TYPHON
POLYGLOT AND HYBRID PERSISTENCE ARCHITECTURES FOR BIG DATA ANALYTICS

Project Number 780251

D4.6 Data Access Layer Generator

Version 1.0
9 July 2020
Final

Public Distribution

Centrum Wiskunde & Informatica (CWI) and SWAT.engineering

Project Partners: **Alpha Bank**, **ATB**, **Centrum Wiskunde & Informatica**, **CLMS**, **Edge Hill University**, **GMV**, **OTE**, **SWAT.Engineering**, **The Open Group**, **University of L'Aquila**, **University of Namur**, **University of York**, **Volkswagen**

Project Partner Contact Information

Alpha Bank Vasilis Kapordelis 40 Stadiou Street 102 52 Athens Greece Tel: +30 210 517 5974 E-mail: vasileios.kapordelis@alpha.gr	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de
Centrum Wiskunde & Informatica Tijs van der Storm Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 9333 E-mail: storm@cwi.nl	**CLMS** Antonis Mygiakis Mavrommataion 39 104 34 Athens Greece Tel: +30 210 619 9058 E-mail: a.mygiakis@clmsuk.com
Edge Hill University Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk	**GMV Aerospace and Defence** Almudena Sánchez González Calle Isaac Newton 11 28760 Tres Cantos Spain Tel: +34 91 807 2100 E-mail: asanchez@gmv.com
OTE Theodoros E. Mavroeidakos 99 Kifissias Avenue 151 24 Athens Greece Tel: +30 697 814 7618 E-mail: tmavroeid@ote.gr	**SWAT.Engineering** Davy Landman Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 633754110 E-mail: davy.landman@swat.engineering
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org	**University of L′Aquila** Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it
University of Namur Anthony Cleve Rue de Bruxelles 61 5000 Namur Belgium Tel: +32 8 172 4963 E-mail: anthony.cleve@unamur.be	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Volkswagen Behrang Monajemi Berliner Ring 2 38440 Wolfsburg Germany Tel: +49 5361 9-994313 E-mail: behrang.monajemi@volkswagen.de	

Document Control

Version	Status	Date
0.3	First full draft	26 June 2020
0.8	Version for partner reviews draft	3 July 2020
1.0	Final updates from QA review	9 July 2020

Table of Contents

Executive Summary

There is usually a mismatch between databases' internal representation model and the one that programming languages expect. The quintessential example would be the relational model of a RDBMS and the object model of an object-oriented language like Java. A Data Access Layer (DAL) is an API that aims to bridge the gap between these two representations by providing an abstraction layer over the backend technology, making it more accessible to programmers. In the case of the RDBMS-OO case, a well-known DAL would be the Object-Relational Mapping (ORM) technologies.

The Typhon project provides diverse tools and languages in order to abstract over heterogeneous data backend technologies. Abstract data models can be defined using the TyphonML modeling language, independently of the specific data stores in which the data will end up. Such data can be queried using the TyphonQL query language, an SQL-like language to query over TyphonML-defined data.

Typhon programmers, thus, face similar challenges in term of the representation gap as those described above. To bridge this gap between the TyphonML conceptual model and a particular programming model, a Typhon Data Access Layer has been developed. This DAL provides a convenient optional tool to program systems that use a Typhon polystore as backend. This document discusses the DAL's design, technological stack, internals and usage.

1 Introduction

1.1 Overview

This document reports on Task 4.5, namely, Data Access Layer Generation. The objective thereof is stated in the Amendment to the Technical Annex [3]:

> This task will design and implement a framework for generating ready-to-use APIs through which application developers will be able to perform CRUD (create, read, update and delete) operations and queries on hybrid polystores. Multiple API technologies will be considered including native Java, REST and Apache Thrift.

In this report we discuss the design of the Typhon Data Access Layer, describe its implementation and present use scenarios.

