Top-Down Model-Driven Engineering of Web Services from Extended OpenAPI Models

David Sferruzza

Top-Down Model-Driven Engineering of Web Services from Extended OpenAPI Models

David Sferruzza
LS2N - UMR CNRS 6004
Nantes, France
david.sferruzza@ls2n.fr

ABSTRACT

Web services engineering is a crucial subject, because web services are often built to be used by other programs; thus they should have a good documentation targeting developers. Furthermore, when building a digital product, engineers need to build several programs that interact with a central instance of web services. OpenAPI, a popular industry standard, makes possible to document web services in order to quickly make a prototype of the product. It allows a top-down process where developers iterate to build an OpenAPI model that describes the web services they want, and then implement both the web services and the programs that will consume them. However, when doing such rapid prototyping, developers tend to either skip this design phase and implement web services right away, or stop updating the OpenAPI model when the product is released; in both cases they cannot take advantage of having an OpenAPI model aligned with the implementation. We show how OpenAPI can be extended to add implementation details inside models. These extensions link services to assemblies of components that describe computations. Hence a top-down development process that keeps model and implementation aligned. Moreover, this makes possible for developers to benefit from more support features while keeping the same flexibility.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; *Domain specific languages*; *Software prototyping*;

KEYWORDS

Web Services, Model-Driven Engineering, Documentation, Code Generation, OpenAPI 3.0

ACM Reference Format:
David Sferruzza. 2018. Top-Down Model-Driven Engineering of Web Services from Extended OpenAPI Models. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3238147.3241536

1 INTRODUCTION

Context. Web services are key elements of many digital products. Indeed, these products often require to be available on several platforms (web, Android, iOS, …) thus can benefit from separating concerns: web services centralizing data and business logic have different life cycles than the several user interfaces that rely on them. This architecture is quite common; for example, after several years of coaching startups at Startup Palace[1], it appears that a lot of them are using it or should be. One of the difficulties of this approach is to design and maintain a consistent interface between the web services and the other programs, which are sometimes developed by other teams or even unrelated people. Fortunately, many tools and languages exist to help designing, writing and maintaining such interfaces. OpenAPI [6] is one of them: it defines a specification to describe the HTTP APIs of web services in a language-agnostic way. It is quite famous in the industry and has a rich ecosystem of tools[2].

Motivation. Building or fast-prototyping digital products can be achieved using a top-down approach. First, they design an OpenAPI model and make sure it is suitable as an interface between the different actors. Then, they can rely on this model to implement them independantly using programming languages. Some tools like *Swagger Code Generator* [13] provide support to developers in this process: they generate code based on the OpenAPI model, targeting one of the numerous supported languages and frameworks. However, because OpenAPI is language-agnostic, the generated code does not contain implementation details; developers still have to complete it in a way that respects the contract defined by the OpenAPI model, so that they obtain a working implementation. Focusing on web services, this means that, apart from this one-time generation, the OpenAPI model and the actual implementation will need to be maintained in a separate manner, which in many cases results in the implementation being maintained and the OpenAPI model being outdated. This becomes more problematic when the digital product evolves a lot (i.e. it is a MVP[3]) because developers cannot benefit again from the code generation step when doing another top-down cycle as it would override precedently customized code.

Contributions. To overcome this issue and give even more support to developers, we extend the OpenAPI 3.0 Specification to make possible to add some implementation details in models, and

[1]https://www.startup-palace.com
[2]https://github.com/OAI/OpenAPI-Specification/blob/master/IMPLEMENTATIONS.md
[3]Minimum Viable Product: a finished but lightweight product made to test market hypotheses.

provide a tool to generate working web services from extended models.

Implementation details added to OpenAPI models consist of assemblies of components that represent computation units. These components can be defined in two ways: *(i)* by a contract and an implementation in a programming language, or *(ii)* in term of already defined components. Every service is linked to a component whose purpose is to generate an HTTP response.

Our tool, SWSG [10], provides support to automatizes the top-down development process. It can read these extended models, check several properties to ensure their consistency, and generate code of working web services. On top of solving the alignment problem, this gives even more support to developers.

The article is structured as follows. Section 2 describes the component system and how it is integrated in OpenAPI 3.0. Section 3 introduces SWSG and shows how it checks models and generates code. Section 4 presents related work. Finally, Section 5 concludes the article with some lessons and future work.

2 EXTENDING OPENAPI

The OpenAPI Specification defines a standard to express interfaces to HTTP APIs in a language-agnostic way. It aims at allowing "both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection" [6]. As in Model-Driven Engineering (MDE), the point of having such meta-models is to have tools that can rely on them in order to safely manipulate models and offer support to developers. Indeed, the growing ecosystem of OpenAPI contains numerous tools that can be used in a top-down process, which often means that an OpenAPI is given as input to a tool that will refine it and produce another artifact. For exemple, both *Swagger Editor* [14] and *Swagger UI* [15] provide an interactive graphical user interface from a model, and *Dredd* [1] generates functional tests from a model.

