# A Model-Driven Method for Fast Building Consistent Web Services from OpenAPI-Compatible Models

David Sferruzza, Jérôme Rocheteau, Christian Attiogbé, Arnaud Lanoix

# A Model-Driven Method for Fast Building Consistent Web Services from OpenAPI-Compatible Models

David Sferruzza[1,3], Jérôme Rocheteau[1,2], Christian Attiogbé[1], and Arnaud Lanoix[1]

[1] LS2N - UMR CNRS 6004 / F-44322 Nantes Cedex 3, France
david.sferruzza@ls2n.fr, christian.attiogbe@ls2n.fr, arnaud.lanoix@ls2n.fr
[2] ICAM / 35, avenue du Champ de Manœuvres, 44470 Carquefou, France
jerome.rocheteau@icam.fr
[3] Startup Palace / 18, rue Scribe, 44000 Nantes, France
david.sferruzza@startup-palace.com

**Abstract.** Lots of software companies rely on web technologies to test market hypotheses in order to develop viable businesses. They often need to quickly build web services that are at the core of their Minimum Viable Products (MVPs). MVPs must be reliable whereas they are based on specifications and hypotheses that are likely to change. Web services need to be well documented, to make it easy to develop applications that consume them. Model Driven Engineering approaches have been proposed and used to develop and evolve web services on one hand, and document them on the other hand. However, these approaches lack the ability to be suitable for both *(i)* rapid prototyping, *(ii)* model verification, *(iii)* compatibility with common programming languages and *(iv)* alignment between documentation and implementation. Here we propose a meta-model to express web services, the related tool to verify models consistency and an integration of this approach into the OpenAPI Specification. We adopt a shallow verification process to allow rapid prototyping by developers who are not formal methods experts, while still offering design-time guarantees that improve product quality and development efficiency. Web services are defined using parametric components which enable to express and formally verify web service patterns and to safely reuse them in other contexts. We built a tool to check consistency of extended OpenAPI 3.0 models and associated components implementations in order to generate corresponding web services. This allows us to give flexibility and verification support to developers, even in the context of an incremental development, as illustrated by a case study.

**Keywords:** Web Applications, Web Services, Model-Driven Engineering, Formal Verification, Code Generation, OpenAPI 3.0

## 1 Introduction

*Context.* Web agencies are software companies that often work with their customers to help them develop new projects involving the web. Most of these customers need web applications to bring value to their own customers. These web applications are built iteratively to limit their cost while allowing startups to converge toward a viable market. This approach begins by building a Minimum Viable Product (MVP). In this context, it is a web application with a high level of quality but a limited set of features. MVPs (among other kinds of web applications) need to be functional, reliable, usable and designed with users' emotions in mind, with only the features required to test market hypotheses. In this perspective, it is common to develop standalone web services that will be consumed by one or several applications which may provide user interfaces (UIs). For example: one instance of web services can be simultaneously used by several UIs, such as several mobile applications (Android, iOS, ...) and a web application. This allows separation of concerns and centralization of data and process logic, even in the case where there is only one consumer application. Therefore, for *Startup Palace*[4], a web agency that evolves in this context, the process of designing, building and evolving web services is important, because it is at the center of the MVP approach. When developing a MVP it is important to focus on features that really bring value to users, also called *game-changers*. Other features, called *show-stoppers*, do not bring value directly but are required to make the application functional, reliable or usable. *Show-stopper* features are often tedious to implement and so error-prone. Furthermore, because of the iterative process and the purpose of MVPs, specifications are likely to evolve, which can introduce bugs.

Moreover, it is important to design web services in a way that makes them actually usable. This implies providing a good documentation that can be used by developers writing consumer applications or by these consumer applications if they can adapt their behavior dynamically. While this kind of documentation can take many forms, some standards such as OpenAPI [11] and RAML [15] are widely used by the industry. These standards define specifications which describe HTTP APIs of web services and are the center of ecosystems of tools. This article focuses on OpenAPI 3.0 because it is the standard used at *Startup Palace*.

*Motivation.* Developers need abstractions to be able to safely express, isolate, reuse and evolve features. Some programming languages do provide such abstractions, along with modern type checkers that are able to statically verify consistency of programs. But this is not practicable in our context because we would like to leverage existing expertises of developers instead of forcing them to learn a new programming language and its ecosystem from scratch. We need to master the development of *show-stoppers*. We want to fix a common issue related to the tools around OpenAPI which rises when the process of evolving

[4] https://www.startup-palace.com

the web services occurs. These tools can be used in two main ways. First, with a forward engineering process, developers create manually an OpenAPI model, and use a tool to transform it. For example: a one-time generation of an implementation skeleton in a given technology. Second, with a reverse engineering process, a tool is used to extract an OpenAPI model from a working implementation (which can be enhanced by annotations). While the second process is useful to document existing web services, it cannot leverage the benefits of building web service using a top-down approach: from a high-level (model) to a low-level (implementation). Yet [6] shows the advantages of using different languages for programming-in-the-large and for programming-in-the-small. The first process makes possible to partially leverage these advantages, but the issue is that it lacks the ability to keep the OpenAPI model and the implementation aligned throughout the life of the project. Indeed, implementation is often obtained by a projection of the model that is then manually modified to implement business logic. Any following model evolution needs to be projected again which would require manual modifications to be re-applied from the beginning.

*Contribution.* This work is an attempt to solve the problem of building web services using safe abstractions on top of an existing programming language in order to ease development and reuse of *show-stopper* features, while keeping the web services in sync with their documentation (i.e. an OpenAPI model).

We introduce a meta-model to express web services and a corresponding semantics for verification. This leverages existing theory in Model Driven Engineering (MDE) [3, 16, 17] and a component-based approach in order to provide an expressive and language-agnostic solution to build web services. This meta-model does not allow to completely specify web services but is rather limited to a high-level representation in order to provide support while keeping models simple.

In [19] we have proposed a tool *(i)* to check models consistency and *(ii)* to generate working web services from a given valid model. As example, every component preconditions must be fulfilled in their instance contexts in order for a model to be consistent. This mechanism, coupled with the consistency checking, gives developers means to quickly and safely write and use components, and to reuse them in a reliable way.

We integrate the proposed approach in a top-down design process that relies on extended OpenAPI models. Extensions contain the parts of our meta-model that do not have any equivalent in the OpenAPI specification: the business logic. Because the OpenAPI 3.0 meta-model and ours are merged, the resulting models provide single sources of truth for building both web services and their documentation.

This article is an extended version of [21], published in the proceedings of MODELSWARD 2018. It differs by the following aspects:

1. the meta-model was slightly improved in order to add the possibility to better specify service parameters, to improve the type system and to make

re-usability easier by adding a mechanism to bind parameters in component definitions and in their arguments in instances' contexts;
2. the whole approach was redesigned as part of a more practical top-down workflow based on tools such as OpenAPI instead of being completely standalone;
3. this workflow is tested in a two-steps case study, showing how it works in the context of incremental development.

