Electronic Communications of the EASST
Volume 82 (2022)

11th International Symposium
on Leveraging Applications of Formal Methods, Verification
and Validation
-
Doctoral Symposium, 2022

DSL-driven Integration of OpenAPI based Web Services into DIME

Bruno Steffen

15 pages

Guest Editors: Sven Jörges, Salim Saay, Steven Smyth
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

DSL-driven Integration of OpenAPI based Web Services into DIME

Bruno Steffen

Technische Universität Dortmund

Abstract: Since nowadays the vast majority of web developers integrate third party web services into their web applications, communities formed to standardize the API landscape. This resulted in formats such as the OpenAPI Specification (OAS)[1], GraphQL[2], RAML[3] and many more. The communities further strengthen their efforts by providing tools that assist web developers when working with third party web services. However, these tools are typically aimed at traditional code environments. This paper discusses an approach to make use of these formats, primarily the OAS, to automatically integrate web services into the low-code development environment called DIME. This is done by mapping the machine readable OAS files to domain-specific languages that in turn generate code for the final DIME application.

Keywords: Domain specific languages, API languages, Visual languages

1 Introduction

Over the years the overall usage of web services accessed via APIs has steadily increased, with some sources claiming that more than 94% of software developers are using third party web services [Pol21]. These services can range from ad services, to video integration or for social features and marketing. This means that there are many good reasons for web developers to use these services which most of the time are accessed via public APIs.

This led to the emerging of communities that focus on standardizing the representation of APIs to ultimately simplify their design, development and usage. There are many standards that arose from those efforts, such as the OpenAPI Specification, RAML, WADL, GraphQL and many more. With these standardized representations available, web developers have great assistance when using third party APIs, especially given that the communities develop large sets of tools that make use of their respective standard to further improve productivity.

However, these tools typically focus on assisting web developers by generating code that can then be pasted into their projects. Hence, these tools are rendered useless once developers work with low-code, or no-code environments where the simple copying and pasting of code is not only unwanted but sometimes even impossible.

One of these environments is DIME, an integrated modeling environment (IME) for web applications which is developed at the TU-Dortmund university. The idea behind this IME is to enable users to create full-stack web applications by means of graphical models rather than actual coding. This means that, given the context of web development, the integration of third party

[1] OpenAPI is an API specification maintained by the OpenAPI Initiative (https://www.openapis.org/what-is-openapi)
[2] GraphQL is a query language for APIs (https://graphql.org/learn/)
[3] RAML is a specification language for APIs (https://raml.org/)

web services into DIME applications is desired by most users. However, the tools provided by the formerly mentioned communities will not work, as the simple pasting of code is not possible.

A first attempt to solve the issue of integrating web services into DIME was made in early 2022 with the development of the HTTP-DSL [Ste22]. The HTTP-DSL offers an easy-to-learn textual configuration of HTTP requests, that can then be integrated and used within DIME with the click of a button. However, this approach leaves room for improvement, since it still requires a manual integration of web services and makes no use of the plethora of standardized API specifications that are implemented by many web services around the world.

The idea is to use the HTTP-DSL as a foundational layer to automate the process of web service integration into DIME. We can use the DSL that is specific to the HTTP domain to profit from an easy development of the automation and the already existing code generator that ultimately provides the production code.

However, the automation of web service integration can only work if the respective APIs are machine-readable, which traditionally was not the case. To counter this, a big push by communities such as the OpenAPI Initiative[4] moved companies to adopt their machine-readable API specification format. Large corporations such as Paypal, Twitter, and Netflix already use this specification format, with databases such as apis.guru[5] providing the specification files of thousands of APIs.

With the machine-readable API formats available, we want to design and implement automated web service integration for DIME. In this work we focus on using the OAS, however, this does not mean that the other formats are less useful. Our solution is meant to provide a way to include the data schema and all available operations of a web service given its specification file.

How this solution works will be discussed in the coming sections, beginning with Section 2 where the required technologies are introduced. Then, the core idea for the implementation of the solution is detailed in Section 3, followed by the conclusion of the paper in Section 4.

2 Preliminaries

In this section, the three fundamental technologies for this paper are described in detail. Beginning with the OpenAPI Specification in Subsection 2.1, followed by the DIME framework in Subsection 2.2, and finally the HTTP-DSL in Subsection 2.3.

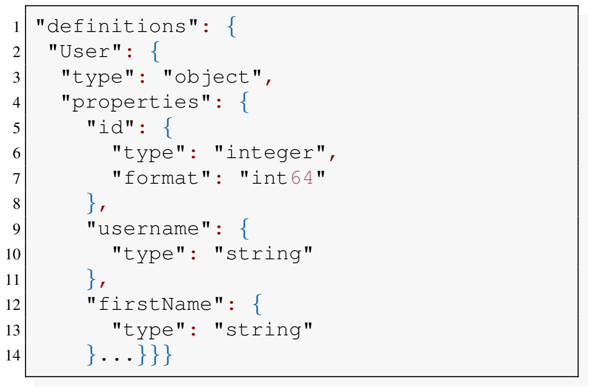
2.1 OpenAPI Specification

The OpenAPI Initiative is an open source community, that collaborates with a number of industry partners to develop vendor-neutral open-source specifications for APIs [Fou22]. The most well known of these specifications is the OpenAPI Specification (OAS), a machine-readable specification (in JSON or YAML format) for RESTful [Fie00] web services. There are various tools available that produce, describe, or visualize APIs with the help of specification files, and tools that read APIs (e.g. by embedding code into server code) to generate specification files[6].

