An Interoperability Tool for Low-Code Development Platforms

Rita Clode Silva Jardim Fernandes
ritacsjfernandes@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2021

Abstract

The development of software systems commonly requires integration use cases, such as the data exchange between multiple tools. Interoperability is defined as the ability of multiple software intermediaries to exchange information so that a tool is able to handle the information generated by another. Over the last few years, Low-Code Development Platforms (LCDPs) have gained popularity, with a rising number of companies using them to build enterprise-grade apps and transform their businesses. The lack of interoperability will raise a common problem, since more and more client-side applications are moving from thick clients to thin web clients. We create a method to expose an OData service dynamically from a Low-Code Development Platform application, in order to be further consumed by other systems such as business intelligence tools. OData is a protocol that allows web clients to publish, query, and update data in data services using simple HTTP requests. From an initial data model obtained from an LCDP application, we derive all the artifacts required to have an OData service up and running on top of a database that conforms to the model definition, including the conversion of OData requests to SQL queries and compliance with the OData protocol. Our approach creates an API exposing the data retrieved from the LCDP application. The 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 as well as allowing it to be consumed by other applications.

Keywords: Open Data Protocol (OData), Data Integration, Low-Code Development Platforms, OutSystems

1. Introduction

The development of software systems commonly requires integration use cases, such as the data exchange between multiple tools. Interoperability is defined as the ability of multiple software intermediaries to exchange information so that a tool is able to handle the information generated by another. Since the representation of transferred data differs between tools, implementing an interoperability solution frequently calls for the use of syntactic and semantic mapping.

Low-code is a form of software development in which applications and processes are developed with little to no coding. A Low-Code Development Platform (LCDP) employs visual interfaces through simple logic and graphical features instead of sophisticated code languages. These platforms have become more popular as a cost-effective and time-saving alternative to traditional software development [10].

However, there is a lack of advanced functionalities within the LCDPs resulting in a need for interaction with other tools, such as data analytics and visualization tools. Such interaction frequently involves exposing web services, through Application Programming Interfaces (APIs), tools that help support interoperability. So as to develop a dynamic interoperability tool for Low-Code Development Platforms using the Open Data Protocol (OData), we followed the guidelines provided by the Design Science Research methodology [20].

2. Research Background

2.1. Low-code Development Platforms

Low-code development platforms (LCDPs) are simple visual environments that are increasingly introduced and promoted by major IT companies [23]. Such platforms help dealing with the shortage of highly-skilled software developers by en-

abling end users, with little to no programming experience, to contribute in software development processes. The most representative LCDPs are OutSystems [2], Mendix [6], Appian [1] and Kissflow [5].

LCDPs allow the development and deployment of fully functional software applications using powerful graphical User Interfaces (UIs) and visual abstractions requiring minimal or no procedural code [25]. They are frequently delivered on the cloud via a Platform as a Service (PaaS) model. PaaS is a cloud development and deployment environment that contains tools for building everything from simple cloud-based applications to sophisticated enterprise software enabled through the cloud [11]. PaaS helps avoiding the cost and complexity of purchasing software licenses, development tools, managing application infrastructure and other resources. Model-driven engineering techniques are used to design these fully functional applications, which take advantage of cloud infrastructures, automatic code generation and graphical abstractions. To ensure effective and efficient development, PaaS models are used alongside deployment and maintenance, and software design patterns and architectures.

2.2. Open Data Protocol

Microsoft presented the Open Data Protocol (OData) back in 2007. By 2012, OData had been proposed to OASIS, and in 2014 Version 4.0 was released by the international open standard consortium. As of 2020, OData Version 4.01 which is a highly compatible, incremental release over OData 4.0 was published.

OData is an HTTP protocol that allows web clients to use basic HTTP queries to publish, query, and update information in data services [18]. It enables you to develop resources that are specified by an Entity Data Model (EDM) and queryable by web clients through an SQL-like URL-based query language [16]. This query language has a range of query options that allows customers to exactly define the instance data they want. Simply described, OData is a standardized data transport format with a defined data access interface [13]. The data is serialized and sent via HTTP using the XML or JSON standards. The latter provide alternative data formats, both of which are supported in just about all web application technologies. The OData client ecosystem has grown over the previous few years to the point that client libraries are available for the main client devices and platforms, with more on the way [13]. The OData ecosystem is composed of service producers and service consumers. OData service producers use the OData protocol to expose their data, whereas OData consumers are simply applications that consume data exposed using the OData protocol. OData consumers can range in sophistication from a simple web browser to a custom application that exploits all of OData's features.