By design, OpenAPI does not contain implementation details that describe how the web services are to be implemented. As stated in the introduction, this can lead to a misalignment between the OpenAPI model and its implementation. To overcome this limitation while keeping the benefits of using OpenAPI models in a top-down process, we extend OpenAPI to add a new layer of information that describes the implementation of the web services at a high-level of abstraction. This layer consists of components definitions and instanciations such as those we introduced in a previous work [12].

Section 2.1 describes this new layer, and Section 2.2 shows how it is integrated in OpenAPI 3.0.

2.1 The Component System

Components are units of processes and computations that occur inside web services and whose purpose is to produce HTTP responses. Their execution happens in an isolated context, that can contain variables. They can mutate this context by adding and removing variables, or return an HTTP response. Components can be of two kinds: either atomic or composite.

Atomic components are defined by a name and four sets of variables: parameters, preconditions, additions and removals. Parameters are variables whose values must be provided when instanciating the component; it allows to design generic components that are easier to reuse. The other three sets of variables form the contract of the component: they define what variables it needs to access from the context (preconditions), what variables it will add to the context (additions) and what variables it will remove (removals). The point of this contract is to support a lightweight kind of static verification we call structural consistency. Atomic components must go along with an implementation in a programming language, which makes them very flexible.

Composite components are defined by a name, a set of parameters and a list of instances of components. Parameters work in the same way as for atomic components. Instances of components reference a component by its name and provide bindings and aliases, that are specific to the instance. Bindings associate a parameter of the component to a value, and aliases allow to rename a variable from the component's contract in a local way. Composite components do not need to be implemented using a programming language because they are defined in term of other components. Indeed, their behavior consists of executing their subcomponents sequentially, each getting the context output by the previous. However, this sequential execution is interrupted if any of the components outputs an HTTP response instead of a new context.

This component system was voluntarily designed to be simple and offer a good trade-off between expressiveness and ease of support. That is why, for example, sequential execution is the only way to combine components.

2.2 Extensions to OpenAPI 3.0

To make it possible to use this component system and describe implementations of web services from inside an OpenAPI model, the OpenAPI 3.0 Specification must be extended. Two kinds of informations must be added: definitions of components and, for each service, an instance of component. Hopefully, OpenAPI provides an extension mechanism that preserves tools compatibility: most of the schemas of the specification can be enhanced with more attributes, if their names start with `x-`. The full specification of our extensions is available in [11].

The Listing 1 shows an extract of the *Petstore* example [7], an official example of OpenAPI 3.0 model, that has been enhanced with our extensions. The `components` object contains two new properties: `x-swsg-cc` and `x-swsg-ac`. The first is a set of composite component definition; here we define a component called `FindPet` in terms of two other components named `GetPetById` and `RenderPet`. The second is a set of atomic component definition that contains the definitions of `GetPetById` and `RenderPet`. The Listing 2 shows how the `FindPet` component is instanciated from a service, in the `x-swsg-ci` attribute.

3 GENERATING WEB SERVICES

To use our extensions to OpenAPI 3.0, we propose the following top-down development process:

(1) developers write a standard OpenAPI model that fits their needs;

```
components:
  schemas:
    Pet:
      allOf:
        - $ref: '#/components/schemas/NewPet'
        - required:
          - id
          properties:
            id:
              type: integer
              format: int64
    NewPet:
      required:
        - name
      properties:
        name:
          type: string
        tag:
          type: string
  x-swsg-cc:
    - name: FindPet
      components:
        - component: GetPetById
        - component: RenderPet
  x-swsg-ac:
    - name: RenderPet
      pre:
        - name: pet
          type:
            entity: Pet
    - name: GetPetById
      pre:
        - name: id
          type: String
      add:
        - name: pet
          type:
            entity: Pet
```

Listing 1: Components Definition

```
paths:
  /pets/{id}:
    get:
      parameters:
        - name: id
          in: path
          description: ID of pet to fetch
          required: true
          schema:
            type: integer
            format: int64
      responses:
        '200':
          description: pet response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
      x-swsg-ci:
        component: FindPet
```

Listing 2: Service Definition

(2) they use our extensions to design components and associate them to every web services;
(3) they write an implementation for each atomic component;
(4) they run SWSG to check the model and generate code.

The step (2) is made possible by our extensions and provide a transitional step between the modelling (1) and the implementation (3) in the refinement process. It simplifies the implementation (3) by reducing it to several small functions instead of a whole program.