[4] The official website of the OpenAPI Initiative can be found at https://www.openapis.org/

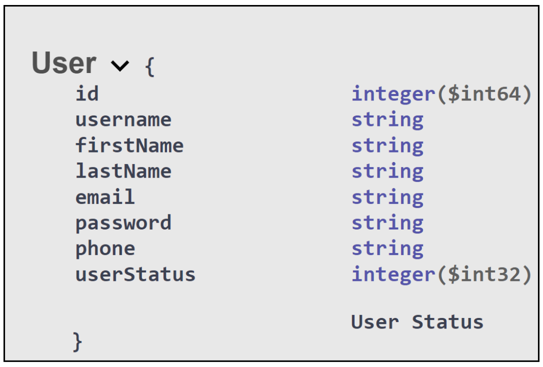
[5] Apis.guru is a website that provides a catalog of APIs and their OpenAPI specification files, see https://apis.guru/

[6] A full list of available tools can be found at https://openapi.tools/



```
"definitions": {
 "User": {
  "type": "object",
  "properties": {
    "id": {
      "type": "integer",
      "format": "int64"
    },
    "username": {
      "type": "string"
    },
    "firstName": {
      "type": "string"
    }...}}}
```

(a) Representation in JSON format



```
User ⌄ {
   id                    integer($int64)
   username              string
   firstName             string
   lastName              string
   email                 string
   password              string
   phone                 string
   userStatus            integer($int32)

                         User Status
   }
```

(b) Representation provided by Swagger's UI tool.

Figure 1: Different representations of the user object taken from the pet store example.

The key idea behind OAS is to have a unified standard for the representation of APIs, which enhances the ability to use and connect different web services altogether. To make the product as transparent as possible, OpenAPI provides a public Github repository, that includes the source code and a guide on how to use the specification. [7]

An example of such a specification can be seen in Figure 1a, which is an excerpt from an exemplary API provided by Swagger[8]. The code represents a `User` object in JSON format. We can see that it has multiple properties, however, only `id`, `username`, and `firstName` are depicted to keep the image to an appropriate size.

The same object is shown in Figure 1b, which is a visual representation of the previously shown source code. The image is a screenshot from Swagger's website[9] where the *Swagger UI* tool is presented. This tool can provide full visualization of the API, given the OAS file.

2.2 DIME

DIME [BFK+16] is an integrated modeling environment that provides a number of Domain Specific Languages (DSLs) that are used to create a web application.

The goal of DIME is to minimize the amount of misunderstanding (mostly caused by error prone communication between software developers and domain experts) by enabling domain experts to develop a web application by themselves. This is possible because DIME is a low-code environment that provides DSLs that are easy to use and understand, even for non-programmers. Additionally, DIME solves many of the typical concerns for web applications out of the box, such as security or the interplay between frontend and backend. Hence, DIME users can channel their focus on the unique characteristics of their application, rather than the components that are

[7] A detailed description of the specification (version 3.0.3) can be found on OpenAPI's Github page: https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.0.3.md

[8] Swagger is a company that provides a set of tools that synergize with OAS. The complete JSON for the Swagger pet store example can be found at https://petstore.swagger.io/v2/swagger.json

[9] The website with the pet store example can be found at https://petstore.swagger.io/

required to create, run and deploy web applications.

DIME follows the Single Source of Truth paradigm [PS14], meaning that replication is minimized because all versions of an object stem from the same source. In the case of DIME the artifact is the code for a web application and the single source of truth is the collection of DSL code for a DIME application. Application code is fully generated from the DSLs and never funneled back into DIME, thus mitigating any round-trip problems that typically occur when using DSLs or other abstract models of code. Round trip engineering is not mandatory when working with DIME, because the expressiveness of the DSLs is detailed enough to be able to code and configure all the important components for a web application. Any kind of fix or upgrade to the code is done by enhancing DSL code and re-generating the application.

The three most significant DSLs are graphical DSLs that are used to develop the data schema, the business logic, and the user interface (UI) for web applications. The respective models of these DSLs, among other things such as operations, are wrapped in Service Independent Building Blocks (SIBs), which offer a uniform interface to connect all components within DIME. However, to fully understand DIME, a detailed description of the respective DSLs is given.

Data The *Data Language* specifies a domain model that consists of data objects. These data objects are made up of attributes, such as typical data types such as `Integers`, `Strings` and `Characters`, or `arrays` and `lists` of these types. However, these attributes can also be more complex, including abstract types, types with inheritance, associations, or bidirectional associations between objects. As can be seen in Figure 5b, the model resembles a UML class diagram [Gro17], which is easy to learn and known to most programmers.