The article is structured as follows. Section 2 presents related work. Section 3 describes the meta-model of web services. Section 4 defines consistency of compliant models and shows how they can be checked. Section 5 shows how this approach can be integrated with OpenAPI. Section 6 introduces a tool to generate actual implementations from models. Section 7 shows a two-steps case study that illustrates our approach. Finally, Section 8 concludes the article with some lessons and future work.

## 2 Related Work

The use of MDE for development and automatic generation of web services or web applications is not a new topic [3, 2, 17]. Indeed, this work is built on top of the approach of SWSG [21] and Reifier [16].

SWSG shares the meta-modeling approach with tools such as *M3D* (introduced in [3] and extended in [2]) 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. For example, it allows to automatically verify some properties about the structural consistency of models. Even if SWSG is definitely related to existing standards such as BPEL [7] or WSDL, our approach differs on several aspects. First, we want to avoid the shortcomings described in [8], 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 [27]. Finally, SWSG now relies on OpenAPI.

OpenAPI [11] that is also involved in various research areas. In [4], it was chosen for its popularity over WADL and other industry standards in order to automatically transform plain HTML documentations of web services to a machine-readable format. Moreover, [5] provides a great state of the art of service description formats that brings out OpenAPI as the most promising choice at the moment and enriches it with semantic annotations. It seems to be an updated version of the work proposed in [26, §3.2]. [18] shows an approach that is agnostic to service description formats but uses OpenAPI in the article. The popularity of OpenAPI is also highlighted by its use in other domains. For example [30] describes a case where OpenAPI 2.0 is used in combination of other tools from the life sciences community and points out that the specification extension mechanism of OpenAPI 3.0 (that we use in this article) might be an

interesting opportunity of improvement. Another example in the telecommunication domain is presented in [14] which provides a section to emphasizes the trade-offs of Model-Driven Engineering and argues that they can be overcomed "with increased investment in the tools that support the development process"; we share the same vision in the present proposal.

One of the tools featured in the OpenAPI ecosystem is *Swagger Code Generator* [22]. It aims at generating client librairies, server stubs or documentations from an OpenAPI model. The server stubs generation supports many languages and frameworks, but, as its name states, only generates stubs. This helps developers to write new services by generated boilerplate code, which is a tedious task, but 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. It is worth mentioning that the code generators in themselves are based on similar model-to-text mechanisms: *Swagger Code Generator* uses the mustache[5] or the Handlebars[6] (depending on the version) template format whereas SWSG uses Twirl[7]. Finally, SWSG supports OpenAPI 3.0 models which is not the case of *Swagger Code Generator* at the time of writing[8].

## 3 A Meta-Model to Express Web Services

We introduce a meta-model of web services. This meta-model is voluntarily simple in order to provide two advantages: *(i)* to give developers good abstractions to write reusable code while giving them a good flexibility and *(ii)* to allow tools to provide support to developers, such as design-time consistency verification (see Section 4). This is obtained with a trade-off on a lower verification aspects.

### 3.1 Preliminary Notations

*Union.* The union or sum type of two types $T_1$ and $T_2$ is denoted $T_1 \uplus T_2$.

*Tuple.* A tuple $T$ is a product type between $n$ types $T_1$ to $T_n$, with $n \geq 2$. It is denoted $T \equiv T_1 \times \ldots \times T_n$. A value $t$ of type $T$ is written as $t = (t_1, \ldots, t_n)$ where $t_1 \in T_1, \ldots, t_n \in T_n$. The notation $t(x)$ is also used to designate $t_x$, where $x \in \{1, \ldots, n\}$.

[5] https://mustache.github.io/
[6] https://handlebarsjs.com/
[7] https://github.com/playframework/twirl
[8] The unstable version `3.0.0-rc0` does support OpenAPI 3.0 but is not yet finished and handle only a few languages and frameworks.

*Record.* A record $R$ is a tuple with labeled elements. It is denoted $R \equiv \langle label_1 : T_1, \ldots, label_n : T_n \rangle$. It is syntactic sugar over a tuple $T_1 \times \ldots \times T_n$ and $n$ functions $label_1 : R \to T_1, \ldots, label_n : R \to T_n$. As for tuples, a value $r$ of type $R$ is written as $r = (t_1, \ldots, t_n)$ where $t_1 \in T_1, \ldots, t_n \in T_n$. Associated functions can also be written $r.label_1, \ldots, r.label_n$. For example, for a record $person = (\text{“}Batman\text{”}, 35) \in \langle name : String, age : Int \rangle$ we have $name(person) = person.name = \text{“}Batman\text{”} \in String$.

*Set.* A set whose elements are all of type $T$ has the type $\mathcal{P}(T)$.

*List.* A sequence or list of type $T$ is a set of elements of type $T$ which are ordered. It is denoted $List(T)$. It is similar to a function from indexes $I$ to values of $T$, that is $List(T) : I \to T$ where $I$ is a $1..N$ and $N = card(List(T))$. It can also be written as $List(T) \equiv [t_1, \ldots, t_n]$ where $t_1, \ldots, t_n \in T$.

*Projection.* The projection of a set or list of tuples $S$ on the $i^{\text{th}}$ element of a tuple is written $Prj_i(S)$. Similarly, the projection of a set or list of records $S$ on one of the elements of a record $label_i$ is written $Prj_{label_i}(S)$. For example, for a set of records $S \in \mathcal{P}(\langle name : String, age : Int \rangle)$ where $S = \{(\text{“}Batman\text{”}, 35), (\text{“}Robin\text{”}, 26)\}$, $Prj_{name}(S)$ has type $\mathcal{P}(String)$ and is equal to $\{\text{“}Batman\text{”}, \text{“}Robin\text{”}\}$.

### 3.2 A Meta-Model of Web Services

A meta-model of web services is defined by the BNF grammar given in Figure 1. Figure 2 shows a slightly simplified version of this grammar in the form of a UML class diagram, to provide a quick glance.

This meta-model does not aim to replace existing standard meta-models like for instance the OpenAPI Specification [11] or RAML [15], but rather to be compatible and complementary with them (see Section 5). These meta-models allow to define programming language-agnostic interface descriptions for HTTP APIs (i.e. informal contracts) in order to be able for both humans and computers to discover and understand their capabilities, while our meta-model allows to express actual implementations of such HTTP APIs. Precisely, our meta-model is designed in order to match needs and requirements about verification (see Section 4), generation and ease of writing (see Section 6). For these reasons, and to be more accessible to practitioners, our approach does not rely on existing standards such as BPEL [7] or WSDL [8].

