APIComposer: Data-driven Composition of REST APIs[⋆]

Hamza Ed-douibi[1] [0000−0003−4342−4818], Javier Luis Cánovas Izquierdo[1] [0000−0002−2326−1700], Jordi Cabot[1,2] [0000−0003−2418−2489]

[1] Internet Interdisciplinary Institute (IN3)
Universitat Oberta de Catalunya (UOC). Barcelona, Spain
{hed-douibi,jcanovasi}@uoc.edu
[2] ICREA. Barcelona, Spain
jordi.cabot@icrea.cat

Abstract. More and more companies and governmental organizations are publishing data on the Web via REST APIs. The increasing number of REST APIs has promoted the creation of specialized applications aiming to combine and reuse different data sources to generate and deduce new information. However, creating such applications is a tedious and error-prone process since developers must invest much time in discovering the data model behind each candidate REST API, define the composition strategy, and manually implement such strategy. To facilitate this process, we propose an approach to automatically compose and orchestrate data-oriented REST APIs. For an initial set of REST APIs, we discover the data models, identify matching concepts, obtain a global model, and make the latter available on the Web as a global REST API. A prototype tool relying on OpenAPI for describing APIs and on OData for querying them is also provided.

Keywords: REST API, Modeling, OData, OpenAPI, API Composition

1 Introduction

More and more individuals and organizations are sharing their data on the Web, including governments and research initiatives. Web APIs have been increasingly used to make these data available on the Web and allow third parties to infer new information not visible at first glance. In particular, the REpresentational State Transfer (REST) has become the prominent architectural style mainly due to its adaptability to the Web, as it allows creating Web APIs by relying only on URIs and HTTP messages.

By enabling a programmatic access to data sources, REST APIs promote the creation of specialized data-driven applications that combine data from different sources to offer user-oriented value-added APIs. Creating such applications requires API discovery/understanding/composition and coding. Such tasks are not easy since developers should [11, 2]: (i) know the operations and data models of the APIs to compose; (ii) define the composition strategy; and (iii) implement an application (usually another Web API) realizing such strategy.

While automatic Web API composition has been heavily studied for the classical WSDL/SOAP style [20], REST API composition is of broad and current interest specially

[⋆] This work has been supported by the Spanish government (TIN2016-75944-R project).

after the emergence of new REST API specifications such as the OpenAPI specification[3] and OData[4]. OpenAPI is a vendor neutral, portable, and open specification initially based on Swagger[5] which allows defining the resources and operations of a REST API, either in JSON or YAML. The OpenAPI specification has become the choice of reference to describe REST APIs. As a result, OpenAPI is at the core of many research initiatives to, for instance, discover OpenAPI definitions [9, 6], provide semantic descriptions for OpenAPI definitions [7, 14], identify candidate REST APIs for selection [3], and allow semantic integration of REST APIs [19]. On the other hand, OData is an open protocol especially useful to expose and consume data sources as REST APIs.

In this paper we propose a lightweight model-based approach to automatically compose data-oriented REST APIs given an initial set of OpenAPI definitions (potentially inferred when not explicitly available). As a result of the composition, we obtain a global API that hides the complexity of the composition process to the user. Indeed, a user queries the global API and, in a completely transparent way, the global queries triggers a fully automatic process that accesses the individual APIs and combines their data to generate a single response.

To facilitate the consumption of the global API, we expose it as an OData service. OData allows creating resources which are defined according an *Entity Data Model (EDM)* and can be queried by Web clients using a URL-based query language in an SQL-like style. In our approach, this EDM corresponds to the data schema behind the global API, which is generated during the composition process based on the discovery of matches between the individual data schema of each single API. All these schemas are represented as models and their manipulation (e.g., concept matching or composition) are implemented as model transformations. Working at the model level helps us focus on the domain concepts while abstracting from the low level technical details [18].