Business Logic The *Process Language* is used to create so-called Process Models which define how the web application works. The main mechanic behind this type of model is a process flow by the connection of the previously mentioned SIBs. These SIBs are reusable items representing data types, simple operations (e.g. the transformation of an `Integer` to a `String`), or even other processes that themselves have been modeled using process models. This allows for a logical layering of processes, enabling the DIME user to create complex workflows for their application.
Furthermore, the process model has data-flow capabilities, that are managed in combination with the process flow. The data-flow graph is used to handle how data is stored, altered, or transferred between SIBs.

User Interface In order to specify the user interface of the web application, DIME offers dedicated *GUI models*. These models represent the structure of the web application and consist of components such as `buttons`, `text`, and `forms` that can be found in almost every web application. The components can be customized in terms of layout and behavior. This is done with data-binding (to data types) and `if` and `for` edges, that introduce logic to the UI.

For the scope of this paper, the two relevant languages are the Process Language and the Data Language. To understand how they are operated and how the dynamic between the two plays out, we present an example which is based on the pet store toy example provided by Swagger. Looking at Figure 5b, we see how objects look in DIME, in this case a `Pet` and a `Category` object. While the really detailed discussion of this example takes place in Subsection 3.2, we can now take a look at the structure of the language. The objects hold attributes that are either

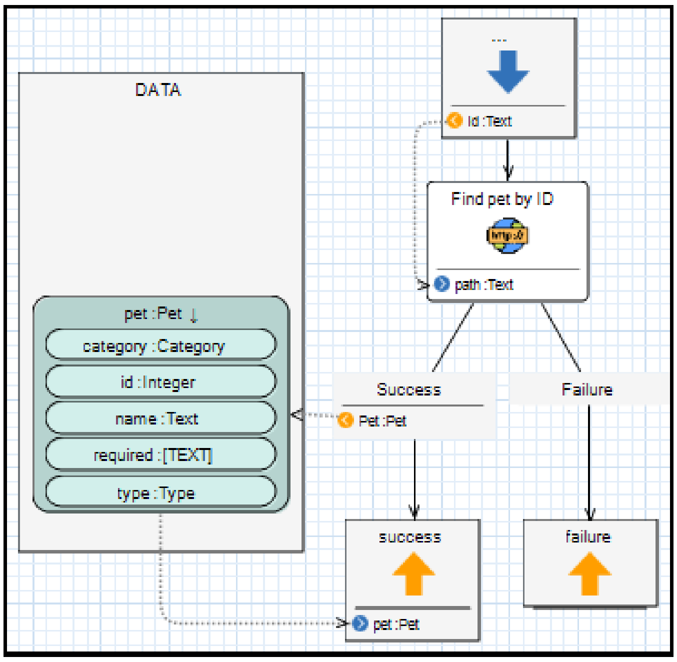


Figure 2: DIME Process Model for retrieving a pet object from the pet store API provided by Swagger.

primitive data types, or a reference to other objects. In this scenario, the `Pet` object has a *category* attribute, which is referencing the `Category` object as a data type. Furthermore, the Data Language allows for enums, such as the `Type` enum in this example. Once an object is defined in the Data Language, it can be used in the Process Language. An exemplary process is depicted in Figure 2. Each process begins at the start node (blue arrow) in the top right, and ends in an end node (orange arrow). Following the solid arrows, we can can see the process flow going from node to node. Each of these nodes is a SIB, which can either hold simple operations, whole other processes or in this case the *Find pet by ID* SIB, which holds the operation of calling the pet store API and fetching a pet object. Further following the solid arrows we notice that the SIB can either succeed or fail. If retrieving the pet fails, the process ends in failure. In the case of a success, the SIB provides a pet object to the process and the process will end in the success node.

However, Process Models not only provide a process flow, but also a data flow. The data flow is signified by the dotted arrows and the *data context* that is represented by the big grey box on the left hand side of the model. The *data context* serves as the transition from the Data Language to the Process Language. The DIME user can instantiate the objects that were previously defined in the Data Language and store or retrieve either whole objects or their attributes. Going back to the example, the *data context* is used to store the pet object (dotted arrow from the success branch) that was retrieved from the *Find pet by ID* SIB. The pet object is then passed along to the end node, meaning that it will be returned by the overall process.

2.3 HTTP-DSL

The HTTP-DSL is a textual domain specific language that is designed for quick and easy configuration of HTTP messages. In the context of DIME, the HTTP-DSL provides "an increased velocity of development regarding HTTP requests" [Ste22], which is beneficial when trying to connect a DIME application with other web services.

The traditional way of accessing APIs within DIME is to extend the *native SIBs* library. Out of the box, native SIBs cover simple operations such as String concatenation, however, DIME allows users to create custom native SIBs that can then be used in Process Models. In order to do so, the DIME user first has to write a Java method that executes the desired behaviour, followed by a few lines of code in a DSL to define how the SIB should be named and what kind of inputs and outputs are required. While this process is quick and easy to do for basic operations (especially if the DIME user has already worked with Java), it is tedious and error prone when developing more complex workflows. If the user wants to send HTTP requests, he would have to find a viable Java library for that purpose, followed by an implementation and manual bug fixing.