1.2 Structure of the Deliverable

In Section 2, we discuss the alternatives we consider for implementing a Data Access Layer and our main arguments behind our choice (a REST-based API). In Section 3, we discuss the architecture of the solution together with technical implementation details. In Section 4 we present an overview of the Typhon Data Access Layer specification, that is, the operations made available on a Typhon polystore by means of this abstraction layer. Section 5 presents scenarios of use by means of using third-party generic tools or generating language-specific clients. Finally, Section 6 concludes.

2 Technological Alternatives

We considered mainly two alternatives for implementing a DAL for the Typhon ecosystem. In both cases, a central artifact is the TyphonML model that constitutes the abstract data model from which DAL operations can be extracted (see Deliverable 2.4 [2]). The alternatives under consideration were:

1. *Java Generator*: Reads a TyphonML model and outputs a set of classes representing the model, each of them with a set of methods implementing the CRUD operations.
2. *REST API*: Acts as an intermediate layer from whom clients for a given programming language could be generated.

In this section, we discuss these alternatives with their relative weaknesses and advantages.

2.1 Java Generator

The idea behind this alternative is to automatically generate Java classes for the entities defined in a TyphonML model. We have chosen to consider the generation to Java since the internal library providing a connection to the polystore (namely the `PolystoreConnection`) is being coded in Java as well.

Each entity in the TyphonML model would correspond to a class, featuring a series of CRUD methods. The content of these methods corresponds to calls to the Java Polystore API (see Deliverable 4.5 [1]), passing the right TyphonQL queries that execute the intended CRUD logic. Listing 1 shows an example of how such a generated method would look like.

```
public class Product {
  private String uuid;
  private XMIPolystoreConnection connection;
  private String xmiModel;
  private List<DatabaseInfo> databases;
  ...
  public void delete() {
    String query = "delete Product p where p.@id == #" +this.uuid;
    try {
      connection.executeUpdate(this.xmiModel, this.databases, query);
    } catch (RuntimException e) {
        throws DeleteException("Problems removing product " + this.uuid, e);
    }
  }
}
```

Listing 1: Generated class example (extract)

The development cycle for a developer who wants to interact with a Typhon polystore consists of: a) generation of the Java classes from a given TyphonML model representation, b) coding against the generated classes.

2.2 REST API

REST (Representational STate transfer) is the de-facto technique for programmatic communication over the Web. The premise is simple: URIs identify resources, and a set of HTTP standard verbs allow us to execute actions on such resources. For instance, a resource `http://foo.bar/Product` represents a collection of products and using the HTTP method POST we can send the representation of a new product to such collection, expecting the server to create the corresponding resource for us. The format of such a representation depends on the server, being JSON a popular exchange standard.

Businesses expose APIs as REST services in order to guarantee technology-independence: REST services are not tied to a particular language implementation. To communicate with them, it suffices to have an HTTP client. However, communicating with a REST endpoint in this way poses similar challenges to those described at the beginning of this document: there will be a gap between the server representation and the domain objects in the language. To overcome this, a REST service can be described using the OpenAPI specification [1], a machine-friendly description of the available services in a particular endpoint, together with the expected payload formats. In order to keep the OpenAPI document up to date with a certain TyphonML model, it could be generated from it, as shown in Figure 1.

[1] https://www.openapis.org/

```
entity Product {
  name : string[256]
  description : string[256]
  price : int
  productionDate : date
  availabilityRegion: polygon

  reviews :->
    Review."Review.product"[0..*]
  tags -> Tag[0..*]
  inventory :-> Item[0..*]
  category -> Category[1]
  wish :-> Wish."Wish.product"[1]
}
...
```

