Introspected REST: An alternative to REST and GraphQL?

Florian Gerlinghoff

16th February 2021

The following pages summarize my bachelor's thesis, which I have written in German. The complete translated title is *A comparison of Introspected REST with alternative approaches to the development of web APIs with regards to performance, evolvability, and complexity*. Please find the complete thesis attached.

1. Introduction

In 2017, Filipos Vasilakis introduced a new way of developing web APIs: *Introspected REST*. This came after the world had seen two other API styles arise very recently. Two years earlier, Facebook announced GraphQL and Google made gRPC available to the public. But in contrast two both of these, Introspected REST did not attract a lot of attention and still only leads a niche existence. Yet the question arises whether the industry misses out on something in ignoring Introspected REST. Could it be that Introspected REST really is an alternative to REST and GraphQL, as suggested by Vasilakis, or even a better way of doing things?

To get to the bottom of this question, Introspected REST needs to be tested and compared against the other approaches used by the web API community today. The goal of my thesis was to present a comparison of Introspected REST with REST, GraphQL, and gRPC regarding three comparison criteria: performance, evolvability, and usability. In this summary, I will focus primarily on whether Introspected REST can do better than REST.

In section 2, the problems of REST as identified by Vasilakis are described. These problems motivated the creation of a new API style. After that, the two pillars of Introspected REST that help to avoid those problems, introspection and microtypes, are introduced. For each of the three comparison criteria, section 3 contains an outline of the method and presents the results of the comparison. Finally, in section 4, the results are shortly summarized and discussed.

2. Introspected REST

In [1, sec. 8.2], Vasilakis points out the problems he sees with REST, based on his own experience and without providing empirical evidence.

First, in a usual REST API, hypermedia information and other metadata are sent along with the plain data. Because the server does not know what information the client is actually interested in, it has to send all the metadata available, which can lead

to huge response messages. Clients who do not make use of some or any of the metadata still have to handle the higher complexity that comes with these additional information. Moreover, it deteriorates the performance of the communication, because possibly unnecessary data is sent to the client and the (static) metadata cannot be cached independently of the plain data, although the former tend to change less frequently than the latter [1, secs. 8.2.1-8.2.4].

Second, REST APIs use media types, which are mostly big monolithic structures with many responsibilities. In order to conform to the specification, clients and servers have to take an all-or-nothing approach when implementing a media type. Media types are also not composable, which, for example, makes it hard to switch out one part while keeping the other parts. When using REST, this would require a completely new media type specification [1, sec. 8.2.7.2].

Third, REST makes it difficult to incrementally evolve an API that does not yet provide hypermedia elements to become a 'Level 3' REST API [2]. Because plain data and metadata are intertwined in a single response, such a change could break existing clients. And since media types lack composability, individual parts of a media type cannot evolve without needing to change the whole media type [1, secs. 8.2.5, 8.2.6.1; 3].

With Introspected REST, Vasilakis wanted to create an API style that solves these problems while maintaining the benefits of REST [1, ch. 2] – especially evolvability, which makes REST a good choice for public APIs that are supposed to be around for a long time [4, p. 41]. Introspected REST carries over most of the six REST constraints unchanged, but replaces *Hypermedia as the engine of application state* with *Introspection as the engine of application state*. Furthermore, Introspected REST encourages the use of *microtypes*.

2.1. The Introspection Principle

The introspection principle states that a client should be able to examine the capabilities and properties of an API and its resources at runtime [1, sec. 9.3]. That is, in Introspected REST, clients can at any given moment request information like the schema of the expected response, next available actions, or the functionality the API provides for a specific resource. Similar to REST APIs [5], a client of an Introspected REST API only needs to know the initial URL and support the media- and microtypes used by the API.

An important characteristic of the introspection process is that it separates data from metadata [1, sec. 9.3.2]. Clients will only receive plain data with one request and have to issue another one in order to receive additional metadata.

What distinguishes the introspection principle from REST's hypermedia principle is, besides this separation, that clients can customize the introspection responses they receive through microtypes [1, sec. 9.3.1]. Microtypes make it possible for a client to precisely indicate which metadata it wants to receive.

2.2. Microtypes

The idea of microtypes is to break down media types into small, independent, composable units of functionality [1, sec. 9.2]. For example, one microtype could deal with pagination, another one with semantic information about the resources via RDF, and a third one with semantic information via JSON-LD.

The main advantage that microtypes offer to API clients is a more fine-grained content negotiation. Clients can choose exactly what data they want to receive from the server, therefore reducing the amount of unnecessary data sent. Furthermore, microtypes let choose clients from every possible combination of microtypes a server provides, satisfying the needs of lots of different clients [1, sec. 9.2.1.1]. An additional advantage is that microtypes enable the reuse of functionality across APIs [1, sec. 9.2.1.2] and could turn out to be a good platform for an ecosystem of patterns and building blocks of APIs.