The rest of the paper is organized as follows. Section 2 describes our approach, while Sections 3 and 4 explain its main steps. Section 5 illustrates our approach using an example. Section 6 presents our tool support. Section 7 discusses some related works. Finally, Section 8 concludes the paper and presents some future work.

2 Our Approach

We propose a model-based approach to compose data-driven REST APIs. From a set of initial REST APIs, our approach creates a global API exposing a unified data model merging the data models of the initial APIs. The global model is exposed as an OData service, thus allowing end-users to use the OData query language to get the information they need in an easy and standard way.

Figure 1 shows an overview of our approach. APIComposer takes as input the OpenAPI definitions of the REST APIs to be composed. Such definitions may be (i) supplied by the API provider, (ii) generated using tools such as APIDiscoverer [9] or AutoREST [6], which are able to infer OpenAPI definitions from API call examples

[3] `www.openapis.org`.
[4] `www.odata.org`.
[5] `https://swagger.io`

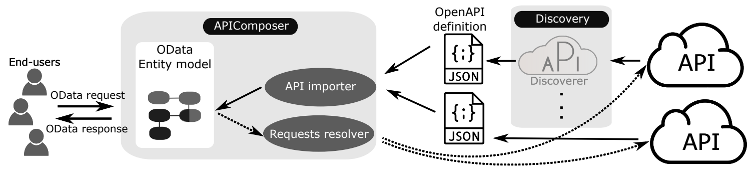


Fig. 1. Overview of our approach.

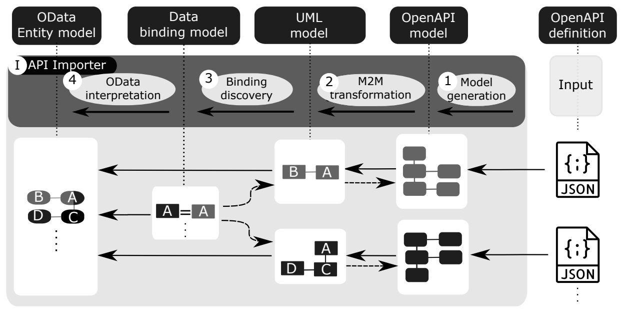


Fig. 2. Composition process.

or API documentation pages, respectively. (iii) or derived from other API definition formats (e.g., API Blueprint or RAML) using tools such as API Transformer[6].

Our approach includes two components, namely: (i) API importer, in charge of integrating a new REST API to the global API; and (ii) Requests resolver, responsible for processing the user requests and returning the queried data. We explain each component in the following sections.

3 API Importer

Figure 2 shows the API importer process. For each input OpenAPI definition, the API importer first generates an equivalent model conforming to our OpenAPI metamodel (see step 1 in Fig. 2). We previously introduced this metamodel alongside the discovery process [9]. The generation of the OpenAPI model is rather straightforward since the OpenAPI metamodel conforms to the OpenAPI specification and only special attention had to be paid to resolve JSON references.

The second step of the process (see step 2 in Fig. 2) performs a model-to-model transformation to generate a UML model, which emphasizes the data schema of the input API to facilitate the matching process later on. This process consists on iterating over the data structures in the OpenAPI model (i.e., the *schema* elements) to generate the adequate UML elements (i.e. classes, properties and associations elements). This process relies on our tool OpenAPItoUML[7] which generates UML models from OpenAPI definitions [10].

[6] http://apimatic.io/transformer
[7] http://hdl.handle.net/20.500.12004/1/A/O2U/001