*Model.* A model of web services $m_i \in M$ is specified as a record of three elements: a set of entities of type $E$ that stands as data model, a set of components of type $C$ that stands as process model and an ordered list of services of type $S$ that exposes component to the outer world: $M \equiv \langle entities : \mathcal{P}(E), components : \mathcal{P}(C), services : List(S) \rangle$.

```
model ::= ⟨entities : entity*, components : component*,
           services : service*⟩
identifier ::= [A-Za-z][A-Za-z0-9_]*
entity ::= ⟨name : identifier, attributes : variable*⟩
term ::= variable | constant
variable ::= ⟨name : string, type : type⟩
constant ::= ⟨type : type, value : object⟩
type ::= string | boolean | integer | float | date | datetime | entity-ref
       | seq-of | option-of
entity-ref ::= ⟨entity : identifier⟩
seq-of ::= ⟨seqOf : type⟩
option-of ::= ⟨optionOf : type⟩
component ::= atomic-component | composite-component
atomic-component ::= ⟨name : identifier, params : variable*, pre : variable*,
                      add : variable*, rem : variable*⟩
composite-component ::= ⟨name : identifier, params : variable*,
                         components : component-instance*⟩
component-instance ::= ⟨component : identifier, bindings : binding*, aliases : alias*⟩
binding ::= ⟨param : variable, argument : term⟩
alias ::= ⟨source : variable, target : variable⟩
service ::= ⟨method : method, path : path, params : service-parameter*,
             component : component-instance⟩
method ::= [A-Z]+
path ::= .+
service-parameter ::= ⟨location : parameter-location, variable : variable⟩
parameter-location ::= query | header | path | cookie | body
```

**Fig. 1.** Meta-model BNF grammar

*Entity and type.* Entities are non-primitive data types. An entity $e_i \in E$ is represented by a record of two elements: a name and a set of variables that represent attributes: $E \equiv \langle name : Id, attributes : \mathcal{P}(V)\rangle$. An attribute of an entity can be another entity, making it a recursive type. An identifier $id_i \in Id$ is a string that matches the following regular expression: `^[A-Za-z][A-Za-z0-9_]*$`. A variable $v_i \in V$ is defined by a record composed of a name and a type: $V \equiv \langle name : String, type : T\rangle$. A type $t_i \in T$ can be *primitive* or *parametric*. A primitive type can be one of the followings: $String$, $Boolean$, $Integer$, $Float$, $Date$ or $DateTime$. Parametric types are represented by records that have one element by parameter of the type; available parametric types are defined in Table 1.

*Components.* Components are units of processes and computations that occur inside web services. Their execution happens in an isolated context, that can contain variables. They can mutate this context by adding and removing variables. A component $c_i \in C$ is defined as the union type of atomic components $AC$ and composite components $CC$: $C \triangleq AC \uplus CC$. Both types of components are defined by a name and a set of variables that express components' parameters. Because they have parameters, we call them parametric components. An atomic compo-

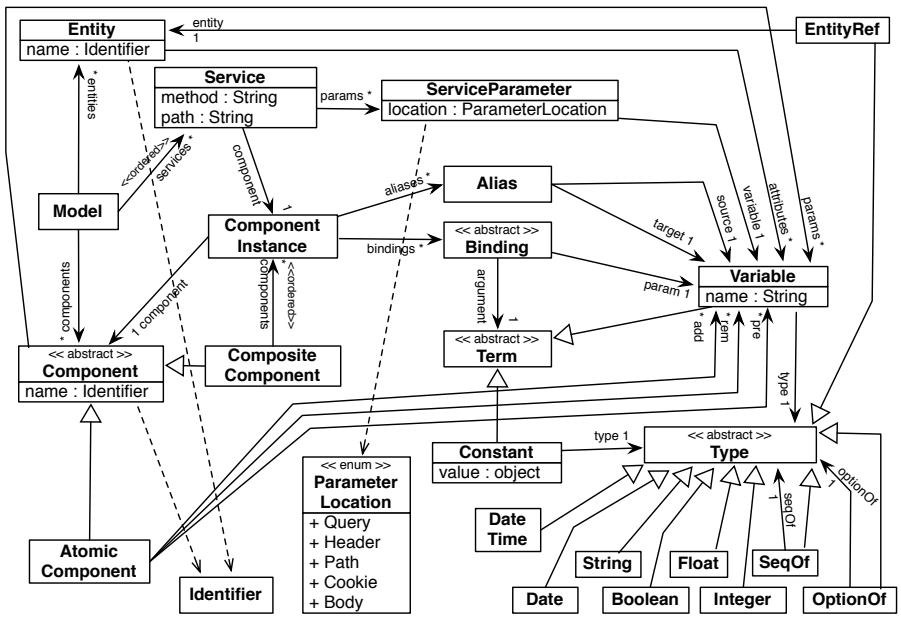


**Fig. 2.** Meta-model diagram

nent $ac_i \in AC$ is represented by a record of the following elements: name, parameters, preconditions (a set of variables that might be needed in the execution context), additions (a set of variables that will be added to the execution context) and removals (a set of variables that will be removed from the execution context): $AC \equiv \langle name : Id, params : \mathcal{P}(V), pre : \mathcal{P}(V), add : \mathcal{P}(V), rem : \mathcal{P}(V)\rangle$. The last three elements are sometimes referred to as the component's contract. A composite component $cc_i \in CC$ is represented by a record of the following elements: name, parameters and an ordered list of component instances: $CC \equiv \langle name : Id, params : \mathcal{P}(V), components : List(CI)\rangle$. A component instance $ci_i \in CI$ is represented by a record of three elements: a component, a set of bindings used to instantiate the component by associating arguments to

**Table 1.** Parametric types

| Type constructor | Parameters | Description |
|---|---|---|
| $Entity$ | $\langle entity : Id\rangle$ | A reference to an entity. The element `entity` given in parameter must correspond to the name of an entity defined in the model. |
| $SeqOf$ | $\langle seqOf : T\rangle$ | A sequence of elements of a given type. |
| $OptionOf$ | $\langle optionOf : T\rangle$ | The optional version of a given type, that is every element of the given type plus a special value `null` that represent the absence of value. |

its parameters and a set of aliases that allow to rename variables of the component's contract on the instantiation context: $CI \equiv \langle component : C, bindings : \mathcal{P}(\langle param : V, argument : Term \rangle), aliases : \mathcal{P}(\langle source : Id, target : Id \rangle) \rangle$. Terms can be variables or constant literal values ($Const$): $Term \triangleq V \uplus Const$. Atomic components are meant to be along with an implementation written using a programming language whereas composite components are not. Components can be seen as an abstraction to encourage separation of concerns and reusability by leveraging two mechanisms: composition and parametrization.