OData consists of the following four main parts [24]:

- OData protocol - The protocol specifies the way consumers can interact with data sources. Create, Read, Update and Delete (CRUD) operations along with XML and JSON serialization standards are supported. The query language includes a set of query parameters that enable customers to describe the data they want.
- OData data model - An abstract data model, the EDM, defines the data structure and provides a general mechanism to detail and arrange the data. It's an instance of an entity relationship model implementation, in which data is represented as entities and relationships between them. A Service Metadata Document is provided by an OData service, and it describes the service's EDM-based model in the XML-based Conceptual Schema Definition Language (CSDL).
- OData service - An OData service exposes a callable endpoint that is used for accessing data or calling functions. It employs the data model, implementing the OData protocol.
- OData client - An OData client uses the OData protocol and the corresponding OData data model to connect to an OData service.

The service document lists all the top-level feeds for users to access them, since a service may contain one or more feeds [18]. It helps service consumers to find the locations of the available resource collections, since the document lists the collections of available resources provided by the service. The service document is returned when making a get request on the service root URI. The service metadata document specifies its Entity Data Model, through the "$metadata" request [18]. The OData metadata document is the standard way to let end-users know how to query the data, as it presents information about the structure and organization of all the resources. The result is in CSDL format.

OData uses HTTP verbs (GET, PUT, POST,

Figure 1: URI components of the Open Data Protocol.

DELETE) to define actions on resources, and it uses a common URI syntax to identify those resources. The client must perform an HTTP POST, GET, PUT, or DELETE request to create, read, update, or delete an object, accordingly [14].

OData also defines a set of rules for producing URIs to identify the data and information given by an OData service [18]. The service root URI, the resource path, and the query options are the three main URI components, which are displayed in Fig. 1. The root of an OData service is identified by the service root URI [24].

The resource path specifies the resource with which the service consumers wants to interact with. It is mostly used to address a collection, an entity within a collection, an entity's attribute or a relationship.

Query options in the URL request allow you to influence how the service processes a request. To customize a request, OData offers a set of system query options. System query options are prefixed with the $ character (optional in OData 4.01). The most used query options are explained in Table 1.

3. Research Problem

The paradigm of distributed IT architectures is shifting away from monolithic programs running on a single node and moving towards distributed, dynamic environments [15]. Such environments enable the creation of applications by assembling existing services, increasing code reuse while reducing development time [15]. Ensuring the quality of data integration between systems and applications in these environments is essential [19]. Data integration refers to the transfer, replication, and transformation of data from from one application to another without regard for application or business logic.

Over the last few years, Low-Code Development Platforms have gained popularity, with a rising number of companies using them to build enterprise-grade apps and transform their business. According to Gartner [12], low-code application development will account for more than 65% of all app development functions by 2024, with 66% of large companies adopting at least four low-code platforms. The lack of interoperability will raise a common problem, since applications are changing from thick clients to thin web clients [15].

Instead of implementing complex integration frameworks for every enterprise-integrated application (e.g., Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), Supply Chain Management (SCM) systems), working towards a generic data integration method that allows interoperability among several applications should be our goal. This would allow interoperability between Low-Code Development Platforms and enterprise-integrated applications, Business Intelligence (BI) tools and other systems.

As an OASIS approved standard, OData is a viable alternative for open data exchange services [15]. Previous work has only focused on creating bridging applications or theoretical approaches for exposing OData services in a non automatic way.

Thus, the problem identified in this research is that there is a **lack of dynamic approaches to expose an OData Service from an LCDP to be further consumed by other applications**, without the support of an extra tool or a bridging application.

4. Proposal

This section describes the objectives of the solution and explain in detail our proposal.