In essence, the HTTP-DSL is designed to mitigate all these problems. It allows for a configuration using a HTTP-tailored DSL, that is then transformed into a SIB, which in turn can be used within the Process Models. No Java code has to be manually written since the code generator for the HTTP-DSL is automatically triggered when the web application is generated from the DIME model. The generated code makes use of the Apache HTTP Client library[10] to send HTTP requests and to manage responses.

Furthermore, the generated Java code handles error codes, redirects and many other HTTP-specific workflows. However, since sometimes details of this kind need to be handled in a specific manner, the DSL allows for customization when necessary. What kind of customization is possible and how certain problems are addressed is beyond the scope of this work and can be read in the thesis [Ste22] that initially introduced the DSL.

2.3.1 The Structure

The structure of the HTTP-DSL can be best explained using an example. In Figure 3 we can see how an exemplary HTTP request is configured using the HTTP-DSL. The given HTTP request is used to access a pet object from a pet store API.

Before the actual request is defined, one can import DIME data models, to reference the domain objects within, as can be seen in line 1. Each request is encapsulated by `http{ }`, seen in the third and the twelfth line. The HTTP request then requires a unique name, since the generated SIB inherits the name and should be easy to identify. The `url` and `path` parameters specify the location of the requested source. In this specific case, the path includes the ID of the pet, meaning that it has to be dynamically changed based on the pet object we want to retrieve. Hence, a variable is used, denoted by the leading `$` sign. These variables are inserted during runtime of the web application, and the input is controlled in the process model (e.g. data that was fetched prior in the process is inserted dynamically).

[10] The documentation and an overview for the Apache HTTP Client library is given at https://hc.apache.org/httpcomponents-client-5.2.x/

```
1 import "dime-models/app.data" as data
2
3 http{
4   name "Find pet by ID"
5   url petstore.swagger.io path $path
6   type GET
7   response type application/json
8        return object data#Pet
9 }
```

Figure 3: HTTP request to retrieve a pet object from a pet store API by ID, using the HTTP-DSL.

The `type` parameter is used to specify the HTTP type of the HTTP request, in this example, it is set to `GET` which means that we want to retrieve data from the source (compared to e.g. `POST` which is used to send data). The last two lines of code, namely lines seven and eight, are used to specify the response for the request. In this case, the response is a JSON object and it is mapped to a Pet object that is defined in the data model of the DIME application.

2.3.2 The Workflow

Going back to the example that was already discussed earlier, we can now explain the typical workflow that a DIME user would go through when integrating the pet store API into their web application. Below is the list of steps that the DIME user has to go through to include a HTTP request (in third example using the *find pet by ID* operation) into their Process Model:

1. Find the correct operation.
2. Find out which data types are required for the parameters of the operation and what it returns.
3. Define the proper data types using the Data Language, as shown in Figure 5b.
4. Define the HTTP request, as portrayed in Figure 3.
5. Save the DSL file, which automatically generates a SIB.
6. Include the SIB into the Process Model, as depicted in Figure 2.

While this workflow is a lot quicker and easier than the traditional workflow without the HTTP-DSL, there is still a lot of manual labour involved. This means that it takes time and there are many steps where the user can cause errors. Additionally, the first and second step require a representation of the API that is detailed and comprehensible, which often times is not available.

3 Automating Web Service Integration

The introduction of the HTTP-DSL into DIME made it faster and easier to connect and use web services in DIME applications. However, the HTTP-DSL can also serve as a strong foundation for the fully automatic integration of web services into DIME.

However, a few requirements have to be met to enable these capabilities. First, we require machine-readable specifications for APIs, which are provided by the OAS. Then, we need to be able to process the specification files, as can be read in Subsection 3.1, to be able to use their information within DIME. The next step is to map the given information to the data model within DIME, which is described in Subsection 3.2. Similarly, the API endpoints found in the OAS need to be met with HTTP requests. This will be done using the HTTP-DSL, which can be read in Subsection 3.3.

3.1 Processing OpenAPI Specifications

Processing OASs means that the input in form of a specification file has to be automatically mapped to a domain model within DIME. The domain model consists of domain objects, as discussed earlier in Subsection 2.2, meaning that we essentially perform a deserialization of the OAS.

The OAS comes in two shapes, in the JSON format or in the YAML format. In this paper we focus on the prior, however, these formats are interchangeable for the most part. They are both machine-readable and they offer the capability to represent objects with `Strings`, `Integers`, `Booleans`, `Arrays`, and other nested objects. However, as simple as these formats may be, they can be used to describe almost everything. Hence, even if the decoding of such objects is possible, more knowledge is required to give them a semantic meaning.

3.1.1 The OpenAPI Specification meta-model

The decoding, or deserialization, of OASs is only possible if we can map the given JSON file to a model. Otherwise, we would be left with a set of deserialized objects that bear no meaning. This observation was already made earlier by Ed-douibi, Izquierdo, and Cabot in 2018, as they tried to automate the conversion of OpenAPI specifications (they called it OpenAPI Definitions) to UML[11] Diagrams [ECC18]. Amidst the many steps in their process to retrieve a valid UML model from the OAS, they made use of a meta-model that describes the OAS in detail. This meta-model originated in their paper "Example-Driven Web API Discovery" from 2017 [ECC17]. This model is openly available in an open source Github project and realized in the Eclipse Modeling Framework (EMF). Since DIME in its current version is also based on the EMF, the integration and use of the meta-model within DIME is quick and easy to realize.