⇒

```
{
  "host": "localhost:8080",
  "basePath": "/crud/",
  "paths":{
    "/Product": {
      "post": {
        "tags": [
          "Product-rest-controller"
        ],
        "summary": "storeProduct",
        "operationId": "storeProductUsingPOST",
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [ ... ],
        "responses": {
          "200": {
            "description": "OK",
            "schema": {
              "type": "boolean"
            }
          },
          "201": {
            "description": "Created"
          },
          ...
      }
    },
    "/Product/{id}": {
      "get": {
        "tags": [
          "Product-rest-controller"
        ],
        "summary": "getProduct",
        "operationId": "getProductUsingGET",
        "produces": [
          "application/json"
        ],
        "parameters": [ ... ],
        "responses": {
          "200": {
            "description": "OK",
            "schema": {
              "$ref": "#/definitions/Product"
            }
          },
          ...
        },
...
}
```

Figure 1: TyphonML model and corresponding OpenAPI specification generated from it (extracts)

2.3 Comparison between the alternatives

The two aforementioned alternatives have relative advantages and drawbacks. In this Section we elaborate on them across a number of criteria, and then discuss how we weighed on them in order to decide which one we adopted.

- *Language Independence.* It goes without saying that the code generator would target one specific platform, in our case, Java. The clients of the DAL will then need to be coded in Java as well. The REST API, on the other hand, exposes the operations on the entities in a technologically agnostic manner. Clients for a diverse number of languages can be generated from the OpenAPI specification using standard generation tools [2].
- *Customizability.* When generating Java code, we as Typhon meta-programmers have the liberty to conceive all sorts of custom abstractions that reflect the TyphonML model in code. In the case of a REST API, instead, the adherence is to the OpenAPI standard and therefore all the clients will rely on this generic model.
- *Nature of the generation.* In the case of the Java code generator, the DAL artifacts are generated from a TyphonML representation. If the model changes, all the classes need to be regenerated based on the new model. Similarly, if the model changes in the case of a REST DAL, a new OpenAPI document would be automatically regenerated and therefore whichever clients that had been generated based on the previous document need to be regenerated as well. The REST DAL features an intermediate representation, namely the OpenAPI description, that is always up to date with the last ML model that resides in the polystore.
- *Genericity of clients.* The Java code generator will only generate one set of classes per model. Programmers can generate diverse applications based on the specific artifacts associated with a ML model using the generated classes as a library. A REST-based DAL instead provides a REST interface that can be exploited by generic tools to query the polystore using a higher level model. Thus, generic tools such as curl, Postman, etc. can be used. Besides that, as we already mentioned, clients for diverse programming languages can also be generated.

When looking at these different criteria we see that the biggest relative advantage of a custom generator lies in its customizability, i.e., that the generated classes can adhere as much as we decide to the TyphonML model, probably exploiting better OO abstractions since we target Java. However, we considered that the advantages in term of language independence and flexibility of clients in the case of a REST API outweigh the benefits of customizability of the generation-based solution and we have thus decided to implement the Typhon Data Access Layer using REST.

[2] https://github.com/OpenAPITools/openapi-generator

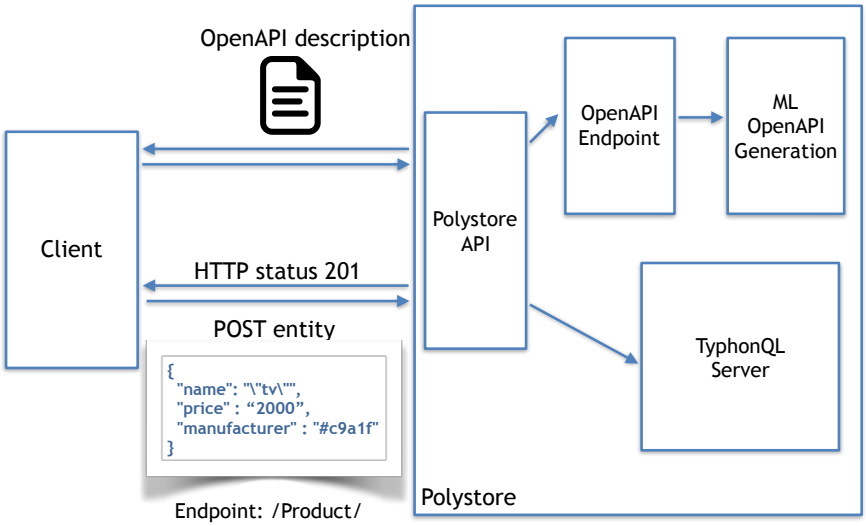

Figure 2: High-level view of the DAL components

3 Architecture and Implementation

The Typhon REST-based Data Acces Layer consists of two independent components:

- OpenAPI generator: Generates an OpenAPI specification from a given TyphonML model, in order for it to be made available through the polystore.
- REST service: Maps HTTP requests to TyphonQL queries. This component is model-agnostic, in the sense that it merely translates messages conformed by endpoint patterns and payloads to TyphonQL queries, provided that the right formats are used. In order to take advantage of model-level consistency guarantees, users can use the specifications provided by the OpenAPI document that resides in the polystore.

Figure 2 shows a high-level view of the architecture, showing how the DAL components fit in the polystore context. The Polystore API is the front-end component that acts as the façade for all the services provided by a Typhon polystore, from authentication to analytics, including access to the current ML and DL models, and a TyphonQL endpoint to send queries to the polystore via HTTP. The DAL components needed therefore to be integrated within the Polystore API. In the upper right part of the figure we can see the OpenAPI generation components, while in the lower right section, the component implementing the REST service, namely, the TyphonQL Server.

The ML OpenAPI generation component is the one invoked by the Polystore API every time a new TyphonML model is set in the polystore. The generated OpenAPI schema is made available at an OpenAPI endpoint.

In the case of the REST service, this is implemented within the TyphonQL server, and thus, when the Polystore API receives an invocation at its CRUD endpoint, it redirects the call to the TyphonQL server.

The next sections provide details about the implementation of the DAL components.