*Service.* Services are the entry points of web services. A service $s_i \in S$ is represented by a record of four elements: a HTTP method, a path, a set of expected input parameters and a component instance: $S \equiv \langle method : M, path : P, params : \mathcal{P}(\langle location : L, variable : V \rangle), component : CI \rangle$. A method $m_i \in M$ is a valid HTTP method name, as defined by RFC 7231[9]. A path $p_i \in P$ is a relative URL that can contain parameters whose names are to be placed inside braces; for example: `/user/{id}`. A location $l_i \in L$ represents where a given service parameter can be found in a HTTP request: $L \equiv \{query, header, path, cookie, body\}$. In a model of web services, services are gathered in an ordered list. This abstraction is very common in web frameworks and is often called *router*. Instead of considering a web application as a huge function of HTTP requests to HTTP responses, a router allows to dispatch HTTP requests to several such functions by filtering them declaratively by method and by path. That is to reduce complexity of the whole application by encouraging separation of concerns.

### 3.3 Concrete Syntax for Models of Web Services

We previously presented a mathematical definition of our meta-model in Section 3.2 and an equivalent BNF grammar in Figure 1. These notations are meant to introduce formal definitions that are used to define properties on the meta-model, which we do in Section 4. But they are cumbersome to read or write actual models.

We also introduce a concrete syntax that is more compact and readable. Because it is equivalent to BNF grammar in Figure 1, we won't define it formally here but only provide some intuition on it.

A model is represented, with respect to its formal definition, as an unordered list of definitions, each on its own lines. A definition starts with an element identifier: `e` for entity, `s` for service, `ac` for atomic component and `cc` for composite component. Element's properties are placed on their own indented line and prefixed with the property name (or a short alias). Listing 1 gives an example of a service declaration.

For readability reasons, every item of a service parameters set must be written on its own line. For example line 4 of Listing 1 shows a `param` item instead of the whole `params` set.

[9] https://tools.ietf.org/html/rfc7231

**Listing 1.** Definition of a service using the concrete syntax

```
1 s
2   method POST
3   path /getName/{email}
4   param path email: String
5   ci GetName
```

The concrete syntax of the other structures of the meta-model is defined in a similar way. A full example is available in the repository of SWSG[10].

### 3.4 Evaluating a Model of Web Services

Our meta-model gives helpful abstractions to develop web services but the built models need a rigorous evaluation semantics. The following successive steps describe how web services based on an instance of this meta-model could handle incoming HTTP requests.

*Routing.* First, the application receives a HTTP request. Its list of services is sequentially scanned until a service matches the request; that is, the HTTP method is the same and the URL matches the path. If no service matches, then a static `404` HTTP response is sent back.

*Flattening.* The component instance contained in the service is reduced to a flattened ordered list of instances of atomic components. Arguments passed to the different components (through bindings in component instances) are resolved so that they become only constants (no more variables), and aliases are propagated to instantiated components and their subcomponents. Instances of composite components are then recursively replaced by their subcomponents. We define by cases a function $flatten(m, c)$ (with $m \in M$) that flattens a given component:

$$flatten(m, c) = \begin{cases} [c] & \text{if } c \in AC \\ \bigcup_{ci \in c.components} flatten(m, ci) & \text{if } c \in CC \\ flatten(m, c.component) & \text{if } c \in CI \\ flatten(m, resolve_c(m, c)) & \text{if } c \in Id \end{cases} \quad (1)$$

The function $resolve_c$ used in Formula (1) takes two parameters: a model and a component name. It outputs the definition of the componant with the given name in the given model. It is of type $M \times Id \to C$. When called on a model that verifies the consistency rules defined in Section 4 and on a component name extracted from such a model, this function returns a deterministic result.

[10] https://gitlab.startup-palace.com/research/swsg/blob/master/examples/registration/registration.model

*Evaluating components.* An initial evaluation context is created by extracting parameters (if present) from the request URL and putting them into an empty context. This flattened list is then evaluated: every atomic component is executed given the previous context as an input and produces a new context as an output. This behavior is very similar to state monads (see [29, § 2.5]).

*Responding.* Finally, a HTTP response is built. There are two cases to consider. If one of the evaluated components returned a HTTP response instead of a new context, the following components are not evaluated and this response is returned to the client. Otherwise, the context is serialized and encapsulated into a HTTP response of code `200`.

## 4 Consistency of Web Services

Section 3 showed that our web services meta-model uses components as an abstraction to improve separation of concerns and reusability. In order to allow developers to safely use this abstraction we propose a way to do verification of models. This verification checks if a model is consistent. It can happen at *design-time* – outside any evaluation context – so that inconsistent models won't be run in production.

*Definition.* A model of web services $m \in M$ is consistent if it verifies all the following properties identified by Formulas (2) to (22).

*Component name unicity.* Every component in a model has a unique name.

$$\forall c, c' \in m.components \cdot (c.name = c'.name \Rightarrow c = c') \tag{2}$$

*Entity name unicity.* Every entity in a model has a unique name.

$$\forall e, e' \in m.entities \cdot (e.name = e'.name \Rightarrow e = e') \tag{3}$$

*Attribute name unicity.* Every attribute of an entity has a unique name.

$$\forall e \in m.entities, \forall a, a' \in e.attributes \cdot (a.name = a'.name \Rightarrow a = a') \tag{4}$$

*Service parameter name unicity.* Every parameter of a service has a unique name.

$$\forall s \in m.services, \forall p, p' \in s.params \cdot (p.name = p'.name \Rightarrow p = p') \tag{5}$$

*Parameters of location `body` unicity.* There is a maximum of one parameter per service that has its location equal to `body`.

$$\forall s \in m.services, \forall p, p' \in s.params \cdot (p.location = p'.location = \texttt{body} \Rightarrow p = p') \tag{6}$$

*Reference consistency.* Table 2 describes the reference locations in a model. It gives formulas to extract from a model every possible set of references to entities or components.

**Table 2.** Exhaustive list of reference locations

| **Referenced element** | **References locations** |
|---|---|
| Component $C$ | $refs_{c1} = Prj_{components.component}(m.components \cap CC)$ |
| | $refs_{c2} = Prj_{component.component}(m.services)$ |
| Entity $E$ | $refs_{e1} = Prj_{entity}(Prj_{attributes.type}(m.entities) \cap EntityRef)$ |
| | $refs_{e2} = Prj_{entity}(Prj_{params.type}(m.components) \cap EntityRef)$ |
| | $refs_{e3} = Prj_{entity}(Prj_{pre.type}(m.components \cap AC) \cap EntityRef)$ |
| | $refs_{e4} = Prj_{entity}(Prj_{add.type}(m.components \cap AC) \cap EntityRef)$ |
| | $refs_{e5} = Prj_{entity}(Prj_{rem.type}(m.components \cap AC) \cap EntityRef)$ |
| | $refs_{e6} = Prj_{entity}(Prj_{params.variable.type}(m.services) \cap EntityRef)$ |

Every reference to a component designates an element that exists in the model.