The meta-model comprises three levels of abstraction, each highlighting a certain aspect of the OAS:

Behavioral The behavioral level of the model consists of all elements of an API that define the behavior of the API. The visualization of the model in form of a class diagram can be seen

[11] The Object Management Group created the UML, a modeling language that helps to visualize the design of software systems. See: https://www.omg.org/spec/UML/

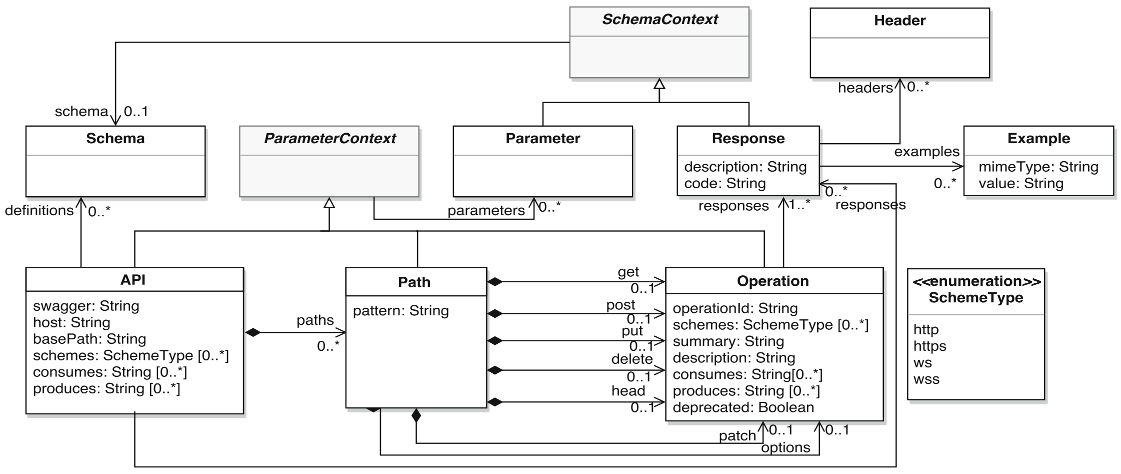


Figure 4: Behavioral level of the OAS meta-model. [ECC17]

in Figure 4. Here we can see all the components, beginning with the `API` element that represents the REST API and is thus the root element. It has many attributes, including the `version`, the `base path`, and many more. Additionally, the API element can have references to path elements, which in turn have operations and so on. This model could be useful to extract the objects from a given JSON file, or for the validation of deserialized objects within DIME.

Structural The structural level includes all structural elements, which are then used to specify the behavioral elements discussed before. The structural elements use an adapted subset of the JSON Schema Specification[12], however, the details are beyond the scope of this paper.

Serialization On the serialization level, the model gives information that can help with the serialization and deserialization of OASs.

If applied correctly, this meta-model could provide the structure of all possible valid specifications within the OAS. This could be helpful for deserialization, but also for the validation of generated models. One could either apply the model during the deserialization to exclusively fetch valid models, or one could first deserialize the objects and then check for validity. The second approach could be better at detecting faulty specifications, or even help with repairing the faulty models.

3.2 Mapping the API to Domain Objects

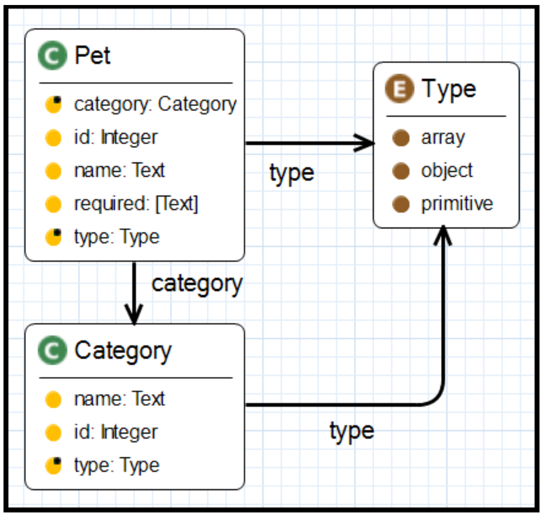
Within each OAS specification file all the data types are listed at the very end, encapsulated in the `definitions` object. This is also reflected in the meta-model in Figure 4, in the top left corner. Each data object, therefore, has its own schema, meaning a description of how it has to be encoded within the specification file.

Going back to the pet store example that was first introduced in Subsection 2.1, we can now

[12] JSON files can be required to conform to a certain schema, called JSON schema. More on that can be read at https://json-schema.org/draft/2020-12/json-schema-validation.html

```
 1 "Pet": {
 2       "type": "object",
 3       "required": [
 4         "name"
 5       ],
 6       "properties": {
 7         "id": {
 8           "type": "integer",
 9           "format": "int64"
10         },
11         "category": {
12           "$ref": "#/definitions/Category
               "
13         },
14         "name": {
15           "type": "string",
16           "example": "doggie"
17         }
18     }
```

(a) Representation in JSON format



(b) Representation as domain object in DIME.