3.1 Generation of the OpenAPI schema

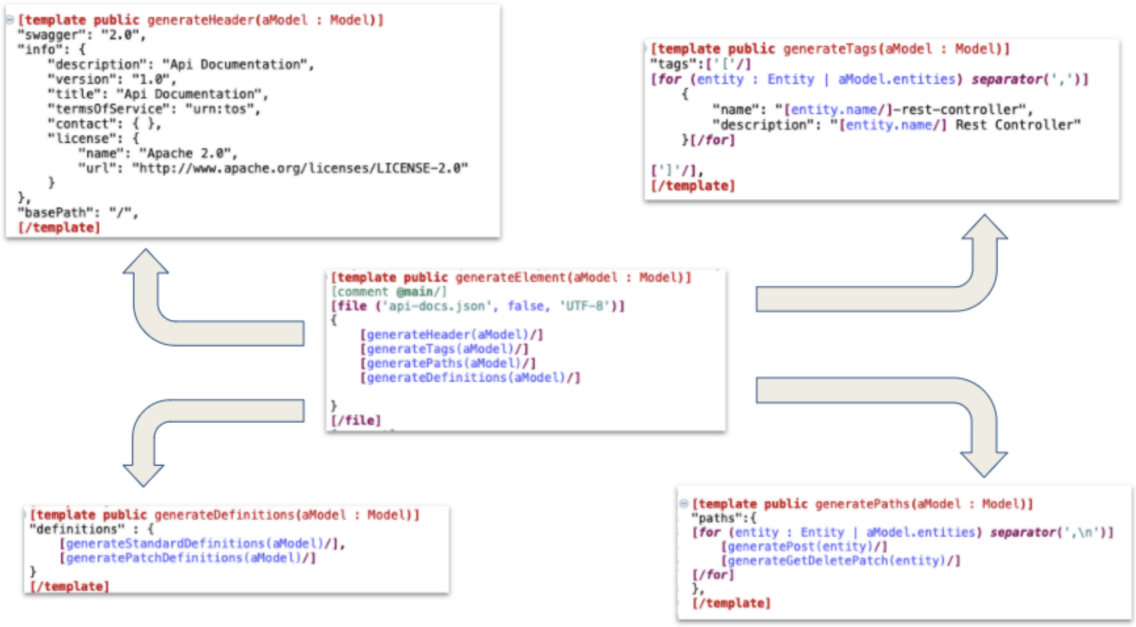

Figure 3: Coordinated Acceleo-based templates

In this section, we describe how the OpenAPI specification[3] is generated from the TyphonML model. OpenAPI is a specification for describing, consuming, and visualizing RESTful web services which allows both humans and machines to discover and understand the provided services. An OpenAPI definition can be used for many purposes, e.g., documentation, generation of clients in various programming languages, displaying APIs as a web UI, testing, and many other use cases.

If a TyphonML specification is available, a synthesis tool can be applied to generate the corresponding OpenAPI specification by using a set of coordinated Acceleo-based[4] model-to-code transformations depicted in Figure 3. The `generateElement` main template relies on different templates, where each one is responsible for generating a specific part of the OpenAPI specification. The `generatedHeader` template provides the static header. The `generatePath`, `generatesTags`, and `generateDefinition` templates generate the list of CRUD REST resources for each conceptual entity, a list of tags to each API operation, and the data model definition respectively.

It is worth noting that the OpenAPI specification depends on the TyphonML model: if the model changes, the OpenAPI specification should be regenerated. The it.univaq.disim.typhonml.acceleo generator is an Eclipse-based pluging component that allows us to generate the OpenAPI specification in different ways, i.e., by an Eclipse UI contextual menu, by Java APIs, and by an executable jar.

- The Typhon Eclipse UI contextual menu allows the modeler to produce the OpenAPI specification of a given TyphonML model. Then, she can use it to directly generate clients in various programming languages that programmatically interact with the polystore resources.

[3] https://www.openapis.org/
[4] https://www.eclipse.org/acceleo/

- When a model is uploaded to the polystore, the polystore will interact with the Polystore API to generate the OpenAPI specification. As we have seen before, such specification will be made available on the polystore in a dedicated endpoint.
- The executable JAR can be used for different purposes, for instance, continuous integration frameworks can run the generator as standalone program during build and test steps.

3.2 REST service

Figure 4 shows a more detailed view of the REST service part of the DAL. The TyphonQL server is a component inside of the polystore that provides an HTTP endpoint for executing queries (see Deliverable 4.5 [1]). It parses the JSON objects that contain the TyphonQL queries and executes them accordingly, marshalling the results as JSON if needed. All the querying endpoints are implemented using Java servlets, so we decided to use the same technology to implement the DAL endpoint.