The step (4) makes use of our tool, SWSG [10], that follows the process shown in Figure 1. We present this tool in the following sections: Section 3.1 focuses on the model consistency verification and Section 3.2 on the code generation.

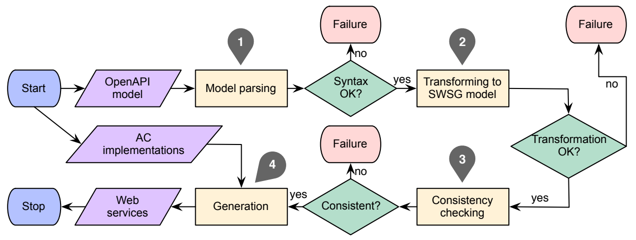

Figure 1: Process of SWSG

3.1 Consistency Verification

The third step of the process of SWSG (Figure 1) is a static verification of the structural consistency of the model. It consists of a set of formal rules that check the semantics of models to determine whether it allows for a safe code generation. This set of rules is derived from the one we presented in [12, ¶3]. Its purpose is to help developers to spot inconsistencies in models as earlier as possible.

For example, the model presented in Listings 1 and 2 is inconsistent: SWSG indicates a `PreconditionError`. Details of the error tell us that the `GetPetById` component is not given a variable of type `String` called `id`, as it requires. Indeed, the service definition shows that the `id` variable comes from a service parameter that has the `Integer` type. This kind of mistakes is quite common in software development, but having the right tools can mitigate its consequences. Here, SWSG detected the problem before we even run the code in production, so we just lost a small amount of time; we could have lost a lot more time if this code had been deployed (just for a typo).

When this error is fixed in the model, SWSG can proceed to the code generation step.

3.2 Code Generation

The process depicted by Figure 1 is generic: it does not rely on a specific language or technology. Yet the language and technologies used to implement atomic components must be identical or compatible with those of the code generation target. Because we experiment in *Startup Palace*'s context, the current implementation of SWSG targets the PHP programming language [4] with the *Laravel* web framework [5], which is a common tool stack.

The generation in itself is done with a model-to-text transformation that uses a template engine called *Twirl*[6]. Apart from some static classes and from the implementations of atomic components (that are not altered), SWSG generated two kinds of source files: implementations of composite components and a *route file*. The Listing 3 shows the code generated for the `FindPet` component (defined in Listing 1).

We took the approach of generating code that should never be manually edited. The generated code does not override any existing

[4]https://php.net/
[5]https://laravel.com/
[6]https://github.com/playframework/twirl

```php
<?php
// This is a generated file, do not edit
namespace App\Components;
use App\SWSG\Component, App\SWSG\Ctx, App\SWSG\Params;
class FindPet implements Component
{
    public static function execute(Params $params, Ctx $ctx)
    {
        $ctx0 = \App\Components\GetPetById::execute(new \App\SWSG\Params([]),
        ↪  $ctx);
        if ($ctx0 instanceof \Illuminate\Http\Response) return $ctx0;
        if ($ctx0 instanceof \SWSG\Ctx) $ctx0 = $ctx0;
        $ctx1 = \App\Components\RenderPet::execute(new \App\SWSG\Params([]),
        ↪  $ctx0);
        if ($ctx1 instanceof \Illuminate\Http\Response) return $ctx1;
        if ($ctx1 instanceof \SWSG\Ctx) $ctx1 = $ctx1;
        return $ctx1;
    }
}
```

Listing 3: Generated Code for the `FindPet` Component

files in a *Laravel*'s architecture and can be easily hooked to an existing web application through configuration.

In case of evolution of the needs (which is likely when working with MVPs), developers have to go through the top-down development process again, but can reuse most of the existing items (parts of the extended OpenAPI model, implementation of atomic components). This approach preserves alignment between the model and the web services, and fosters reuse of components.

4 RELATED WORK

The use of MDE for development and automatic generation of web services or web applications is not a new topic [2, 3, 9]. Indeed, this work is built on top of the approach of SWSG [12] and Reifier [8].

Through SWSG, our process shares the meta-modelling approach with tools such as *M3D* (introduced in [2] and extended in [3]) that also focus on building web services using MDE. One of the main differences between SWSG and *M3D* is that SWSG was developed with a focus on design-time support. Even if SWSG is definitely related to existing standards such as BPEL [4] or WSDL, our approach differs on several aspects. First, we want to avoid the shortcomings described in [5], that is WSDL models contain too much technical details and are difficult to understand for humans. Indeed our meta-model is simpler and less expressive than WSDL or BPEL. Second, this allows SWSG to provide more support to users; the balance between flexibility and support is discussed in [16].