Figure 5: The pet object taken from the pet store example.

discuss how an integration could look on the data layer. Figure 5 depicts an approach for a transfer from the OAS into DIME, with the example of a pet object. The schema of the object can be quickly retrieved from the OAS JSON file. The excerpt from the specification including the pet in JSON format can be seen in Figure 5a. The attributes of the object will now be described in detail.

type The `type` attribute defines that the pet object is indeed an `object`, in contrast to an `array` or a `primitive` data type.

required The `required` attribute (lines 3 to 5) lists all the properties that are mandatory in order for the object to be valid. In the case of the pet object these properties are the `name` and the `photoUrls` which refers to a picture of the animal.

properties The `properties` attribute (lines 6 to 17) is the key element of the whole object, as it describes which properties the object can hold. In the pet store example, the pet can have an `ID` of type `int64`, a `category` that refers to a category object (that is also found in the definitions segment of the OAS), and a name that is of type `string`.

It is possible to map OAS objects to DIME domain objects, as these in turn have a detailed expressiveness that allows for a smooth transfer. How this would look like in the data model can be seen in Figure 5b. The object is mapped into DIME, using a reference to an enum for the `type` attribute, a reference to a category object for the `category` attribute, and primitives for the `properties` attribute. Only the `required` attribute poses a challenge, as one cannot refer to the same properties twice within DIME. However, in this example the work around was to use a string array, to list all necessary properties.

This example also shows how the UML-like graphical representation of the objects immediately clarifies the relations between the objects. This effect would increase with the size of the data schema from the OAS and the complexity of the relations between the objects. Hence, the integration of the object structure of an API into DIME could serve the additional benefit of an easier comprehension of the structure.

3.3 Mapping the API to SIBs

In a similar vein to the data schema of an API, all the endpoints are also given in the OAS. The previously discussed behavioral level of the meta-model, depicted in Figure 4, shows the endpoints as a mixture of paths and operations. Each path can have up to seven operations, and each operation should, if possible, be mapped to a SIB that can be used in processes within DIME. While this could mean that APIs with hundreds of operations could in turn result in hundreds of available SIBs that might clutter the IME, this approach also has many advantages. On the one hand, the catalogue of SIBs can be browsed effortlessly since DIME provides a search bar to quickly find and filter SIBs, which is more organized than looking for the operations using tools like Swagger UI. On the other hand, the approach to have one single generic SIB per API that has to be configured, takes away from the typical LDE conviction to have one specific node per operation which improves the readability and structure of models. Furthermore, a generic node that reflects the complexity of hundreds of operations, would require a vast amount of configurational capabilities. This could result in a situation where the user takes more time to configure the SIB than it would take to browse through the SIB catalogue to find the fitting SIB for the job.

With this design choice in mind we can discuss how the HTTP-DSL comes into play. As much as it would technically be possible to take the OAS and create a code generator for it by hand, it would take a lot of time and effort. In contrast, the mapping of the OAS to the HTTP-DSL is easy to do, as the HTTP-DSL is tailored to the HTTP use case and already supports all the key elements that the OAS requires. Once the operations of the API are mapped to the HTTP-DSL, the DSL provides the framework to create SIBs and generate code if necessary.

3.3.1 An Exemplary Solution

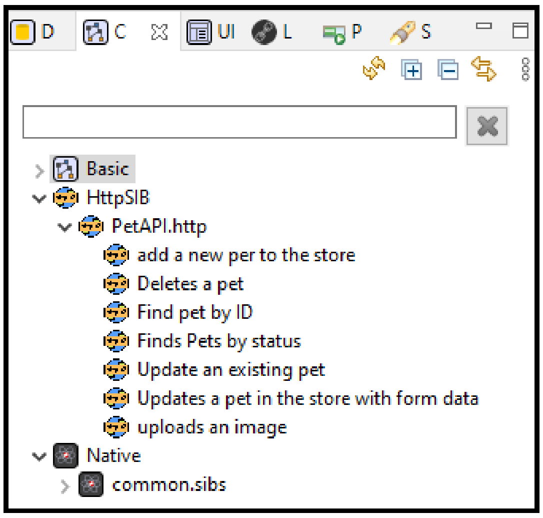
To see once again how the mapping could look like, we can compare the JSON code for the “Find pet by ID” operation illustrated in Figure 6a, with the HTTP-DSL version found in Figure 3.

As specified in the OAS meta-model, the overarching object is the path object (see line 1), which holds the operations (in this case GET). These two components are perfectly mapped to the path and type components in the HTTP-DSL. The summary of the operation found at line 6 of the JSON file can be used as the name for the HTTP request in the DSL, however, it is important to guarantee that the name is unique. The URL for the request is found at the very beginning of the OAS, thus it is not given in the JSON snippet here.