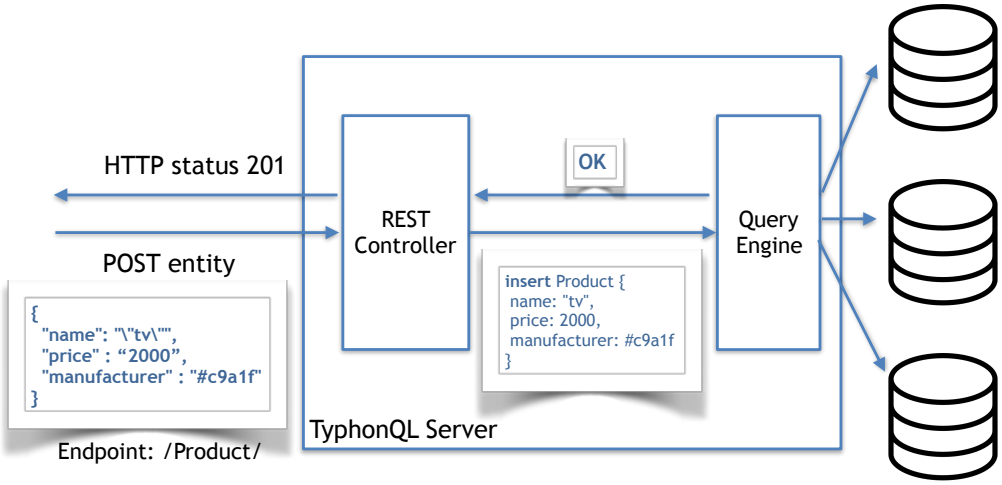

Figure 4: Architecture of the REST service implementing the DAL

For implementing the DAL services, we have added a new servlet dubbed "CRUD" (in Figure 4 shown as REST controller), that instead of plain TyphonQL queries, receives HTTP methods and JSON payloads representing entities. This servlet is hosted in the TyphonQL server and note that users of the polystore access its functionality only via the Polystore API, which acts as a reverse proxy.

In Figure 4 we can see, for example, that a JSON document with a name, price and manufacturer is sent to an endpoint `/Product` using the method POST. This is interpreted as follows:

- POST means creation of a resource
- The endpoint pattern `/Product` indicates the entity type
- The JSON payload represents the attributes and relations of the entity being created

This translation occurs in the so-called CRUD servlet, which delegates the diverse tasks to a set of Java classes that feature JAX-RS annotations. JAX-RS is the Java API for RESTful Web Services[5], an API designed with

[5]https://www.oracle.com/technical-resources/articles/java/jax-rs.html

the goal of simplifying the development of REST services in Java. Listing 2 shows an example of the class in charge of interpreting actions on endpoints of the form `/[EntityId]/[UUID]`, where EntityId corresponds to the type of the entity, and UUID the unique identifier of the entity instance we want to execute actions on. For a complete summary of the available endpoints and actions, please refer to Section 4.

```
import javax.ws.rs.DELETE;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
...
@Path("/{entityName}/{uuid}")
public class EntityResource extends TyphonDALResource {
  ...
  @DELETE
  public Response deleteEntity(@PathParam("entityName") String entityName,
      @PathParam("uuid") String uuid) throws IOException {
    String query = "delete␣" + entityName + "␣e␣where␣e.@id␣==␣#" +uuid;
    QLRestServer.RestArguments args = getRestArguments();
    CommandResult cr = getEngine().executeUpdate(args.xmi,
      args.databaseInfo, query);
    return Response.ok().build();
  }
}
```

Listing 2: Class representing operations on a Entity with JAX-RS annotations (extract)

In Listing 2 we can see that the method `deleteEntity` is annotated with `DELETE` which tells the framework that this method will be executed when this HTTP method is invoked. On the other hand, the class-level `Path` annotation ensures that this class will interpret actions on endpoints with the associated pattern. The body of the methods consists of the construction of the TyphonQL query based on all the parameters provided by the HTTP request. Note that the query is built up using purely dynamic information. If a user, for instance, would send a `DELETE` to a non-existing uuid, or even more, an entity type that does not exist in the model (e.g. `/NonExisting/`), the query will be built nonetheless and it will be the `executeUpdate` call on the TyphonQL engine the one that will fail, resulting in a failure reported to the client.

This fully dynamic nature makes the use of the generated OpenAPI specification (cf.3.1) rather important if guarantees about the conformance of the queries and endpoints to the right Typhon model are needed.

4 Overview of the Data Access Layer

This section describes the allowed CRUD operations on Typhon entities and how do they map to endpoints and HTTP methods.

4.1 Design considerations

- Endpoints begin with `/[Entity]/`, where `Entity` corresponds to the entity type.
- The entity representation is a JSON object whose fields correspond to the entity attributes and relations, including the unique identifier (UUID)[6], e.g. `{"@id":"#918df",`

[6]In the examples of this section as well as in the figures, we use abbreviated versions of UUIDs. The actual implementation uses full-blown UUIDs, such as #a398fb77-df76-3615-bdf5-7cd65fd0a7c5

`"name": "\"Bob\"", "age": "19", "country": "#a398f"}`. The entity type is implicitly specified by the URL pattern this representation is retrieved from/send to.