4.1. Objectives

The main goal related to this research is creating a method to expose an OData service dynamically from a Low-Code Development Platform application, in order to be further consumed by other systems (OData consumers) such as Business Intelligence tools. For this to be achieved we defined the following objectives:

- allow complex queries against the exposed information;
- enable CRUD operations over the service data;
- provide the means to navigate through relationships between entities;
- ensuring the OData service can be consumed by an OData consumer;

Trying to accomplish these objectives, we had in mind keeping the cost of our solution as low as possible, without the support of extra tools or a bridging applications.

Query Option	Description
$top=n	The service returns the number of available items up to but not greater than the specified value n.
$skip=n	The service returns items starting at position n+1
$orderby=PropertyName	Specifies the property to order by the items returned from the service.
$count=true	Specifies that the total count of items within a collection matching the request be returned along with the result.
$filter=PropertyName eq Value	Restricts the set of items returned over one or more specified properties.
$select=PropertyName	Requests that the service return only the properties, dynamic properties, actions and functions explicitly requested.
$search=SearchExpression	Restricts the result to include only those items matching the specified search expression.
$expand=RelatedEntity	Indicates the related entities that must be represented inline.

Table 1: Most used query options.

4.2. Proposal Description

We generate all the artifacts required to have an OData service up and running from a data model retrieved from an LCDP application, through the translation of OData requests to SQL queries and compliance with the OData protocol [17]. Our approach creates an API exposing the data retrieved from the LCDP application. The model is exposed as an OData service, allowing end-users to obtain data using the OData query language in a simple way, and for the data to be consumed by other applications [16].

Developers should first specify their data models in the EDM format, then add business logic to resolve URLs using the OData query language to handle querying and modifying the data, and finally translate such queries into SQL statements [17]. Furthermore, to exchange messages with OData clients who follow the protocol, a de/serialization mechanism is necessary. Thus, the entire process of generating an OData service includes the following tasks:

1. The creation of the OData data model (EDM) from the received table structures.
2. The mapping between OData requests and SQL statements
3. The de/serialization process.

4.2.1 Creation of OData's EDM

The first step in defining an OData service is designing the entity model [22]. The main components of a database table are the table's name, primary key and a list of attributes which correspond to the table's columns. Each attribute has a name, a type, whether it is a primary key, whether it is foreign key referencing another table's primary key and if this stands the corresponding referenced primary key.

The Entity Data Model is a collection of concepts that describe the structure of data, regardless of how it is stored [4]. The entity type is the fundamental building block to express the structure of data. A group of instances of a particular entity type is referred to as an entity set. Each entity must have its own entity key within an entity set. Entity sets are all grouped in an entity container. Properties establish the structure and characteristics of entity types. A Product entity type, for example, might have properties like ProductId, Name and Price. A property can hold either primitive data (e.g., text, integer, or Boolean values) or structured data using complex types. Association types are used to describe relationships between two entity types. Every association has two association ends which correspond to the two related entity types and a multiplicity representing the maximum number of entities that can be at an association's end. A navigation property on an entity type is an optional property that allows users to navigate from one end of an association to the other end.

The mapping between a table structure and the corresponding EDM is the following:

- for each table an entity type is created, with the table's name, primary key as the entity's key;
- for each list of attributes from a table, properties are created within the corresponding entity, with the respective name and type;
- for each foreign key a navigation property is created, linking the source and target en-

tities and is stored in the respective entity type;
- each record is placed within the corresponding entity set;
- an entity container is created to store the entity sets;

4.2.2 Mapping between OData requests and SQL statements

To transform OData requests to SQL statements we consider:

- **HTTP method** - specifies if the request is either a query or a data modification action;
- **resource path** - identifies the resource to query or update (e.g., products, a single product, supplier of a product);
- **query options** - allows to specify the required instance data

We created a query model to perform the transformation of the target resource path into the corresponding SQL statement with the specified query options. The model has the following structure:

- **list of output tables** - the tables to select from;
- **list of join references** - a join reference is composed of the table and the two columns to join on;
- **list of selected columns** - the columns that are selected from the output and join tables;
- **list of where conditions** - a where condition is composed of unary or binary expressions that contain the column name, the operator and the value (e.g., Name = 'John'); a where condition can have more than one expression by using the logical operators *OR* and *AND* (e.g., Name = 'John' *AND* Age > 25);
- **list of orderby conditions** - an orderby condition is composed of the column name and the sort order (i.e., ascending or descending);
- **offset number** - the number of records to skip from the beginning;
- **next number** - the top *n* records to display;
- **count option** - a boolean value that indicates if the total number of records should be displayed;

We designed a method to perform the transformation of the target resource path into the corresponding SQL statement. To query a collection of entities, the name of the collection is added to to the list of output tables. Now, to get a specific entity within a collection besides adding the name of the collection to the list of output tables, one must specify a where condition of the primary key stated. A property of a particular entity is retrieved the same way as a single entity with the name of the property listed in the selected columns. Similarly, to navigate through a relationship of a specified entity we add the attributes and table as a joining reference and only specify the end of the relationship's columns.

OData requests are refined through query option, so we also had to take them into account in the mapping of the SQL statement. Query options *top* and *skip* specify the number of records to be included and excluded in the request result through the *offset* and *next* numbers. To order the OData payload a particular column and the sort order are added as orderby conditions. If no sort order is specified, our method assumes an ascending order. The inline *count* option is mapped into a boolean value and two SQL statements are run. Each filter condition is added to the where conditions list in our model. The selected columns are also added to a designated list. The *search* query option gets modelled into as much filter conditions using the like operator as columns exist in the target resource. Finally an expansion is similar to requesting a relationship between to entities, but instead of only exposing the end entity, the end entity is exposed within the source entity. This is done by adding the end entity to the join tables list.

4.2.3 De/Serialization Process

This process creates an OData serializer and deserializer that supports both the OData JSON and XML formats.

In order to construct the textual representation of the OData records according to the protocol's norms, the serializer applies a model-to-text transformation to the OData query result [17]. For example, an entity is represented by a JSON object representing its properties composed of list of key/value pairs, and an entity collection is transformed to a JSON array holding the entities. The serializer additionally takes into consideration the query model while generating the JSON representation, for example the number of key/value pairs correspond to the number of selected columns. Apart from the entity's properties, the JSON object also includes the annotation *odata.context* as metadata, which provides the payload's root context URL.

The deserializer parses and processes the body of OData requests POST and PUT to construct the details of the INSERT and UPDATE SQL queries [17]. In the resulting SQL statement,

each key/value pair in a JSON object is transformed to the corresponding field in the relevant database and its respective value. For DELETE requests, we only need the resource path component in the URL that targets a specific entity within a collection to be removed.

The XML representation format follows a similar procedure for the metadata document of the OData service.

5. Demonstration

5.1. Context

This master's thesis was implemented in a professional environment, integrated in a company dedicated to delivering digital solutions through developing cutting-edge enterprise software, *PhoenixDx* [8].

The Low-Code Developed Platform used for this validation was OutSystems [2]. OutSystems is a modern low-code application platform that speeds up the creation of applications while also providing exceptional flexibility and efficiency [2]. It enables the development of desktop and mobile applications which may run either in the cloud or on local infrastructures. OutSystems has three significant components:

- Integration Studio: allows database connections through either Java or .NET
- Service Studio: where the behaviour of the application being developed is specified
- Platform Server: the cloud server used to develop, orchestrating all runtime, deployment, and managing activities for all applications

In order to validate our OData service, we consumed it through PowerBI [3]. PowerBI allows to connect and visualize any data using the unified, scalable platform for self-service and enterprise Business Intelligence tool [3]. It is a tool that is easy to use and helps gaining a deeper data insight.

To the extent of our knowledge, OutSystems has no component to support OData.

5.2. Exposing an OData service in OutSystems

According to our proposed approach in Section 4, the integration process involves three steps, which are described in the following sections.

5.2.1 Creation of OData's EDM

To expose an OData service dynamically, we used OutSystems' metamodel. The metamodel specifies what can be found in the model, from the data migration point of view.

