API Modelling Techniques– Increasing Software's Appetite for the World

Sirvan Almasi
Imperial College London
s.almasi@imperial.ac.uk

William J.Knottenbelt
Imperial College London
wjk@imperial.ac.uk

***Abstract*—The importance and number of web applications are increasing. It is also common for many of these web applications to have well documented and consistent APIs. The question of how to communicate one's API to a human and machine is solved by tools and standards such as the OpenAPI. However, writing and maintaining a lengthy JSON or YAML document is not user friendly. In this paper we take a design first approach and look at human-centric methods in crafting an API from scratch using model-driven and visual techniques. This short paper attempts to be exploratory and thus offers three methods in how together or alone they can add value and be better than existing methods. These methods are more concise in communicating an API and offer an engaging route to the development of an API. We believe this approach is most beneficial to innovative organisations that require rapid prototyping. The end results of our methods are that the user can go from an idea to a working API infrastructure quicker than using existing tools and methods.**

I. Introduction

Web services are eating the world. This is of course a play on Marc Andreessen's 2011 essay title—Why Software is Eating the world [1]. The internet and the web is growing and migration of desktop applications to the web are evidence to our statement: Autocad [2] and Microsoft Office [3] are two important examples of popular desktop applications that have traces on the web now. There are several reasons for this to-the-web migration: 1) Users are using multiple devices; 2) Collaborative working. The likes of Google Docs [4] have proven that this web-based working is a successful innovation. The Chromebook is an implied intent that web-based working can become the norm. So, not only web applications are growing naturally but more desktop applications are migrating to the web too.

Our motivation for this paper is based on the arguments that software is an economic activity and that innovation requires software for which rapid development and testing is required. We look at three methods in improving the process of web API development by making them more human-centric.

Software development is an economic activity [5]. Programmers have often spent time on the wrong aspects and fallen to *"premature optimisation"* [6]. We have increased our pace in the software economic race through abstractions. One can tell the history of software through a lens of abstraction. Abstraction has enabled greater productivity whilst allowing for an increasing complexity underneath the abstraction. Why is this abstraction needed? Because it reduces the friction costs (time, cognitive demand, etc.) Do we need to treat software development as an economic activity? As of 2021 we see more innovations that are either software based or are dependent on software. John Holland describes the innovation process as a search through a large space of possibilities [7]. Whilst we are getting more intelligent with innovation, there is still a need to search through a large space of possibilities. Therefore, we need to decrease the cost of the search whilst increasing the pace of the search. So, yes, software has to be treated as an economic activity.

The OpenAPI specification [8] is a popular tool and is often used to create machine readable API specification for RESTful APIs. OpenAPI is an API descriptive language authored by Tony Tam in 2010 (first release in 2011), according to Tony Tam [9], the motivation was due to the problem of communication related to consuming web services. That is not knowing how to consume an API. According to Tony himself, the inception of Ruby, Python and PHP as web development languages and complexity of WSDL and SOAP led to the problem. OpenAPI is a simple contract for an API (what to give and what to expect). Why not begin with designing that contract? Planning an interface for software first is actually a good idea. This is the design first approach. We believe a design first approach can save time in developing what we call *Whole Software*—source code, documentation and architecture. The OpenAPI approach can link documentation with source code and it can also be used to generate code. Therefore, building whilst designing is possible.

Our main contribution is the exploration of alternatives methods for designing and building web APIs rapidly. We develop three methods that show promises in being more engaging and more human-centric than writing raw OpenAPI documents.

II. Background

Representational State Transfer (REST) is an "architectural style for distributed hypermedia systems" [10], and it was created by Roy Fielding in 2000. It is merely an attempt to standardise the usage of HTTP when creating web applications and their corresponding APIs. The usefulness of the REST architecture style is the deliberate attempt in creating constraints in the design process. These constraints are useful in focusing

the decision making flow. REST constraints and elements are shown in Table I.