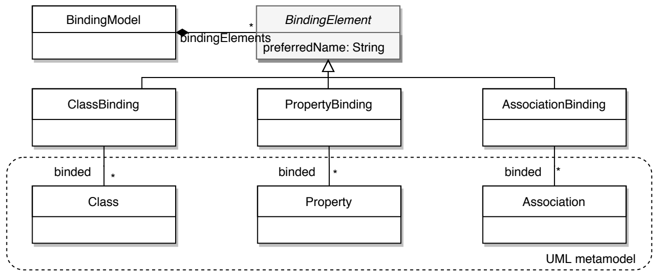


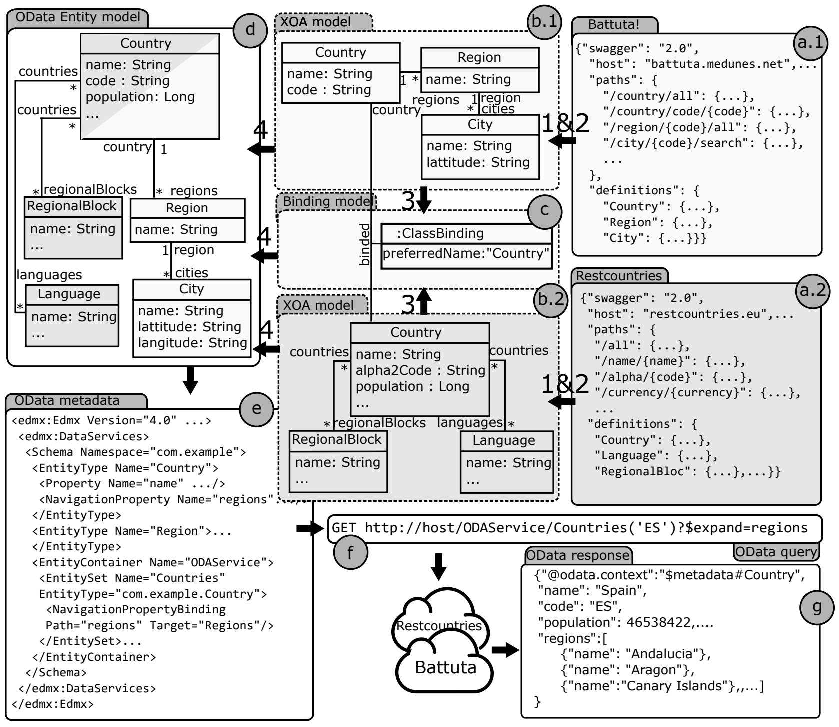
Fig. 3. Excerpt of the binding metamodel.

The third step (see step 3 in Fig. 2) analyzes the UML models to discover matching elements and creates bindings to express the matches between them. The binding model conforms to the binding metamodel which allows creating traceability and binding elements for the data elements in the UML models. Figure 3 shows an excerpt of the binding metamodel. The `BindingModel` element is the root element of the binding metamodel and includes a set of binding elements (i.e., `bindingElements` reference). The `ClassBinding`, `PropertyBinding`, and `AssociationBinding` elements allow defining bindings to `Class`, `Property`, and `Association` elements in a UML model, respectively. Each element includes a preferred name (i.e., the `preferredName` attribute inherited from the `BindingElement` element) and a set of binded elements (i.e., the `binded` references). We currently support a simple two-step matching strategy to define the bindings between elements. The first step finds matching candidates based on their names and types. Then, the second step validates the matches by calling the REST APIs and comparing data related to each candidate. Our experience showed that such strategy is sufficient for APIs coming from the same provider/domain, which share the same concept names across their APIs. However, our approach can be extended in order to support more advanced matching strategies specially for cross-domain composition by relying on, for instance, database schema integration approaches [4] or the new approaches to add semantic descriptions to OpenAPI [7, 14]. Also, a designer can manually curate the initial automatic result.

Finally, the last step creates an OData metadata document from: (i) the generated UML models, and (ii) the binding model. This document includes an OData entity model created by merging all the data models of the input REST APIs and resolving the bindings between them. Thus, the creation process iterates over all the data elements in the UML models and creates a new element in the entity model if there is not a binding linking such element to another element, or merging both elements otherwise. The OData metadata document is the standard way OData provides to let end-users know how to query data using the OData query language.