3. Comparison

As described in section 1, the goal my thesis was to compare Introspected REST with REST, GraphQL, and gRPC. Therefore, comparison criteria had to be defined. Vasilakis writes about REST:

> It's inflexible, difficult to implement, difficult to test, with performance and implementation issues. But most importantly, any implementation of REST model is very complex. [1, sec. 2]

Introspected REST tries to solve the problems of REST like the impact on performance and complexity, while maintaining the benefits, especially the good evolvability. Therefore, for this comparison, the performance, evolvability, and complexity/usability of the four API styles were analysed, assuming a public API. One crucial property of public APIs is that they are used by a variety of clients [6, S. 7] that are out of the provider's control. According to Mitra, the primary goal of a public API is to increase its market share, that is, to convince a lot of client developers to use the API [7, 06:47–07:03]. Thus, the provider has to make the API great from the perspective of a client developer. For this reason, all comparison criteria were observed from this perspective.

3.1. Performance

The most important performance metric for a client developer is the *response time*, which is the time from when the first byte of the request is sent to when the last byte of the response is received. To measure the response time for the API styles under examination, a scenario-based experiment was conducted. Every API client was instructed to execute five tasks using an API for digital note-taking. The five tasks are described in appendix A. They typify interaction patterns between client and server that are commonly found in APIs, like a single request-response, fetching all items of a list (*N+1* queries), or multi-deletion. For every task, the time the client needed to finish it was measured using *Apache JMeter*[1].

Independent variables in this experiment were the latency, network throughput, and the performance of the client and server themselves. The experiment was carried out in a local environment and data transmission happened via the loopback network interface. This made it possible to control bandwidth and latency. The bandwidth was artificially limited to 34KB/s upload and 58KB/s download in a first, and 100KB/s up- and download in a second execution, while the latency was virtually zero. The downside of this setup was that client and server had to share hardware resources.

The API client was constructed using samplers of Apache JMeter. The servers

[1] Apache JMeter: https://jmeter.apache.org/ (visited 14/02/2021)

were implemented based on ASP.NET Core, using third-party libraries for the REST, GraphQL, and gRPC APIs and a library for Introspected REST that was developed as part of the thesis. The latter was inspired by suggestions from Vasilakis [1, ch. 10]. The Introspected REST client requested only links and no additional metadata.

The figure in appendix B plots the average response times per task for every API style relative to the response time for the Introspected REST API. GraphQL shows the best overall performance, followed by gRPC. The reason is, as I discuss in my thesis, that GraphQL sends the least data between client and server, while gRPC offers a performance-optimized protocol. The differences between Introspected REST and REST cannot necessarily be attributed to the API styles themselves, but are more a consequence of the library used to implement the REST API. I concluded that Introspected REST does not offer huge performance benefits compared to REST in this experiment. This result, however, should not be interpreted as if Introspected REST does not offer performance benefits at all. Especially when clients do not request additional metadata, using Introspected REST could lead to a significant performance improvement. But in this experiment, the clients for REST and Introspected REST both requested the same kind of data. Other limitations of the experiment include that requests were not sent in parallel and that all but the gRPC client used version 1.1 of the HTTP protocol.

3.2. Evolvability

Evolvability describes the ability of a system to adapt to a changing environment or changed requirements [8]. When talking about APIs, evolvability refers to the degree to which an API can change without impacting existing clients [9, p. 34]. Changes to the API can be classified as either breaking or non-breaking. Breaking changes force a modification of the client in order to be able to successfully communicate with the server again [10, p. 215].

To compare the evolvability of the four API styles, changes of the note-taking APIs mentioned in section 3.1 were simulated. Then, for every API style, a change was either classified as breaking or non-breaking.

A description of the changes made to the API as well as the results can be found in appendix C and appendix D, respectively. Introspected REST and REST both can be used to design APIs which are more evolvable than GraphQL or gRPC APIs. Especially through the use of hypermedia elements can clients incorporate changes, even changes in the business logic, without modification [11, 37:35-38:47]. Introspected REST achieves its goal of keeping REST's good evolvability and can even improve slightly on it through microtypes.