Finally, the operation in the JSON file has a `responses` attribute, which can also be mapped into the response component of the HTTP-DSL. What is interesting to note, other than the mapping of the application/json response type, is that just like in the OAS, we have data binding in the HTTP-DSL. As discussed in the previous section, we can retrieve the data model from the OAS and integrate it into the DIME application. This data model is now referenced in the HTTP-DSL in line 1, and we can now bind the `Pet` object in the response. If this is transfer

```
"/pet/{petId}": {
      "get": {
        "tags": [
          "pet"
        ],
        "summary": "Find pet by ID",
        "description": "Returns a pet",
        "operationId": "getPetById",
        "produces": [
          "application/json",
          "application/xml"
        ],...,
        "responses": {
          "200": {
            "description": "successful",
            "schema": {
              "$ref": "#/definitions/Pet"
            }
          }
        }
    }
```

(a) Representation in JSON format



(b) SIB Palette of API requests within DIME.

Figure 6: Exemplary images for an integration of the pet store API into DIME.

is done correctly for all operations, the HTTP-DSL allows for the automatic generation of SIBs from the DSL file. This means that the DIME user would end up with a palette of SIBs, as can be seen in Figure 6b. These items categorized under PetAPI.http, are the SIBs for the previously transferred operations, and they can be dragged and dropped into process models to work with the pet store API.

3.3.2 Extending the HTTP-DSL

While for many operations the abovementioned approach is sufficient, there are certain aspects within the OAS that would not be covered. The OAS allows for for operators such as *allOf*, *oneOf*, *anyOf* and *not*[13], which are used to specify allowed schemas for operations. For example, in the case of *allOf* the API user would be required to provide a data type that is valid against a range of given schemas, while the *not* operator explicitly excludes certain schemas from being used.

To be able to perfectly map operations to the HTTP-DSL, the HTTP-DSL would have to be extended. As of now, the DSL is specifically tailored to the Hypertext Protocol, meaning that any kind of logic that goes beyond that was not regarded during development. However, since in the future the DSL should work as a foundation for the solution covered in this paper, it makes sense to update the DSL appropriately.

[13] Detailed description of the oeprators provided by Swagger at https://swagger.io/docs/specification/data-models/oneof-anyof-allof-not/

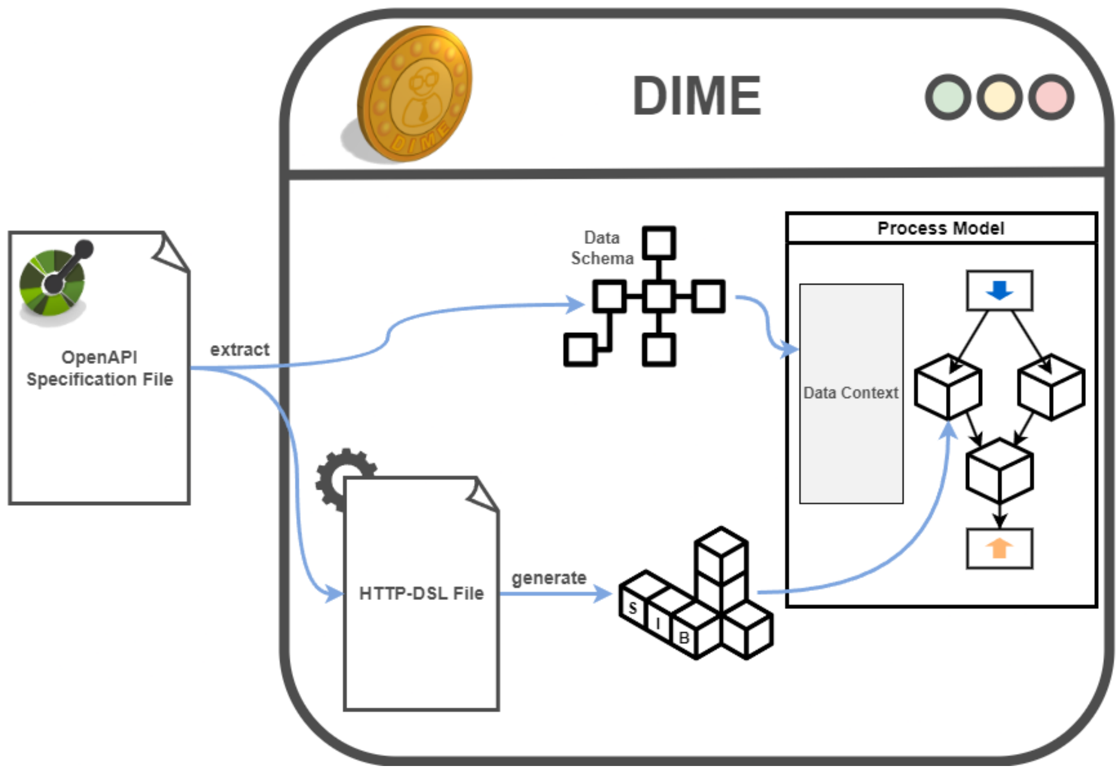


Figure 7: Overview of the process for the integration of a web service into DIME.

4 Conclusion

To conclude the work, we first begin with a summary of the overall process described in the paper in Subsection 4.1, followed by an evaluation in Subsection 4.2 and a discussion about further design ideas in Subsection 4.3.