Constraints	Elements of REST
Client-Server	Resource
Stateless	Resource identifier
Cache	Representation
Uniform Interface	Representation metadata
Layered System	Resource metadata
Code-On-Demand	Control data

TABLE I: Table showing the elements and constraints of the REST architectural style as described by Roy Fielding

As discussed in the introduction, there came a problem with the *communication* of RESTful APIs as this style became more common. How does one discover and learn to use an API? What data is the endpoint expecting and what should I expect back in return for a request? Knowledge of a resource identifier was no longer good enough—especially for complex applications. This is a documentation problem but also a design and development problem at the same time. We can either generate some documentation from our source code or design it first and have that guiding both documentation and development—a blueprint. Moreover, having a *machine-readable* description of an API can also assist in code and document generation. hRESTS [11] is one of the earliest attempt in standardising and structuring the description of an API.

Model-driven design of RESTful APIs have been rare but not non-existent. The work of M. Laitkorpi et al. [12] to our knowledge is the first attempt in applying model-driven techniques to RESTful APIs. Model-driven techniques for the semantic web applications [13] offer a more granular and heavily graphical approach. In our opinion taking a resource first approach yields better outcome and is easier to communicate. Model-driven designs have the disadvantage of becoming overly complex as all the data is displayed on one layer.

III. OpenAPI design patterns

In this section we will look at the relationship between the human and the process of creating an OpenAPI document.

OpenAPI is a JSON/YAML data structure that often comes in a single file. The specification does, however, note that it can be split into multiple files. The structure of OpenAPI Version 3 is as follows.

- **Metadata**: Information about the author, server, contact, etc.
- **Paths**: Details about paths and their methods. The paths include the type of data expected and the type data and response type that will be returned to the user.
- **Components**: These are re-used components such as schema, parameter and security components.

The following are some of the important limitations of OpenAPI that we think needs to be addressed.

A. Top down structure

Majority of OpenAPI documents use references (`$refs`) to link to common schemas or parameters. This linear structure in our opinion is wrong. It forces the user to create paths first rather than the schema or resources. Do remember that RESTful API is *resource* orientated architecture. So, we must begin with the resource first approach in designing an API.

B. Paths

A path is a *resource identifier* (URL, URN). It often points to a known schema or a parameter component. The disjoint structure means that one has to scroll very far to see their components when designing them. Moreover, paths can be predicted from resources. Therefore, we can generate the path from a resource or schema. This step can be automated so that the human designer can modify the generated path rather than creating it from scratch.

C. Verbosity

The next issue is the verbosity of an OpenAPI document. Given that we are taking the design first approach and assuming to be designing a novel API, we require some immediate feedback in order to test and re-adjust our course of action if we are heading in the wrong direction. We may use art as an analogy here: art rarely begins with a master stroke and the completion of one area before moving to another area of the canvas. The artist may begin with a sketch or an outline before adding layers to the whole. Software design and the OpenAPI specification enforces the unnatural habit of perfecting one section before moving on; this also sounds similar to premature optimisation. An example of this is descriptions and verbosity of describing inputs and outputs: A single API will rarely deviate from a given response type, e.g. `json`, however, in OpenAPI we are required to type in this for all of our responses.

Descriptions are seldom useful at the beginning and should come later once the user has some 'sketch' or feedback from their creation. e.g. for the path `/pets/{petId}` [14] with a single parameter of `petID`, the description, *"The id of the pet to retrieve"*, does not add any value.

D. Common parameters in schemas

We find that many schemas either reference each other or can be a subset of one another. A human designer can visually identify this relationship quickly if it was not written in a data structure. Moreover, the human designer can relate the schemas via non-text methods.

IV. New approaches

In this section we will look at new approaches in overcoming the shortfalls of API development and especially OpenAPI. Recall that our intent is to speed up the process of web API development in the context of novel and innovative projects. This intent produces the following requirements.

- Must allow user feedback in order to adjust course of action.

- Improve productivity:
  - Motivation: Engaging and flow simulating workflow.
  - Decreasing repetitive work: Potentially use code generation and automation to reduce repetitive work.
  - Providing assistance: For example, provide data type suggestions.