- N-ary relations are represented by a JSON array of strings, each of it corresponding to the related entity uuid, e.g. `{"users": [ "#918df", "#7ba36"]}`.
- 1-ary relations are represented by a JSON string corresponding to the related entity uuid, e.g. `{"user": "#918df"}`.
- Fields have to be encoded like they are in TyphonQL queries, so a string value should have nested quotes, and a UUID should be prefixed with a pound (#).
- To avoid ambiguitites, the payload in the case of an update operation is special. Replacements, additions and substractions of related entities participating in a N-ary relation are specified separately. Consider the associated reviews of a product. To add and remove relations: `{"reviews": { "add": ["#60bbc"], "remove": ["#7e4b6", "#f44c6"]}}` ; to replace relations: `{"reviews": { "set": ["#09228", "#77004"]}}`.
- The operation "list entities" receives query string parameters in which query clauses and the aggregation operators limit and sort can be specified, e.g. `/Product/where=price<150&limit=100`.

4.2 Operations

Table 1 lists the operations available on the Typhon REST DAL, together with the endpoint pattern, the HTTP method, and the HTTP status codes that they can return.

Operation	Endpoint	Method	Description	HTTP Status codes
List entities	/[Entity]	GET	List entities according to certain search criteria	200 Ok 500 Error
Create entity	/[Entity]	POST	Create a new entity	201 Created 500 Error
Get entity	/[Entity]/[uuid]	GET	Get the representation of an existing entity	200 Ok 404 Entity not found 500 Error
Update entity	/[Entity]/[uuid]	PATCH	Update an existing entity	200 Ok 404 Entity not found 500 Error
Delete entity	/[Entity]/[uuid]	DELETE	Delete an existing entity	200 Ok 404 Entity not found 500 Error

Table 1: Mapping of Entity CRUD operations to Endpoints and HTTP methods

5 Use scenarios

The are two main classes of usage scenarios of the REST-based DAL: using generic tools and building dedicated clients for a specific programming language. In this Section we discuss both.

5.1 Generic tools

There are several HTTP clients and REST tools that developers/end users can use to access REST-based services. Those tools can naturally be used to query the polystore.

Figures 5 and 6 show the results of retrieving a specific entity using two generic tools, the command-line HTTP client CURL[7] and the desktop-based application Postman[8].

```
pablos-mbp:~ pablo$ curl --user admin:admin1@ -G http://localhost:8080
/crud/Product/312f9128-5e04-3e09-bb4a-efef23627994
{"e.description":"\"Practical\"","e.name":"\"Laptop\"","e.price":"150"
,"@id":"#312f9128-5e04-3e09-bb4a-efef23627994","e.productionDate":"202
0-04-14","e.availabilityRegion":"POLYGON ((1 1, 4 1, 4 4, 1 4, 1 1))"}
pablos-mbp:~ pablo$
```