$$\forall ref \in (refs_{c1} \cup refs_{c2}), \exists c \in m.components \cdot (c.name = ref) \tag{7}$$

Every reference to an entity designates an element that exists in the model.

$$\forall ref \in (refs_{e1} \cup refs_{e2} \cup refs_{e3} \cup refs_{e4} \cup refs_{e5} \cup refs_{e6}), \\ \exists e \in m.entities \cdot (e.name = ref) \tag{8}$$

*Component context variable name unicity.* Variables in atomic components cannot have the same name if they are not identical.

$$\forall c \in (m.components \cap AC), \forall v, v' \in (c.pre \cup c.add \cup c.del) \cdot \\ (v.name = v'.name \Rightarrow v = v') \tag{9}$$

*Composite component non emptiness.* Composite components must have subcomponents.

$$\forall c \in (m.components \cap CC) \cdot (c.components \neq \emptyset) \tag{10}$$

*Alias source unicity.* The source name of an alias is unique in its component instance.

$$\forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)), \\ \forall a, a' \in ci.alias \cdot (a.source = a'.source \Rightarrow a = a') \tag{11}$$

*Alias target unicity.* The target name of an alias is unique in its component instance.

$$\forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)), \\ \forall a, a' \in ci.alias \cdot (a.target = a'.target \Rightarrow a = a') \quad (12)$$

*Recursive reference consistency.* An entity is not referenced by its transitive attributes.

$$\forall e \in m.entities \cdot (EntityRef(e.name) \notin deps_e(m, EntityRef(e.name))) \quad (13)$$

where the function $deps_e : M \times T \to \mathcal{P}(T)$ returns the set of transitive dependencies of a given type. $deps_e(m, t)$ is defined as follows:

$$deps_e(m,t) = \begin{cases} deps_e(m,t') & \text{if } t = OptionOf(t') \\ deps_e(m,t') & \text{if } t = SeqOf(t') \\ t'' \cup \bigcup\limits_{t_i \in t''} deps_e(m,t_i) & \text{if } t = EntityRef(name) \\ deps_e(m,t) = \emptyset & \text{otherwise} \end{cases} \\ \text{with } t'' = Prj_{type}(resolve_e(m, name).attributes)$$

where the function $resolve_e : M \times Id \to E$ returns the definition of the entity that as the same name as the one given in the second parameter.

The same is true for composite components that are not referenced from their transitive subcomponents.

$$\forall c \in (m.components \cap CC) \cdot (c.name \notin deps_c(m, c)) \quad (14)$$

where the function $deps_c : M \times C \to \mathcal{P}(Id)$ returns the set of transitive dependencies of a given component. $deps_c(m, c)$ is defined as follows:

$$deps_c(m,c) = \begin{cases} \emptyset & \text{if } c \in AC \\ Prj_{name}(c.components) \cup \\ \qquad \bigcup\limits_{ci_i \in c.components} deps_c(m, resolve_c(m, ci_i)) & \text{if } c \in CC \end{cases}$$

where the function $resolve_c : M \times Id \to C$ returns the definition of the component that has the same name as the one given in the second parameter, as defined in Section 3.4.

*Alias source validity.* The source name of an alias corresponds to the name of a variable of the contract of the instantiated component.

$$\forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)), \\ \forall a \in ci.alias \cdot (c = resolve_c(m, ci.component) \Rightarrow \\ a.source \in Prj_{name}(c.pre \cup c.add \cup c.rem)) \quad (15)$$

*Alias target validity.* The target name of an alias corresponds to the name of a variable that will be added to the execution context by the instantiated component.

$$\forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)), \\ \forall a \in ci.alias \cdot \\ (c = resolve_c(m, ci.component) \Rightarrow a.target \notin Prj_{name}(c.add)) \quad (16)$$

*Service path validity.* The set of names of service parameters that have a `path` location is exactly the same as the set of parameter names declared in the path string of a service.

$$\forall s \in m.services \cdot (\{p \in s.params | p.location = \texttt{Path}\} = extract(s.path)) \quad (17)$$

where the function $extract : P \rightarrow \mathcal{P}(V)$ extracts parameter names from a path; that is, for a given path, it returns the contents of the first matching parenthesis of the regular expression `\{([A-Za-z0-9_]+)\}`.

*Component context immutability.* Components don't override existing variables of the context. Every atomic component does not add a new variable to its output context if there is already a variable with the same name in its input context.

$$\forall c \in (m.components \cap AC) \cdot (c.add \cap c.pre = \emptyset) \quad (18)$$

*Component precondition exhaustivity.* Components depend on the variables they remove. Every atomic component has each variable it will remove from the context in its preconditions.

$$\forall c \in (m.components \cap AC) \cdot (c.rem \subseteq c.pre) \quad (19)$$

*Component instance bindings consistency.* Component instances have bindings that associate a term to a variable. Every binding associates a term to a variable of the same type.

$$\forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)), \\ \forall b \in ci.binding \cdot (b.variable.type = b.argument.type) \quad (20)$$

*Component instance parameters exhaustivity.* Component instances provide values for every parameter of the instantiated component. Every component instance provides exactly as much arguments as the component it instantiates needs parameters. Names and types of the arguments match those of the parameters.

$$\forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)), \\ \exists c \in m.components \cdot \\ (c.name = ci.component \Rightarrow Prj_{param}(ci.binding) = c.params) \quad (21)$$

*Context validity.* Components are instantiated in contexts that fulfill their preconditions. When building flat ordered lists of atomic components for each service (see Section 3.4), every atomic component of these lists has its preconditions fulfilled by its input context:

$$\forall s \in S \cdot (flatten(s.component) \blacktriangleleft s.params) \tag{22}$$

where $\blacktriangleleft$ is a function of $List(AC) \times \mathcal{P}(V) \rightarrow Boolean$ in infix notation[11] that is true when applied to a component and a context that satisfies the component's preconditions. It is defined by the following semantic rules:

$$\frac{ctx_0 \in \mathcal{P}(V)}{[\,] \blacktriangleleft ctx_0}$$

$$\frac{\begin{array}{ccc} ctx_0 \in \mathcal{P}(V) & & ctx_1 = ctx_0 \cup c_0.add \setminus c_0.rem \\ \forall i \in [0,n], c_i \in CI & c_0.pre \subseteq_\triangleleft ctx_0 & [c_1, \ldots, c_n] \blacktriangleleft ctx_1 \end{array}}{[c_0, \ldots, c_n] \blacktriangleleft ctx_0}$$

$$\frac{\begin{array}{c} pre, ctx \in \mathcal{P}(V) \\ \forall v \in pre \cdot v \in ctx \vee (\exists v' \in ctx, v.name = v'.name \\ \wedge v'.type = OptionOf(v.type)) \end{array}}{pre \subseteq_\triangleleft ctx}$$

$pre \subseteq_\triangleleft ctx$ means that the component's preconditions $pre$ are satisfied by the context $ctx$. We use $\subseteq_\triangleleft$ instead of $\subseteq$ because $\subseteq$ requires the types to be strictly identical, which we do not want in order to handle optional types.