The above requirements naturally force us to explore non-text based programming. Domain Specific Languages (DSL) have been the driving force in reducing repetitive work. Ruby, SASS and SQL are examples of such languages. Model and flow-based programming in engineering have proven to be time and cost efficient [15]. LabView [16], Simulink [17], and specific use cases such as at Motorola [18] are great examples of how alternative and more human-centric design and programming have been successful in other fields.

For the following methods we will demonstrate the concepts using the OpenAPI PetStore example[1]. Figure 1 shows the PetStore schemas as a `YAML` format and alternative means of communicating the same thing.

We will focus on two areas with respect to each of the design methods below. These are two weak areas in the current approach that we think can be improved, they are as follows.

1. Representation of resources and their relationships to one another. Using the PetStore example, we see that the schema Pet and NewPet have a relationship which can be noted as follows. $Pet = \{id, name, tag\}$ and $NewPet = Pet \setminus \{id\}$.
2. Automating future steps. Often, paths can be generated from a resource. Most resources undergo some CRUD operation in their lifetime, all of which can be automated to a certain extent.

A. Model and flow-based programming

Flow-based programming was developed by J. Paul Morrison during the early 1970s [19]. It is a non-linear approach to developing applications: it is based on the idea that an application is continuously transformed by multiple data streams as opposed to a single actor manipulating it. It is a type of data-flow programming, and also takes us towards visual programming.

This method is ideal for web APIs, mainly because we can think of API endpoints as nodes with inputs and outputs (independent of other endpoints). Figure 2 visualises a concept in how we can use visual and flow-based programming techniques in creating a web API design tool.

[1]https://github.com/OAI/OpenAPI-Specification/blob/main/examples/v3.0/petstore-expanded.yaml

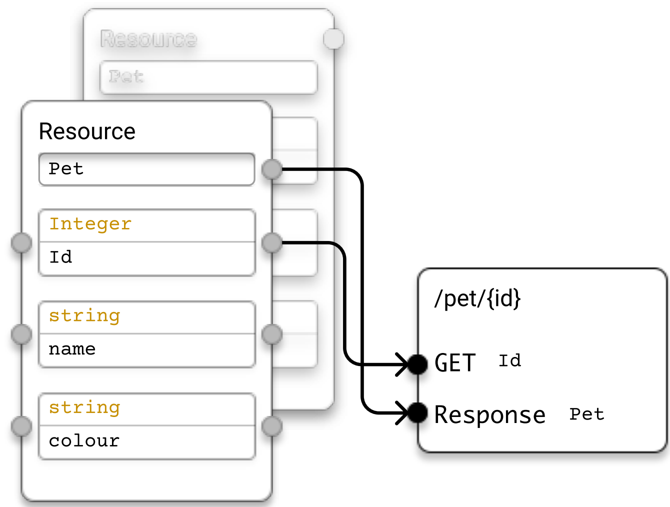

Fig. 2. Concept of how visual programming and data-flow programming can be combined to create a method for developing web APIs.

B. Tabular and spatial-based design

We can visualise the schema relationships via a table. This table is easier to digest and comprehend that `NewPet` is a subset of `Pet`. Figure 3 demonstrates how we might be able to achieve this.

Pet	id: Integer	name: String	tag: String
NewPet	0	1	1

Fig. 3. Tabular representation of the relationship between two schemas as per $Pet = \{id, name, tag\}$ and $NewPet = Pet \setminus \{id\}$.

We can do more with this tabular format of a schema. We can represent more useful information on the schema which then is inherited by future operations. Figure 4 shows how we can represent user security methods and output formats. The benefit of this is that we don't have to specify these security information in our endpoints or specify an output format. Moreover, relationships inherit these features. In our example, `NewPet` would inherit the security constraint that the field `tag` can only be manipulated by an admin.

GET auth-users
POST auth-users
PUT auth-users
DELETE admin

GET admin
POST admin
PUT admin
DELETE _

Pet	id: Integer	name: String	tag: String
NewPet	0	1	1

json xml

Fig. 4. One can place more useful information on the tabular version of the schema. This example shows how 'Pet' and its subsets can only be manipulated by auth-users and admin. We can also set our output formats here on the same table.