3.3. Usability

To compare the usability of the API styles, a heuristic evaluation [12] was performed. This usability inspection method was used despite problems, like incompleteness or lack of validity [13; 14], because of the time-limited nature of this thesis. The heuristics were compiled from common usability principles as well as studies of the usability of class library APIs. They are described in detail in my thesis, section 5.3.

appendix E summarizes the results of the evaluation, using a scale from – (API style does not comply with heuristic at all) to +++ (API style fully complies). Every API style shows individual strengths and weaknesses, obstructing any attempt to even order the

API styles partially by their usability. The precise specification of GraphQL makes all GraphQL APIs look familiar. Furthermore, they are very predictable because client developers only get back the data they selected. Introspected REST offers a similar advantage through microtypes. When designed with this goal in mind, microtypes can make it possible for clients to specify exactly in which data they are interested. Other advantages of microtypes are that they make it easier to use common patterns in APIs, therefore increasing the consistency within and between APIs, and that their modularity enables code reuse for client developers.

4. Summary

In this thesis, a first comparison of Introspected REST with the alternative API styles REST, GraphQL, and gRPC was presented. Introspected REST builds upon REST, but adds the introspection principle and microtypes to it. Thus, of special interest was whether Introspected REST can match or even surpass the qualities of REST. In the experiments and evaluations conducted, Introspection REST shows similar performance and evolvability to REST, but a better usability thanks to microtypes.

This thesis did not include an examination of an Introspected REST client that does not use additional metadata. Such a client is expected to show better performance and usability at the expense of evolvability.

Introspected REST seems to steer a pragmatic middle course between low complexity and high evolvability. Pragmatic, because in the real world, a lot of REST clients do not make any use of hypermedia, but are instead highly coupled to the server. Through the separation of plain data and metadata can these clients benefit from a better performance and lower complexity. Microtypes make it possible to satisfy the needs of many different clients, making it suitable for APIs that nowadays use GraphQL.

References

[1] F. Vasilakis, 'Introspected REST: An alternative to REST and GraphQL,' [Online], 10th Sep. 2017. URL: https://introspected.rest/ (visited on 08/02/2021).

[2] L. Richardson, 'Justice will take us millions of intricate moves,' in *QCon San Francisco 2008*, [Conference Talk], 20th Nov. 2008.

[3] F. Vasilakis. (23rd Nov. 2017). 'Microtypes: Let's break down the monoliths of content negotiation.' [Online], URL: http://web.archive.org/web/20201001115958/http://microtypes.info/ (visited on 15/02/2021).

[4] G. Block, P. Cibraro, P. Felix, H. Dierking and D. Miller, *Designing Evolvable Web APIs with ASP.NET*. Sebastopol, CA: O'Reilly, 2014.

[5] R. T. Fielding. (20th Oct. 2008). 'REST APIs must be hypertext-driven.' [Online], URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (visited on 12/02/2021).

[6] D. Jacobson, G. Brail and D. Woods, *APIs: A Strategy Guide*. Sebastopol, CA: O'Reilly, 2011.

[7] R. Mitra, 'Developers are people too! building a dx based api strategy,' in *Nordic APIs*, [Conference Talk], Stockholm, 18th Sep. 2013.

[8] H. P. Breivold, I. Crnkovic and P. Eriksson, 'Evaluating software evolvability,' in *Proceedings of the 7th Conference on Software Engineering Research and Practice in Sweden*, T. Arts, Ed., Göteborg, Sweden, Feb. 2007, pp. 96–103.

[9] R. T. Fielding and R. N. Taylor, 'Architectural styles and the design of network-based software architectures,' Ph.D. dissertation, University of California, Irvine, Irvine, 2000.

[10] A. Lauret, *The Design of Web APIs*. Shelter Island, NY: Manning, 2019.

[11] O. Drotbohm, 'Rest beyond the obvious,' in *GOTO Amsterdam 2019*, [Conference Talk], Amsterdam: GOTO Conferences, 19th Jun. 2019. URL: https://gotoams.nl/2019/sessions/802/rest-beyond-the-obvious-api-design-for-ever-evolving-systems.

[12] J. Nielsen and R. Molich, 'Heuristic evaluation of user interfaces,' in *Proceedings of the SIGCHI conference on Human factors in computing systems Empowering people - CHI '90*, ACM Press, 1990.

[13] E. de Kock, J. van Biljon and M. Pretorius, 'Usability evaluation methods: Mind the gaps,' in *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on - SAICSIT '09*, ACM Press, 2009.

[14] E. L.-C. Law and E. T. Hvannberg, 'Analysis of strategies for improving and estimating the effectiveness of heuristic evaluation,' in *Proceedings of the third Nordic conference on Human-computer interaction - NordiCHI '04*, ACM Press, 2004.

A. Tasks for performance evaluation

T1 For every note, query its id and title as well as the user name of its creator.

T2 Query the title, content, and date of creation of the first note from **T1**.

T3 Query the keywords of all notes that were edited by an author who also edited the first note from **T1**.

T4 Add the new keyword ‘Neu’ to the first note from **T1**.