Figure 5: Screenshot of curl

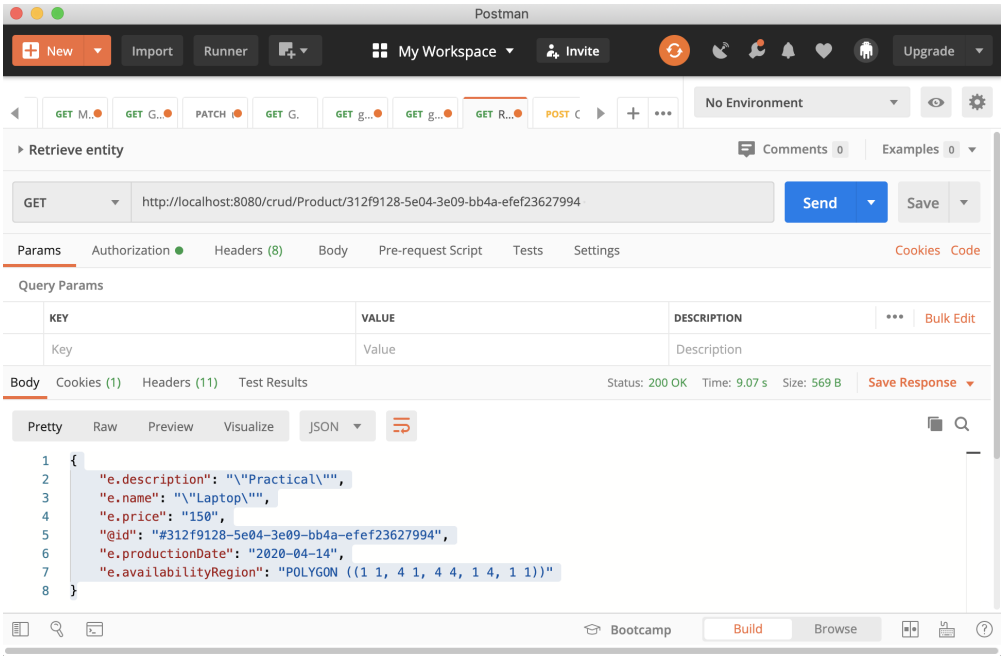

Figure 6: Screenshot of Postman

5.2 Generation of language-specific clients

To generate a Java client based on the OpenAPI specification derived from the TyphonML model, we utilized the OpenAPI generator tools [9]. Listing on Figure 7 shows the generated stub class for an entity defined by a TyphonML model, in this case, `Product`. Note that method `getProductUsingGET`, that gets the representation of a product in a programmatic way, performs the same communication behind the scenes as the one the generic clients needed to do in the screenshots from Figures 5 and 6.

[7] https://curl.haxx.se/
[8] https://www.postman.com/
[9] https://github.com/OpenAPITools/openapi-generator

Besides Java, the OpenAPI generator tools target a substantial array of diverse languages, to name a few: C++, C#, Scala, Ruby, etc. This flexibility was fundamental in choosing a REST service as our implementation technology for the DAL, as we discussed in Section 2.

```
public class ProductRestControllerApi {
    private ApiClient apiClient;

    public ProductRestControllerApi() {
        this(Configuration.getDefaultApiClient());
    }
    public ProductRestControllerApi(ApiClient apiClient) {
        this.apiClient = apiClient;
    }
    ...
    /**
     * getProduct
     *
     * @param id id (required)
     * @return Product
     * @throws ApiException If fail to call the API, e.g. server error or
     * cannot deserialize the response body
     */
    public Product getProductUsingGET(String id) throws ApiException {
        ApiResponse<Product> resp = getProductUsingGETWithHttpInfo(id);
        return resp.getData();
    }
    ...
}
```

Figure 7: Stub class for a Product as defined in a TyphonML model (extract)

6 Conclusion

This deliverable reported on Task 4.5, that is, the provision of a Data Access Layer to the Typhon ecosystem.

We evaluated two main alternatives: code generation of Java classes, and the implementation of a REST service that would translate high-level CRUD operations into TyphonQL queries. We chose the latter because of the benefits it provided in terms of decoupling from a particular infrastructure, and the ample availability of generic tools that can be used to profit from such technology.

The REST-based DAL consists of two main components: an OpenAPI schema generator, that takes a TyphonML specification and produces a corresponding OpenAPI description for that particular model; and a set of REST services (described by the OpenAPI schema) that interpret the HTTP requests sent by the client and executes CRUD actions accordingly. For the former, an Acceleo template was used, and for the latter, servlet technology using JAX-RS: the Java API for RESTful Web Services.

Thanks to the DAL, programmers can have programmatic access to the data available in Typhon polystores, by means of custom clients generated from an OpenAPI definition; and end-users can profit from generic tools that are available to communicate with REST-based services.

References

[1] Centrum Wiskunde & Informatica (CWI). D4.5 – TyphonQL Compilers and Interpeters, 2020.

[2] The University of L'Aquila. D2.4 – TyphonML Modelling Tools, 2019.

[3] The Open Group with contributions from all partners. Amendment technical annex, 2019.