C. Domain Specific Language (DSL)

DSLs are strong contenders to create web APIs. Many DSLs are focused on the entire development of the web application.

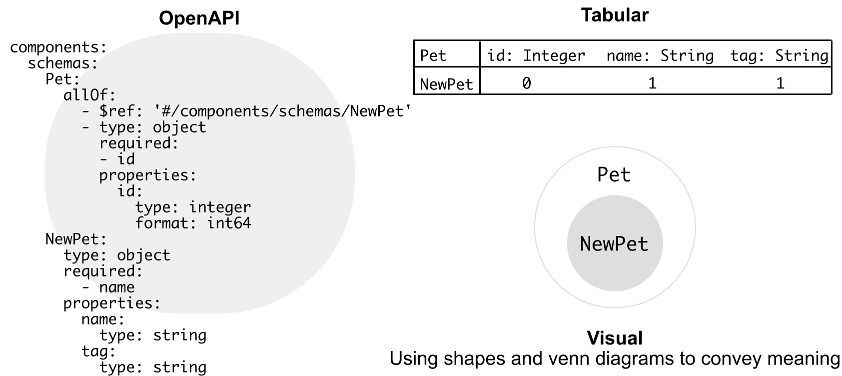

Fig. 1. This figure shows two OpenAPI schemas for which one is a subset of another. On the right we show the corresponding visualisation of the same data structure.

However, we are more concerned with setting up a basic infrastructure. Listing 1 shows a snippet of DSL that we have been experimenting with in creating endpoints for the Pet Store example.

Listing 1. DSL showing how one might be able to generate endpoints for a web API.

```
(endpoint "/pet"
  :get ((tag, limit) -> [Pet])
  :post (newPet -> Pet))

(endpoint "/pet/(pet :id)"
  :get (_ -> Pet)
  :delete (_ -> (204, "Pet deleted"))
  :put (newPet -> Pet))
```

D. Structure

The methods that we have described would benefit from the following structure.

- **Users**: User types and groups that can be used in the other sections.
- **Security**: Users and security have an explicit relationship and therefore they should follow one another.
- **Resources**: We specify our resources in this section and their relationships to one another.
- **Paths**: Certain paths are automatically generated for the resources that were described in the step above.

V. Discussion

We believe a design first methodology can bring efficiency in software development and maintenance. For web APIs, designing the interface first introduces positive constraints. Such constraints can be helpful in focusing the project.

This work has been exploratory in its nature, and there needs to be a comparison of each of the methods described in order to understand their limitations. Whilst we don't yet know which method is faster in developing an API, we do know that they are more concise and more engaging to work with than raw OpenAPI or programming from scratch.

What we have described can be considered a blueprint and one can argue that source code eventually decouples from its blueprint. So far we haven't had a good example of blueprints staying forever in sync with their code. We can argue that in our context of rapid prototyping it is fine to eventually decouple from code and blueprint as we are more focused on small scale web APIs.

VI. Future work

We have provided some novel techniques that each require further design and development. We have been developing and experiment with the proposed methods. Our first step is to finish the tabular method and the code generation that sits alongside it. Next, we will create and compare the tabular method with the flow-based programming using user studies. We will relate these findings with a base case study that uses OpenAPI and programming first methods.

We can extend the approaches described here for non-REST API styles, such as GraphQL [20] and even gRPC [21].

VII. Conclusion

In this paper we noted the weaknesses within the current OpenAPI design and development process. The unnatural structure of the OpenAPI makes it unintuitive to get feedback and design rapidly. OpenAPI is verbose and forces premature optimisation. And lastly, the most important aspect of REST, resources, is demoted to the bottom of an OpenAPI specification. Automating endpoint generation once a resource has been defined is an example of how we can speed up the design cycle.

We proposed three new methods for human-centric web API design and development. Flow-based programming and visual methods allow us to reduce the education barrier whilst improving the feedback of ones work. Our novel tabular format is more concise and is more intuitive to use. The tabular format also allows us to embed security information and response formats for a given resource. These properties improve security as they can cascade down to the endpoints and other operations on the resources.