4 Requests Resolver

The Requests resolver is an OData service exposing the created data model, and in charge of processing the end-user queries and building the query response based on the bindings and extended OpenAPI models generated during the import phase. Such process involves two steps, namely: query resolution and response resolution.



```
{"swagger": "2.0",
  "host": "battuta.medunes.net",...
  "paths": {
    "/country/all": {...},
    "/country/code/{code}": {...},
    "/region/{code}/all": {...},
    "/city/{code}/search": {...},
    ...
  },
  "definitions": {
    "Country": {...},
    "Region": {...},
    "City": {...}}}
```

```
 {"swagger": "2.0",
  "host": "restcountries.eu",...
  "paths": {
   "/all": {...},
   "/name/{name}": {...},
   "/alpha/{code}": {...},
   "/currency/{currency}": {...},
   ...
  "definitions": {
   "Country": {...},
   "Language": {...},
   "RegionalBloc": {...},...}}
```

```
<edmx:Edmx Version="4.0" ...>
 <edmx:DataServices>
  <Schema Namespace="com.example">
   <EntityType Name="Country">
    <Property Name="name" .../>
    <NavigationProperty Name="regions" .../>
   </EntityType>
   <EntityType Name="Region">...
   </EntityType>
   <EntityContainer Name="ODAService">
    <EntitySet Name="Countries"
    EntityType="com.example.Country">
     <NavigationPropertyBinding
     Path="regions" Target="Regions"/>
    </EntitySet>...
   </EntityContainer>
  </Schema>
 </edmx:DataServices>
</edmx:Edmx>
```

```
GET http://host/ODAService/Countries('ES')?$expand=regions
```

```
{"@odata.context":"$metadata#Country",
 "name": "Spain",
 "code": "ES",
 "population": 46538422,....
 "regions":[
     {"name": "Andalucia"},
     {"name": "Aragon"},
     {"name":"Canary Islands"},,...]
}
```

Fig. 4. Illustrative example.

The query resolver interprets first the OData query in order to determine the target resource to retrieve (i.e., a collection of entities, a single entity or a property) and the options associated with the query (e.g., filter or ordering). The resolver transforms then the query into a set of API calls by tracing back the origin of each element thanks to the binding model. From the binding model we navigate first to the UML models then to the OpenAPI models. These OpenAPI models contain all the necessary details to generate the actual calls[8] as they contain the same information as the original OpenAPI definitions.

On the other hand, the response resolver is in charge of providing the result to the end-user by combining the different API answers in a single response conforming to the OData entity model defined in the OData metadata document.

5 Illustrative Example

To illustrate our approach, we consider the following REST APIs: Battuta[9], which allows retrieving the regions and cities of a country; and Restcountries[10], which

[8] We created a set of heuristics which map operations to entity elements. More information can be found at our repository.

[9] https://battuta.medunes.net/

[10] https://restcountries.eu/

allows getting general information about countries such as their languages, currencies and population. Our goal is to create a global API combining both APIs. Thanks to the global API, users will be able to query both kinds of country information (either geographical, general or both) in a transparent way, (i.e., without having to specify in each query what API/s the query should read from). As a preliminary step, we generated the OpenAPI definitions describing Battuta and Restcountries APIs using APIDiscoverer [9]. We used the resulting definitions as inputs for our approach.

Figure 4 illustrates the results of applying our composition mechanism on these APIs. Figures 4.a.1 and 4.a.2 show parts of the OpenAPI definitions of Battuta and Restcountries APIs, respectively. As explained in the previous section, the first step of the process generates an OpenAPI model describing the input definition, while the second step generates UML model where the data aspects have been refined and highlighted. Figure 4.b.1 and 4.b.2 show the generated UML models for Battuta and Restcountries APIs, respectively. As can be seen, the data model for the Battuta API includes the classes *Country*, *Region* and *City*, while the model for the Restcountries API includes the classes *Country*, *RegionalBlock*, and *Currency*. Figure 4.c shows the binding model including a *ClassBinding* element for the *Country* entities of both data models, identified as a valid matching concept.

