **automated test client:** A client that tests whether all actions on all resources return the values they should return. It does not know anything about the particular API in advance. It is just given a user account to use the API and a base URL as a starting point to explore the URI.
- **request:** A request is an URI with a calling type (GET/POST/PUT/DELETE).
- **GET/POST/PUT/DELETE:** The most common types of requests in an API. Each one is used for a different type of action on a resource.
- **resource:** This is all the information that can be found on a certain url, e.g. on *http://example.com/events* all the information about the resource “event“ can be found. The data in this particular resource can be scanned to find out what ’operations’ can be performed from this point.
- **parameters:** The information that a request needs to know to display the proper output. For example, for adding a product to the paying basket an API would possibly need the parameter ’Name’ to know what type of product the user has bought.
- **operation:** An operation can also be described as a HTTP request. It is very similar to the previously defined ’request’. An operation is performed by sending a (GET/POST/PUT/etc.) request to the API along with correctly filled in parameters of the request.
- **keyword:** Part of the URI that has to be filled in by the client and is not predetermined.
- **OAuth:** common way of authenticating requests in a RESTful API.