One of the tools featured in the OpenAPI ecosystem is *Swagger Code Generator* [13]. It aims at generating client librairies, server stubs or documentations from an OpenAPI model. It supports many languages and frameworks, but only helps developers to write new services by generated boilerplate code. This automatizes a tedious task; however they still need to add a lot of code on top of it. Moreover, when services evolve, developer need to manually propagate evolutions into the codebase because *Swagger Code Generator* isn't able to merge them automatically. Our approach solves this issue because it gives flexibility to developers before the code generation step, making useless editing generated code. This ensures alignment between the (OpenAPI) model and the web services.

5 CONCLUSION

We extended OpenAPI 3.0 with a component system to describe implementations of web services from a high-level. This establishes an intermediary artifact that fits well in the top-down development process, between the abstract OpenAPI model and the concrete implementation of web services. This allows to generate an implementation of the web services that does not need manual modifications. As a consequence, alignment between the OpenAPI model and the implementation is guaranteed. Moreover, this paves the way for enhanced support features; for example, verification or interactive visualization. We also presented SWSG, a tool that can check structural consistency of extended OpenAPI models and generate code of the web services.

This approach was tested on several case studies[7], but needs more testing on bigger projects. Another perspective is to generalize SWSG to support more languages and technologies. This would probably lead to the formalization of interfaces to make it possible to plug external code generators. Also, it would be interesting to allow SWSG to check the compliance between implementations of atomic components and their contract. In a similar manner, calling external tools could be a solution to keep SWSG generic.

REFERENCES

[1] Apiary. 2017. Dredd. https://github.com/apiaryio/dredd.

[2] Mario Luca Bernardi, Marta Cimitile, Giuseppe Di Lucca, and Fabrizio Maria Maggi. 2012. M3D: a tool for the Model Driven Development of Web Applications. In *Proceedings of the Twelfth International Workshop on Web Information and Data Management*. WIDM 2012. Maui, HI, USA, (Nov. 2, 2012), 73–80.

[3] Mario Luca Bernardi, Marta Cimitile, and Fabrizio Maria Maggi. 2016. Automated development of constraint-driven web applications. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 1196–1203.

[4] Xiang Fu, Tevfik Bultan, and Jianwen Su. 2004. Analysis of interacting BPEL web services. In *In Proc. 13th Int. World Wide Web Conf.* Citeseer.

[5] Roy Gronmo, David Skogan, Ida Solheim, and Jon Oldevik. 2004. Model-driven web services development. In *E-Technology, e-Commerce and e-Service*. EEE'04. IEEE, 42–45.

[6] Open API Initiative. 2017. OpenAPI Specification. (Dec. 7, 2017). https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1.md.

[7] Open API Initiative. 2017. The Petstore Example. Version 3.0.1. (Dec. 7, 2017). https://github.com/OAI/OpenAPI-Specification/blob/3.0.1/examples/v3.0/petstore-expanded.yaml.

[8] Jérôme Rocheteau and David Sferruzza. 2016. Reifier: Model-Driven Engineering of Component-Based and Service-Oriented JEE Applications. In ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. Saint-Malo, France, (Oct. 5, 2016).

[9] Markus Scheidgen, Sven Efftinge, and Frederik Marticke. 2016. Metamodeling vs Metaprogramming: A Case Study on Developing Client Libraries for REST APIs. In *European Conference on Modelling Foundations and Applications*. Springer, 205–216.

[10] David Sferruzza. 2017. Safe Web Services Generator. https://gitlab.startup-palace.com/research/swsg.

[11] David Sferruzza. 2018. Specification of SWSG extensions for OpenAPI. (2018). https://gitlab.startup-palace.com/research/swsg/blob/master/openapi-extensions-specification/1.0.0.md.

[12] David Sferruzza, Jérôme Rocheteau, Christian Attiogbé, and Arnaud Lanoix. 2018. A Model-Driven Method for Fast Building Consistent Web Services in Practice. In 6th International Conference on Model-Driven Engineering and Software Development. Funchal, Madeira, Portugal, (Jan. 23, 2018).

[13] SmartBear Software. 2018. Swagger Code Generator. Version 3.0.0-rc1. (May 29, 2018). https://github.com/swagger-api/swagger-codegen/.

[14] SmartBear Software. 2018. Swagger Editor. https://github.com/swagger-api/swagger-editor.

[15] SmartBear Software. 2018. Swagger UI. https://github.com/swagger-api/swagger-UI.

[16] Wil M.P. van der Aalst, Maja Pesic, and Helen Schonenberg. 2009. Declarative workflows: Balancing between flexibility and support. *Computer Science-Research and Development*, 23, 2, 99–113.

[7] https://gitlab.startup-palace.com/research/swsg/tree/master/examples