4.1 Summary

This paper presents an approach for the automated integration of APIs into the DIME framework. This approach requires the transfer of API knowledge from OAS files to the data and process models within DIME. The steps given in Section 3 indicate a solution for the automated integration of web services into DIME. It outlines how a meta-model can help and gives examples of how the mapping of specific items within the OAS to the DIME models can look like. Overall, given the OAS file, the process can be split up into four main operations that are illustrated in Figure 7. First, the data schema is extracted from the OAS file and parsed into the Data Language within DIME. Then, the business logic of the API is extracted from the OAS file and parsed into the HTTP-DSL. The HTTP-DSL is then used to generate the SIB palette. Finally, the SIBs and the domain objects can be used within the process model to shape the DIME application.

4.2 Evaluation

The mapping from the OAS files (in JSON format) to the graphical models within DIME show how this transfer can even help to understand and analyze a web service. While there exist tools that can help with the comprehension of APIs, such as Swagger UI which was introduced in Subsection 2.1, the UML-like graphs of DIME are arguably better suited when it comes to the representation of data schemas. In addition, DIME users are already used to the models within DIME, increasing the aforementioned effect even more.

The drawback to this project is clearly the scope of APIs that can be integrated. As was mentioned earlier, the specification files for thousands of APIs can be sourced via the apis.guru service, however, this still excludes the vast majority of web services. An attempt was made to discover the OAS for web services by sending and analyzing exemplary requests [ECC17]. However, whether this approach works as promised or whether the future holds any new approaches to retrieving OAS files remains unanswered. That said, one thing can be said with certainty; internal or private APIs that are built using the OpenAPI stack will most of the time be compatible to the suggested software solution. This means that the integration of self-made web services in a DIME application could be automated, as long as this web service is developed using the proper tools.

An additional concern is the applicability of the solution to the wide range of OAS compliant APIs, since HTTP overall allows for many workflows that might not be covered. Overall it can be said that on the one hand the whole OAS to SIB automation can only be as good as the underlying HTTP-DSL, which means that for certain issues a solution would be to update the HTTP-DSL. Additionally, if edge cases emerge where the automated solution presented in this paper cannot be applied, the user can always use the HTTP-DSL manually or even extend the native SIBs library to have full control over their HTTP requests.

4.3 Open Design Questions

Since the design of the solution is not yet finished, many questions are still unanswered.

Usage within DIME One aspect that has not been discussed in the paper is the usability of the new tool within DIME. Would the DIME user be required to copy and paste the OAS text, or should it be possible to upload the OAS file? Could it even be possible to implement the solution in a way that a link to the desired web service is sufficient?

Generated items Since the current design of the solution requires the generation of domain objects, it should be discussed whether the use of these objects should be allowed throughout the whole DIME application. If many web services were to be used within a single DIME application, the data space might become cluttered.

New features There are many ideas for features that could be added to this software solution. For example, one could use varying icons, namespaces, or colors to differentiate between transferred requests from different APIs. Another idea would be to somehow use the various information texts within the OAS and provide them to the DIME user for ease of understanding. Before the implementation fully begins, the most important features will need to be addressed.

Bibliography

[BFK+16] S. Boßelmann, M. Frohme, D. Kopetzki, M. Lybecait, S. Naujokat, J. Neubauer, D. Wirkner, P. Zweihoff, B. Steffen. DIME: A Programming-Less Modeling Environment for Web Applications. In Margaria and Steffen (eds.), *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*. Lecture Notes in Computer Science, pp. 809–832. Springer International Publishing, Cham, 2016.
doi:10.1007/978-3-319-47169-3_60

[ECC17] H. Ed-douibi, J. L. Cánovas Izquierdo, J. Cabot. Example-Driven Web API Specification Discovery. In Anjorin and Espinoza (eds.), *Modelling Foundations and Applications*. Lecture Notes in Computer Science, pp. 267–284. Springer International Publishing, Cham, 2017.
doi:10.1007/978-3-319-61482-3_16

[ECC18] H. Ed-douibi, J. L. Cánovas Izquierdo, J. Cabot. OpenAPItoUML: A Tool to Generate UML Models from OpenAPI Definitions. In Mikkonen et al. (eds.), *Web Engineering*. Lecture Notes in Computer Science, pp. 487–491. Springer International Publishing, Cham, 2018.
doi:10.1007/978-3-319-91662-0_41

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000.

[Fou22] L. Foundation. Governance for the OpenAPI Initiative. 2022.
https://www.openapis.org/participate/how-to-contribute/governance

[Gro17] O. M. Group. Unified Modeling Language, v2.5.1. Jan. 2017.
https://www.omg.org/spec/UML/2.5.1/PDF

[Pol21] B. Pollard. The 2021 Web Almanac: Third Parties. Technical report, HTTP Archive, Nov. 2021. Issue: 7 Publication Title: The 2021 Web Almanac Volume: 3.
https://almanac.httparchive.org/en/2021/third-parties

[PS14] C. Pang, D. Szafron. Single Source of Truth (SSOT) for Service Oriented Architecture (SOA). In Franch et al. (eds.), *Service-Oriented Computing*. Lecture Notes in Computer Science, pp. 575–589. Springer, Berlin, Heidelberg, 2014.
doi:10.1007/978-3-662-45391-9_50

[Ste22] B. Steffen. *DSL-driven Integration of HTTP Services in DIME*. Bachelor Thesis, Technische Universität Dortmund, 2022.