T5 Delete all notes that are tagged with the keyword ‘Mathematik’.

B. Results of performance evaluation

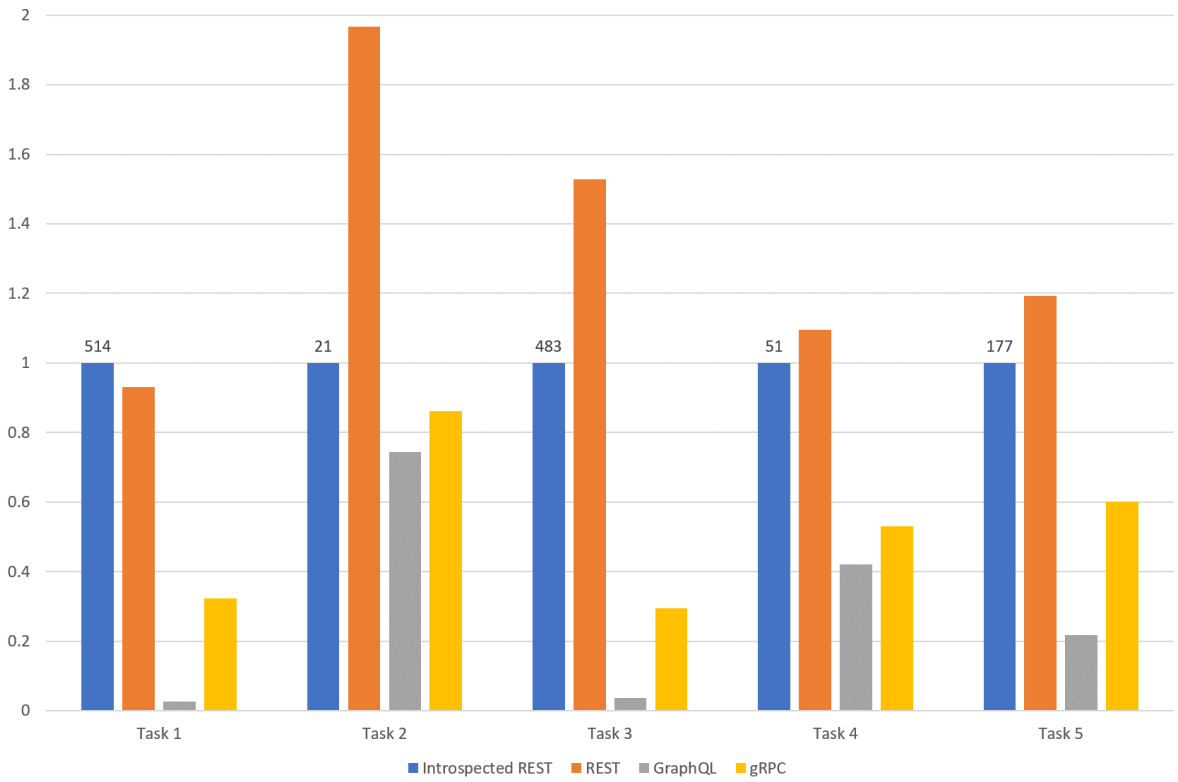

Figure 1: Average response time per task and API client in relation to Introspected REST client

C. Simulated changes for evolvability evaluation

C1 Add a new mandatory field to a request

C2 Add a new optional field to a request

C3 Remove a field from a response

C4 Rename a term

C5 Add a field to a response

C6 Add a step to a workflow

C7 Add a constraint for moving to the next step in a workflow

C8 Prohibit unauthorized access to a resource

C9 Add new functionality to the API

D. Classification of the simulated changes

	Int.	REST	REST	GraphQL	gRPC
C1	✓	✓	✓	✗
C2	✓	✓	✓	✓
C3	✗	✗	✗	✗
C4	✓	✓	✗	✗
C5	✓	✓	✓	✓
C6	✓	✓	✓	✓
C7	✓	✓	✗	✗
C8	✓	✓	✗	✓
C9	✓	✓	✓	✓

Table 1: Classification of changes as non-breaking (✓) or breaking (✗) per API style

E. Results of heuristic usability evaluation

	Int. REST	REST	GraphQL	gRPC
Easy to get started	++	++	+++	+
Visibility of actions	++	++	++	+
Meaningful error messages	+	+	++	+
Helpful documentation	++	++	+++	+++
Prevent invalid inputs	+	–	+++	+++
Not showing impermissible actions	++	+++	–	–
As small as possible	++	–	+++	–
Small steps	++	++	–	–
Layering	+++	+++	+	–
Satisfy expectations	++	+	+++	++
Recognizable patterns	++	+	+++	+

Table 2: Assessment of API styles regarding compliance with heuristics