## 5 Development Process Integrated to OpenAPI

In Section 3 we introduced a meta-model of web services, along with rules to check its consistency in Section 4. In Section 5.1 we introduce a common development process involving OpenAPI. This gives some background on OpenAPI and prepares to the merger of our approach in OpenAPI's in Section 5.2.

### 5.1 A Common Usage of OpenAPI

The OpenAPI Specification [11] 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" [11]: that is a meta-model. As in MDE, the point of having meta-models is to have tools that can rely on them in order to safely manipulate models and offer support to developers. Indeed, an ecosystem of tools was developed around OpenAPI by

[11] $cs \blacktriangleleft ctx$ is equivalent to $\blacktriangleleft (cs, ctx)$

various actors. Such tools have several purposes, including but not limited to: providing an interactive graphical user interface from a model [23, 24], generating functional tests from a model [1] or generating a model from an annotated implementation [28].

*The Petstore example.* To ease the development of tools, some official examples of OpenAPI models are shipped with the specification. In this article, we focus on the *Petstore* example [12]. It describes a simple application that exposes four services to list, show, add and remove data records representing animals. We assume that we are in the context of a company such as *Startup Palace*; that means the goal is to develop these web services in order for them to be consumed by user interface applications; for example, desktop and mobile applications for the owner of the store.

*Development process.* The common top-down development process follows. First, developers make several iterations on writing an OpenAPI model. This model must match the functional specifications and describe web services that are fully exploitable by consumer applications. For example, the description of one of the Petstore services is shown in Listing 2. It references schemas that are defined in another part of the model, as shown in Listing 3.

**Listing 2.** A service in the Petstore example

```
/pets/{id}:
  get:
    description: Returns a user based on a
        single ID, if the user does not have
        access to the pet
    operationId: find pet by id
    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'
      default:
        description: unexpected error
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/
                  Error'
```

**Listing 3.** Schemas in the Petstore example

```
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
    Error:
      required:
        - code
        - message
      properties:
        code:
          type: integer
          format: int32
        message:
          type: string
```

When this OpenAPI model is stable enough, developers can use it as a specification to start building web services and consumer applications. There are some kinds of tools that can take the OpenAPI model as input and help to build compliant web services; for example, by generating a skeleton of application using a given technological stack [10], or by generating automated tests that can be used to check if the web services are conformant [1].

But all these tools have a major limitation: they require humans to manually update the OpenAPI model whenever the web services evolve. This is a very common situation: specifications must evolve either because business requirements have changed or new constraints have been discovered while developing. Even if some tools can mitigate this issue, it is likely that developers will eventually stop maintaining the OpenAPI model after the web services reach production, making the two diverge over time. In long-term projects, this means giving up on every advantage provided by OpenAPI and its MDE approach.

To fix this shortcoming, we present in Section 6 an improved version of this process that leverages a meta-model of web services we introduced in [21]. However, in order for this new approach to be feasible, we first need to extend the OpenAPI 3.0 Specification.

### 5.2 Extending OpenAPI 3.0

Recall that the OpenAPI Specification describes a meta-model to express an interface to web services. But, unlike our approach, it does not describe how web services are implemented. To get the best of both our approach and OpenAPI's, we propose a way to merge the latter with our meta-model of web services.

To preserve tools compatibility we make use of *Specification Extensions*, as defined in OpenAPI 3.0 [11]. This mechanism allows to add data to models without breaking their compliance to the specification or their ability to be used by tools designed to be compatible with it. The details of the extensions to OpenAPI 3.0 can be found in [20]. The following paragraphs present the nature and the main lines of these extensions for each aspect of our meta-model.

Because an OpenAPI model can be seen as a tree (before references are resolved) that has the `OpenAPI` object as root, we make use of the notion of *paths*. The `components` path designate the child of the root named `components`. Sub-children are separated using a `>` symbol; the `components > requestBodies` designates the `requestBodies` child of the `components` child of the root.

*Data model.* An OpenAPI 3.0 model can contain a set of `Schema` objects[12]. A `Schema` object defines a data type for input or output data, which can then be referenced from elsewhere in the model. This mechanism provides more expressiveness than ours and was made for the same purpose. Thus it is a good fit for the *entities* of our meta-model.

[12] In the `components > schemas` path.

*Processes.* On purpose, there are no equivalent of our component system in OpenAPI 3.0, because it describes processes that are internal to the web services which is out of OpenAPI's scope. Accordingly we add two properties in the `Components` object of OpenAPI. They contain sets of atomic and composite component definitions[13]. Exact schemas of these components follow on from their definition in our meta-model.

*Services.* Describing services is the main feature of OpenAPI. As such they can be specified in a quite expressive way. Yet the service meta-model in OpenAPI is not a superset of ours because of two lacks that require it to be extended. First, each service must be associated to a component instance that describes its behavior[14]. Second, if a service has a `requestBody` property (that describes the type of the data required in the request body), the `RequestBody` object must contain a variable name[15]. When generating the web services code, this is used to include the request body contents in a variable of the execution context.

From now on, we consider OpenAPI 3.0 extended with the process and service parts of our meta-model. OpenAPI's data model is not extended as it was already powerful enough for our needs.

## 6 A Tool to Generate Consistent Web Services

To support the approach of automatically building web services from an extended OpenAPI model (see Section 5), we propose a tool named *Safe Web Services Generator* (SWSG) [19] that automates both consistency verification (see Section 4) and code generation. This tool takes two inputs: a model file (both extended OpenAPI models and our own concrete syntax are supported), and a path to a directory which contains implementations of atomic components.

As shown in Figure 3, our tool follows four sequential steps:

1. **Model parsing** Input model is parsed as concrete syntax of the meta-model (see Section 3.3) or as an extended OpenAPI model (see [20]).
2. **Model transformation** If the parsed model was an OpenAPI model, it is transformed to match our meta-model.
3. **Model consistency verification** The model is checked in order to establish its consistency (see Section 4).
4. **Code generation** The model and the implementations of atomic components are used to generate an implementation of the web services they represent.

[13] Their paths are `components > x-swsg-ac` and `components > x-swsg-cc`.
[14] In the `x-swsg-ci` property of the service.
[15] In a `x-swsg-name` property.