Figure 4.d shows the OData Entity model created by joining the elements of both data models and resolving the match between the *Country* entities. As can be seen, the *Country* class is shared between both APIs and includes properties and relationships coming from both APIs. Figure 4.e shows an excerpt of the Metadata document of the OData Entity model. This document can be retrieved by appending `$metadata` to the URL of the OData application and allows end-users to understand how to query the data.

OData defines a URL-based query language sharing some similarities with SQL that allows users to query the data described in the metadata document [16]. Figure 4.f shows an example of an OData request to retrieve the details of *Spain* and its regions using the query option `$expand`[11]. This request relies on the concept binding for `Country`, which allows process the request using Restcountries API (mainly for information about the country) and Battuta API (for information about the regions). Thus, the request is traced back to both Restcountries and Battuta APIs (i.e., the operations `/alpha/{code}` and `/region/{code}/all`, respectively), which are therefore queried. Figure 4.g shows the response in OData format. More query examples can be found in our repository [1].

6 Tool Support

We created a proof-of-concept tool implementing our approach which we made available as an Open Source application [1]. Our tool has been implemented as a Java web application which can be deployed in any Servlet container (e.g., Apache Tomcat). The application relies on JavaServer Faces (JSF), a server-side technology for developing Web applications; and Primefaces[12], a UI framework for JSF applications; to implement a wizard guiding the user through the steps of the API importer and displaying the different models. The OpenAPI metamodel, the extended OpenAPI metamodel, and

[11] `$expand` specifies that the related resources have to be included in line with retrieved one.

[12] `http://www.primefaces.org`

the binding metamodel have been implemented using the Eclipse Modeling Framework (EMF). OData implementation relies on Apache Olingo[13] to provide support for OData entity model, OData query language, and serialization.

7 Related Work

Most of the previous works on REST APIs composition are tight to specific API description languages [12]. For instance, some of them relied on WADL (Web Architecture Description Language) and hREST (HTML for RESTful Services) to describe the behavior of REST APIs, and WSMO (Web Service Modeling Ontology) and SA-REST (Semantic Annotation of Web Resources) to add semantic annotations (e.g., [15, 8, 13]). However, none of them gained a broad support mainly because those languages were not successfully adopted [12]. We decided to rely on the OpenAPI specification, which can be seen as a reference solution for REST APIs. The emergence of OpenAPI definitions has motivated initiatives to annotate OpenAPI definitions with semantic descriptions [7, 14] and identify APIs for selection [3]. Our approach differs from these works by putting OpenAPI specification at the core of the composition strategy, but we can profit in the future from them (e.g., by considering semantic descriptions for concept matching).

Our approach focuses on the composition of data-oriented APIs, which allows us to rely on the family of approaches proposed for JSON data [5] and in the database world for schema matching and merging [17, 4]. To the best of our knowledge, only the work by Serrano et al. [19] proposes a similar approach to ours but theirs require annotating REST APIs with Linked-Data ontologies and uses SPARQL to query to composed APIs.

8 Conclusion

We have presented a model-based approach to automatically compose and orchestrate data-driven REST APIs. Our approach parses OpenAPI definitions to extract data models, expressed as UML models, which are combined following a pragmatic matching strategy to create a global data model representing the union of all the data for the input APIs. The global model is exposed as an OData service, thus allowing users to easily perform queries using the OData query language. Queries on the global model are automatically translated into queries on the underlying individual APIs. In case users are not familiar with OData, OpenAPI definitions could also be easily derived from OData services[14]. Also, note that we illustrated our composition using OData but a similar approach could be followed to generate GraphQL APIs instead.

As future work we are interested in considering semantic descriptions for improving the matching strategy and non-functional aspects (like Quality-of-Service, QoS, or price) in the generation of the global model when alternative APIs have a high degree of overlapping. The latter would allow users to choose different resolution paths for the same query based on their preferences (e.g., by using free APIs when possible). We would like to extend our approach in order to support not only data retrieval but also data

[13] `http://olingo.apache.org/`

[14] `https://github.com/oasis-tcs/odata-openapi`

modification (i.e., support all CRUD operations). We are also interested in improving the maintainability of our approach by allowing the update of the composed APIs as they evolve.