REFERENCES

[1] M. Andreessen, "Why Software Is Eating The World," *Wall Street Journal*, Aug. 2011. [Online]. Available: https://online.wsj.com/article/SB10001424053111903480904576512250915629460.html

[2] "AutoCAD Web App - Online CAD Editor & Viewer | Autodesk." Apr. 2021. [Online]. Available: https://web.autocad.com/login

[3] "Office 365, Microsoft Office," Apr. 2021. [Online]. Available: https://www.office.com/

[4] "Google Docs," May 2021. [Online]. Available: https://docs.google.com

[5] Oracle Developers, "Clojure Made Simple," Jun. 2015. [Online]. Available: https://www.youtube.com/watch?v=VSdnJDO-xdg&ab_channel=OracleDevelopers

[6] D. Knuth and D. J. Fuller, *Art of Computer Programming, Volumes 1-4A Boxed Set, The*, 1st ed. Amsterdam: Addison-Wesley, Mar. 2011.

[7] J. Holland, "Innovation in Complex Adaptive Systems: Some Mathematical Sketches | Santa Fe Institute," 1993. [Online]. Available: https://www.santafe.edu/research/results/working-papers/innovation-in-complex-adaptive-systems-some-mathem

[8] "OAI/OpenAPI-Specification," OpenAPI Initiative, Apr. 2021. [Online]. Available: https://github.com/OAI/OpenAPI-Specification

[9] IBM Developer, "Tony Tam recounts the history of Swagger and the Open API Initiative," Mar. 2016. [Online]. Available: https://www.youtube.com/watch?v=oxqZ9J6t420&ab_channel=IBMDeveloper

[10] R. Fielding, "Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)," 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[11] J. Kopecký, K. Gomadam, and T. Vitvar, "hRESTS: An HTML Microformat for Describing RESTful Web Services," in *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, vol. 1, Dec. 2008, pp. 619–625.

[12] M. Laitkorpi, P. Selonen, and T. Systa, "Towards a Model-Driven Process for Designing ReSTful Web Services," in *2009 IEEE International Conference on Web Services*, Jul. 2009, pp. 173–180.

[13] M. Brambilla, S. Ceri, F. M. Facca, I. Celino, D. Cerizza, and E. D. Valle, "Model-driven design and development of semantic Web service applications," *ACM Transactions on Internet Technology*, vol. 8, no. 1, p. 3, Nov. 2007. [Online]. Available: https://dl.acm.org/doi/10.1145/1294148.1294151

[14] "OAI/OpenAPI-Specification PetStore Example," Aug. 2019. [Online]. Available: https://github.com/OAI/OpenAPI-Specification

[15] "General Motors Developed Two-Mode Hybrid Powertrain With MathWorks Model-Based Design; Cut 24 Months Off Expected Dev Time," Oct. 2009. [Online]. Available: https://www.greencarcongress.com/2009/10/general-motors-developed-twomode-hybrid-powertrain-with-mathworks-modelbased-design-cut-24-months-of.html

[16] "LavView: Graphical system engineering software," May 2021. [Online]. Available: https://www.ni.com/en-gb/shop/labview.html

[17] "Simulink - Simulation and Model-Based Design," May 2021. [Online]. Available: https://uk.mathworks.com/products/simulink.html

[18] T. Weigert, "Practical Experiences in Using Model-Driven Engineering to Develop Trustworthy Computing Systems," in *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing -Vol 1 (SUTC'06)*, vol. 1. Taichung, Taiwan: IEEE, 2006, pp. 208–217. [Online]. Available: http://ieeexplore.ieee.org/document/1636178/

[19] J. P. R. Morrison, "Flow-based Programming." [Online]. Available: https://jpaulm.github.io/fbp/

[20] "GraphQL Query Language," May 2021. [Online]. Available: https://graphql.org/

[21] "gRPC," May 2021. [Online]. Available: https://grpc.io/