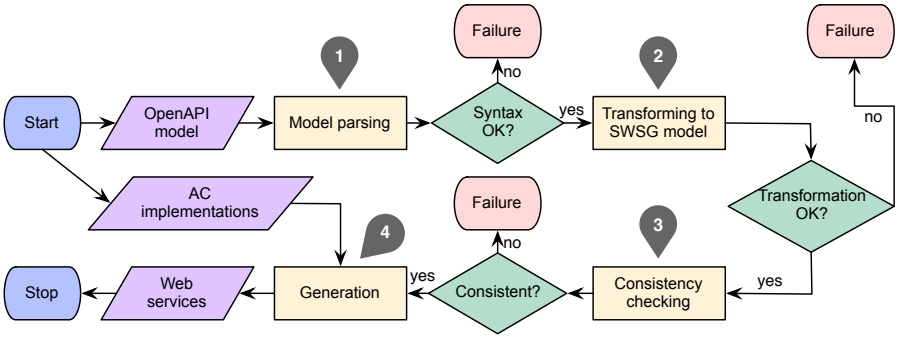


**Fig. 3.** SWSG process

*Model transformation.* Transforming extended OpenAPI models to SWSG models is quite straightforward, except for schemas/types. OpenAPI defines some primitive types and relies on a modified version of the *JSON Schema Specification* [9] for complex types. There are two issues; first, the *JSON Schema Specification* is more expressive than SWSG's type system. For example, it allows to define refined types, e.g. to add a minimum length to a string. Second, it supports both literal and referenced definition of attribute types. Moreover, references are quite expressive and can target many places in the OpenAPI main document or even in another one. In comparison, SWSG only supports literal primitive types or references to other *entities*.

While the second issue is more a technical problem, the first would require to alter SWSG's type system in order for it to support expressing every possible OpenAPI type. Yet, for the sake of simplicity of both our meta-model and our prototype we choose to not address them; they are not essential to test and validate our approach. Therefore our prototype might return errors when working with some OpenAPI models that contain these unsupported types or references in schemas.

*Code generation.* The process defined by Figure 3 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, our prototype targets the PHP programming language [25] with the *Laravel* web framework [13], which is a common tool stack.

Many similar tools (see Section 2) take the approach to generate and output a standalone web application that includes everything necessary to operate it. We took another approach by generating code that should never be manually edited; the generated code does not override any existing files in a *Laravel*'s architecture and can be easily hooked to an existing web application through configuration.

Because of the considered trade-off between provided support and flexibility left to developers, this MDE approach was designed to allow shallow consistency verification and most inconsistencies in the model are caught at compile-time. This does not prevent developers to create flawed or insecure applications, as they have full control on the atomic components implementations. Indeed this flexibility comes at the cost of a bit of support. However, we believe this trade-off is crucial when the developers have to quickly build web services that might grow and stay in production for a while.

## 7 Experimentations and Discussion

*Process.* We derive a new process from the common OpenAPI process explained in Section 5.1. *Step 1:* design a stable OpenAPI model. *Step 2:* design SWSG components using the extensions we introduced in Section 5 to write them inside the OpenAPI model. Every atomic component defined in the model must be provided with an implementation. *Step 3:* use SWSG to check the model and generate working web services if the verification is successful.

*Case study #1. Step 1:* we take the Petstore example [12] presented in Section 5.1. The example service defined by Listing 2 is a part of a standard OpenAPI model. *Step 2:* we need a component that will handle the response generation when this service will receive requests. We create a composite component called `FindPet` and reference it from the service, as shown in lines 26-27 in Listing 4. This composite component has two children that are atomic components. The first takes an ID as input, uses it to query the database and adds the `Pet` result to the context. The second takes a `Pet`, serializes it in JSON and put it in an HTTP response. These three components are defined in Listing 5. Implementations are written for every atomic components. Listing 6 shows the implementation of the `GetPetById` component as an example[16].

*Step 3:* we run SWSG on these inputs and get a *PreconditionError*. This verification error indicates that a component's precondition is not fulfilled in a given instanciation context. In the current case, we learn that the `GetPetById` component misses a string named `id` when instanciated by the `FindPet` component in the `GET /pet/{id}` service. Indeed, we voluntarily introduced an error in Listing 5: the `GetPetById` component is given an integer (by the service) whereas it requires a string. In this particular example, it should require an integer `id` variable in order to be consistent with the service parameter. Nevertheless, in more complex projects, this component might have been used inside several other composite components and services. Thus, it might not be a good solution to just change the component's definition because it might break other

[16] The PHP class in Listing 6 depends on the `Component` interface and on the `Ctx` and `Params` classes. They are defined in code output by the code generator and are just implementation details of the SWSG specification in this specific code generator. Different code generators could require different constraints on implementations of atomic components.

**Listing 4.** A service in the SWSG Petstore example

```
/pets/{id}:
  get:
    description: Returns a user based on a single ID, if the user does not have access to the pet
    operationId: find pet by id
    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'
      default:
        description: unexpected error
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Error'
    x-swsg-ci:
      component: FindPet
```

**Listing 5.** Components in the SWSG Petstore example

```
components:
  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
```

workflows. This is the kind of mistakes SWSG can prevent us to make: because they are reported very early at compile-time, instead of runtime which is too late. Developers can study the problem and decide if they have to build a better implementation or if the process model was badly designed.

When running SWSG on a fixed model, the code generation can proceed. The *Laravel* code generator generates four kinds of files: one file per atomic components (identical to those written manually by the developers; see Listing 6), one file per composite components, a router file and several static files (that do not depend on the model; for example the `Component` interface definition).

*Case study #2.* After it was successfuly implemented, we want to extend the Petstore example and add a new service to handle `PUT /pets/{id}` requests. These requests set a pet with the given attributes and the given ID, by creating it or by updating the attributes of an already exisiting pet that has the same given ID. This is very similair to the `POST /pets` service; the difference is that the latter does not receive any ID, thus it only creates pets, whereas the new service can be used to edit them as well.

To implement this, we follow the same process as earlier. *Step 1:* we extend the OpenAPI model by adding a new service that matches our needs. *Step 2:* we need to attach it to SWSG components. Because of the similarity between this service and the `POST /pets` one, we choose to reuse and extend

**Listing 6.** Implementation of the `GetPetById` atomic component

```php
<?php
namespace App\Components;
use App\SWSG\Component;
use App\SWSG\Ctx;
use App\SWSG\Params;
use DB;

class GetPetById implements Component
{
    public static function execute(Params $params, Ctx $ctx)
    {
        $pet = DB::table('pet')
            ->where('id', $ctx->get('id'))
            ->first();
        $ctx->add('pet', $pet);
        return $ctx;
    }
}
```

existing components. The `CreatePet` atomic component is thereby renamed to `CreateOrUpdatePet`, given a boolean parameter `createOnly` and a new precondition on an optional integer `id` variable. When instanciated for the `POST` service, `createOnly` is given the value `true`, whereas it is `false` when the same component is instanciated from the `PUT` service. Then, the implementation of this atomic component is modified so that it adds or updates pets depending on the value of the `createOnly` parameter and the one of the `id` context variable. Listing 7 shows how the new service instantiates a composite component (defined in Listing 8) that in turn calls the `CreateOrUpdatePet` and passes it the right value for its `createOnly` parameter. *Step 3:* we run SWSG.