References

1. APIComposer. `http://hdl.handle.net/20.500.12004/1/A/APIC/001`
2. Aué, J., Aniche, M., Lobbezoo, M., van Deursen, A.: An exploratory study on faults in web api integration in a large-scale payment company. In: Int. Conf. on Software Engineering: Software Engineering in Practice. pp. 13–22 (2018)
3. Baresi, L., Garriga, M., De Renzis, A.: Microservices identification through interface analysis. In: Eur. Conf. on Service-Oriented and Cloud Computing. pp. 19–33 (2017)
4. Boronat, A., Carsí, J.Á., Ramos, I., Letelier, P.: Formal model merging applied to class diagram integration. Electronic Notes in Theoretical Computer Science **166**, 5–26 (2007)
5. Cánovas Izquierdo, J., Cabot, J.: Composing JSON-Based Web APIs. In: Int. Conf. on Web Engineering. pp. 390–399 (2014)
6. Cao, H., Falleri, J.R., Blanc, X.: Automated Generation of REST API Specification from Plain HTML Documentation. In: Int. Conf. on Service-Oriented Computing. pp. 453–461. Springer (2017)
7. Cremaschi, M., De Paoli, F.: Toward Automatic Semantic API Descriptions to Support Services Composition. In: Eur. Conf. on Service-Oriented and Cloud Computing. pp. 159–167 (2017)
8. De Giorgio, T., Ripa, G., Zuccalà, M.: An approach to enable replacement of SOAP services and REST services in lightweight processes. In: Int. Conf. on Web Engineering. pp. 338–346 (2010)
9. Ed-Douibi, H., Cánovas Izquierdo, J.L., Cabot, J.: Example-driven Web API Specification Discovery. In: Eur. Conf. on Modelling Foundations and Applications (2017)
10. Ed-Douibi, H., Cánovas Izquierdo, J.L., Cabot, J.: OpenAPItoUML: a Tool to Generate UML Models from OpenAPI definitions. In: Int. Conf. on Web Engineering (2018)
11. Espinha, T., Zaidman, A., Gross, H.G.: Web API growing pains: Stories from client developers and their code. In: Int. Conf. on Software Maintenance, Reengineering and Reverse Engineering. pp. 84–93 (2014)
12. Garriga, M., Mateos, C., Flores, A., Cechich, A., Zunino, A.: Restful service composition at a glance: A survey. Journal of Network and Computer Applications **60**, 32–53 (2016)
13. Lanthaler, M., Gütl, C.: Towards a RESTful service ecosystem. In: Int. Conf. on Digital Ecosystems and Technologies. pp. 209–214 (2010)
14. Musyaffa, F.A., Halilaj, L., Siebes, R., Orlandi, F., Auer, S.: Minimally Invasive Semantification of Light Weight Service Descriptions. In: Int. Conf. on Web Services. pp. 672–677 (2016)
15. Pautasso, C.: RESTful Web service composition with BPEL for REST. Data & Knowledge Engineering **68**(9), 851–866 (2009)
16. Pizzo, M., Handl, R., Zurmuehl, M.: OData version 4.0 part 2: URL Conventions. Tech. rep., OASIS (2014)
17. Rahm, E., Bernstein, P.A.: A Survey of Approaches to Automatic Schema Matching. the VLDB Journal **10**(4), 334–350 (2001)
18. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Softw. **20**(5), 19–25 (2003)
19. Serrano, D., Stroulia, E., Lau, D., Ng, T.: Linked REST APIs: A Middleware for Semantic REST API Integration. In: Int. Conf. on Web Services. pp. 138–145 (2017)
20. Sheng, Q.Z., Qiao, X., Vasilakos, A.V., Szabo, C., Bourne, S., Xu, X.: Web services composition: A decade's overview. Information Sciences **280**, 218–238 (2014)