To expose an OData service, the OutSystems application must expose a REST service. Such application exposes a REST service with 3 main procedures exposing the service document, the metadata document, and result of any querying or modifying the data. The OData service endpoint of the application has a defined context, which contains the required configuration for the service to operate. The service context contains:

- ServiceRoot - the absolute URL of the Service Document
- EntityDefinitions - the definition of all entities, and their attributes, exposed by the service
- Body - the http request body for POST and PUT requests

Each OData request, in order to execute, needs access to the service context and to its inputs. The input for a given OData request is the path of the request, which is a possibly empty string starting after the service root and also including any query string of the request, and the body to insert or update data. The path is composed of the target resource path and query options. When the path is empty the service document is returned and the metadata document is returned when the path is *$metadata*.

Microsoft's OData Core library [7] is designed to read and write all kinds of OData payloads, such as service document, model metadata, entity set and references, etc. Through a .NET extension of our OutSystems application we are able create and read such payloads. To build the metadata XML, an EdmModel must be built first, through the OData Core library. In the OData library, the object that represents all of the entities exposed on an OData service is called EdmModel. This contains a list of entities, in a similar fashion to the list of entity definitions. Given the EdmModel and the service root, writing metadata is simple by using *ODataMessageWriter.WriteMetadataDocument()*. The service document can also be generated automatically. The Service Document and the Metadata Document, are requests simpler to execute since they only depend on the EdmModel and service root. Writing other payloads is a more complex process, which is further detailed in the following sections.

5.2.2 Mapping between OData requests and SQL statements

In order to execute a dynamic OData request we need to parse the path information. For this task, we used the ODataUriParser class provided by the OData library. The ODataUriParser provides

detailed information of the multiple segments of the given Path, for example which entity type it refers to, and if it has a key predicate. It also parses expressions related to the query options keywords such as $filter, $orderby, $top, $skip, etc. However, this information is not directly suitable to generate an SQL statement. To transform the output of ODataUriParser into SQL, we use a QueryParser class. This class first builds a QueryModel object, which contains information directly suitable to generate the SQL statement. This object contains information such as the list of selected columns, the list of tables in the FROM clause, the list of conditions on the WHERE clause, etc. Once obtained the QueryModel, transforming it into SQL is a simple operation of mostly concatenating strings. We defined a specific query model to help the mapping to SQL statements for each data transformation operation (i.e., create, read, update and delete).

For DELETE requests the query model contains an entity type and a key/value pair. The entity type is the entity table from which the record is to be deleted, and the key/value pair defines the primary key and the corresponding value to specify the record. These values are retrieved from the target resource segment of the URL path.

To insert a new record through the OData service, the query model is composed of an entity type and a list of key/value pairs. Again, the entity type is the entity table from which the new record is to be inserted. The list of key/value pairs have the name and value of all the attributes of that specific entity type. The entity type and the key/value pairs are specified in the HTTP request's body.

For PUT requests the query model is the similar to the insert model. However the key/value pairs are a subset of the entity type's attributes and corresponding values. In addition, there is a specific key/value pair in the model that refers to a specific record, through the primary key. Once more, the entity type and the key/value pairs are specified in the HTTP request's body.

After obtaining the SQL statement, we execute it using OutSystems' database API. Such API allows to retrieve an object of type IDataReader after executing the SQL command. This object, which defines the query result can then be used to read multiple rows, and multiple columns on each row.

5.2.3 De/Serialization Process

The serialization of the XML metadata document and the JSON service document were already described in the end of Section 5.2.1. For other types of OData payloads, we defined a specific output class to help the the serialization process for each data transformation operation (i.e., GET, POST, PUT, DELETE).

For updating and removing records, the payload has no content, therefore the serialization process only requires setting the status code of the response to 204 (indicating no content).

When querying the data exposed by the service, the output class receives the query model completed in the previous section and the data table with the resulting records of such query. To serialize into an OData payload, we used the ODataMessageWriter class provided by the OData library. The ODataMessageWriter is a writer class used to write all kinds of OData payloads. A collection of entities is described by the ODataResourceSet class and for each data row of the query result we define a list of ODataProperty class to describe the entity's properties. In case there are nested entities that were expanded using the $expand query option, an ODataResource is used to describe them. The status code is set to 200 (OK) upon a successful request.