**Listing 7.** Extract of the new service

```yaml
/pets/{id}:
  put:
    ...
    x-swsg-ci:
      component: AddOrEditPet
      bindings:
        - param:
            name: addOnly
            type: Boolean
          argument:
            type: Boolean
            value: false
```

**Listing 8.** The `AddOrEditPet` component

```yaml
x-swsg-cc:
  - name: AddOrEditPet
    params:
      - name: addOnly
        type: Boolean
    components:
      - component: CreateOrUpdatePet
        bindings:
          - param:
              name: createOnly
              type: Boolean
            argument:
              name: addOnly
              type: Boolean
      - component: RenderPet
```

*Discussion.* Our approach has several advantages over a regular development process, for example writing the whole application using a programming language and a web framework such as Laravel. First, the model-driven approach forces developers to think of their design or design evolutions at a macroscopic scale before writing low-level implementations. Along with the automatic model verification, this provides them an easy way to spot design mistakes early-on in the process, therefore saving some of their time. In a regular process, developers could only rely on the quality tools offered by the language or the framework. Because there is no static verification step in PHP, for example, and because lots of developers do not write any automated tests, they would often be tempted to skip checking the consistency of their design, especially when making small evolutions on it. Second, because the extended OpenAPI model is used to generate both implementation and documentation, both will stay aligned during the life of the project. In a regular process, this property relies on the will of the developers; they might stop maintaining the OpenAPI model and still make evolutions to the implementation.

As illustrated by the second step of our case study, these advantages are especially valuable for incremental development. Indeed they enforce several forms of consistency at different levels of the projects, for a low cost in term of flexibility and productivity. In contexts such as ours where we need to build and evolve web services for MVPs, incremental development is crucial. *The model and code of this case study are available in the repository of SWSG*[17].

## 8 Conclusion

We proposed a method integrated to OpenAPI 3.0 to build web services. It is fast, simple, robust and flexible. It is based on a meta-model that allows developers to define implementations of web services, starting from the corresponding high-level contract as expressed by a standard OpenAPI model. Consistency of models can be verified using an operational semantics so that code generated from these models is safe. We built a tool, SWSG, that leverages this process in the technological context of a web company, *Startup Palace*. The whole approach was illustrated on a two-steps case study to show its advantages. Even if one of the motivations was to develop MVPs applications, the approach is not limited to this scope and is suitable to most applications based on web services.

We have several main prospects. First, the type system used to describe component parameters, preconditions and model's entities (among others) is at the core of the consistency verification, yet it is not flexible enough. Making it more expressive, by allowing subtyping in component preconditions for example, while keeping at least the same level of verification might be necessary to reach a good reusability on bigger projects. Another perspective is to allow and ease safe model composition, so that developers can reuse concepts between projects when it makes sense. Model composition is theoretically handled by OpenAPI

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

but not currently supported by SWSG. Then, the developper experience could be improved if there were tools able to automatically check the compliance of atomic components to their contract in the model. Finally, the whole approach needs to be evaluated on more realistic and larger case studies. This evaluation must rely on metrics that have a good correlation with the benefits of our approach, such as easing new developers to onboard on projects and reuse of exisiting code. This may take several months of practice and reviews.

## References

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

[2] Mario Luca Bernardi, Marta Cimitile, and Fabrizio Maria Maggi. "Automated Development of Constraint-Driven Web Applications". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 1196–1203.

[3] Mario Luca Bernardi et al. "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, pp. 73–80.

[4] Hanyang Cao, Jean-Rémy Falleri, and Xavier Blanc. "Automated Generation of REST API Specification from Plain HTML Documentation". In: *International Conference on Service-Oriented Computing*. Springer, 2017, pp. 453–461.

[5] Marco Cremaschi and Flavio De Paoli. "Toward Automatic Semantic API Descriptions to Support Services Composition". In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2017, pp. 159–167.

[6] Frank DeRemer and Hans Kron. "Programming-in-the Large versus Programming-in-the-Small". In: *ACM Sigplan Notices*. Vol. 10. ACM, 1975, pp. 114–121.

[7] Xiang Fu, Tevfik Bultan, and Jianwen Su. "Analysis of Interacting BPEL Web Services". In: *In Proc. 13th Int. World Wide Web Conf.* Citeseer, 2004.

[8] Roy Gronmo et al. "Model-Driven Web Services Development". In: *E-Technology, e-Commerce and e-Service*. EEE'04. IEEE, 2004, pp. 42–45.

[9] Internet Engineering Task Force. *JSON Schema: A Media Type for Describing JSON Documents*. Oct. 13, 2016. URL: https://tools.ietf.org/html/draft-wright-json-schema-00.

[10] Paulo Lopes and Francesco Guardiani. *Slush-Vertx*. 2017. URL: https://github.com/pmlopes/slush-vertx.

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

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

[13] Taylor Otwel. *Laravel*. 2016. URL: https://laravel.com/.

[14] Jack Pugaczewski et al. "Software Engineering Methodology for Development of APIs for Network Management Using the MEF LSO Framework". In: *IEEE Communications Standards* 1.1 (2017), pp. 92–96.

[15] RAML Workgroup. *RAML*. 2016. URL: https://raml.org/.

[16] Jérôme Rocheteau and David Sferruzza. "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.

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

[18] Simon Schwichtenberg, Christian Gerth, and Gregor Engels. "From Open API to Semantic Specifications and Code Adapters". In: *Web Services (ICWS), 2017 IEEE International Conference On*. IEEE, 2017, pp. 484–491.

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

[20] David Sferruzza. *Specification of SWSG Extensions for OpenAPI*. 2018. URL: https://gitlab.startup-palace.com/research/swsg/tree/master/openapi-extensions-specification/1.0.0.md.

[21] David Sferruzza et al. "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. URL: https://hal.archives-ouvertes.fr/hal-01654287.

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

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

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

[25] The PHP Group. *PHP*. 2016. URL: https://php.net/.

[26] Romanos Tsouroplis et al. "Community-Based API Builder to Manage APIs and Their Connections with Cloud-Based Services." In: *CAiSE Forum*. 2015, pp. 17–23.

[27] Wil M.P. van der Aalst, Maja Pesic, and Helen Schonenberg. "Declarative Workflows: Balancing between Flexibility and Support". In: *Computer Science-Research and Development* 23.2 (2009), pp. 99–113.

[28] Martijn van der Lee. *PHPSwaggerGen*. 2017. URL: https://github.com/vanderlee/PHPSwaggerGen.

[29] Philip Wadler. "The Essence of Functional Programming". In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1992, pp. 1–14.

[30] Egon Willighagen and Jonathan Mélius. "Automatic OpenAPI to Bio.Tools Conversion". In: *bioRxiv* (2017). DOI: 10.1101/170274.