For new data records being inserted, the output class only receives the query model completed in the previous section. From the model, the properties and respective values of the record are extracted and converted into ODataProperty and written through the ODataMessageWriter. The status code is set to 201 (created).

The deserializer parses and processes the body of OData requests to insert and update data, to build the details of query model. For this, we use the ODataMessageReader class which is provided by the OData library. The ODataMessageReader is a reader class used to read all OData payloads. It is created using the request's body and the body is parsed using an ODataReader to read a resource within the body and proceed to filling in the corresponding query model.

6. Evaluation

For evaluating the efficacy of our artifact, we demonstrate its response in several use cases [21]. As we were developing our OutSystems solution, Unit tests were incrementally added to assess our artifact in specific scenarios.

6.1. Test Application

In all of the following use cases, the test application we used was a project exposing the Northwind DB. The Northwind DB is a sample database that is used to exhibit the performance of Microsoft's products. The database contains sales information of a hypothetical company that exports/imports specialty foods called Northwind.

An application was created named Northwind, with all Northwind DB's entities. No authentication is required to access the service and it is live in `https://phoenixpressservicesptyltd-dev.outsystemscloud.com/CDS/rest/odata/Northwind/drafts/v2`.

6.2. Unit Tests

6.2.1 Service and Metadata Documents

Being able to expose the service and metadata documents is a requirement for any OData service. The service document is retrieved with a GET request on the service root URI. This JSON document lists the collections of available resources provided by the project's service. By appending the segment *$metadata* to the service root URI of a GET request the metadata document is retrieved. It describes each entity type of the project, listing their properties and corresponding data types.

6.2.2 CRUD Operations

One of our goals was to be able to manipulate the data exposed in the service using CRUD operations. First we decided to read the list of available shippers using a GET request on `https://phoenixpressservicesptyltd-dev.outsystemscloud.com/CDS/rest/odata/Northwind/drafts/v2/Shipper`.

For a shipping company to be added to the Shipper collection, the service client must send a POST request to that collection's URL (`https://phoenixpressservicesptyltd-dev.outsystemscloud.com/CDS/rest/odata/Northwind/drafts/v2/Shipper`). The POST body has to consist of a single valid entity representation. To test this, we used Postman. Postman is an API platform for both building and using APIs [9]. It allows you to send HTTP requests to web APIs, including simulating requests with bodies which is important for POST and PUT request methods. The request (and corresponding response) in Fig 2 creates a Shipper entity whose company name is *CTT*.

To update an entity within a collection, the service client must send a

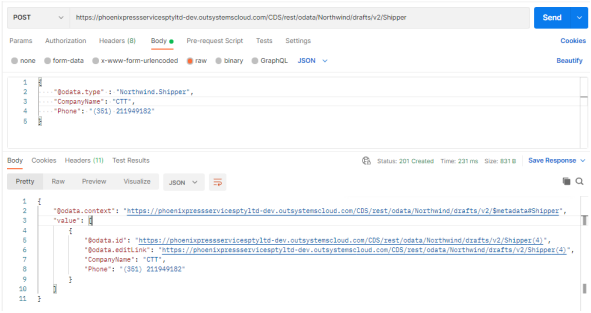

Figure 2: POST request and response of creating a new entity in Shipper collection, using Postman.

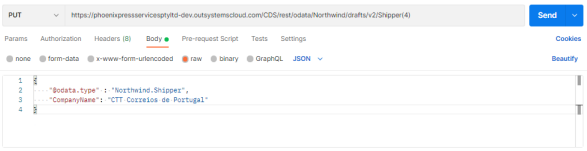

Figure 3: PUT request for updating a specific entity in Shipper collection, using Postman.

PUT request to the specific entity's URL (`https://phoenixpressservicesptyltd-dev.outsystemscloud.com/CDS/rest/odata/Northwind/drafts/v2/Shipper(4)`). The PUT body has to consist of a single valid entity representation, composed of the properties the client desires to update. To test this functionality, we a PUT request using Postman, represented in Fig. 3. PUT requests for OData services have no content responses.

To remove an entity within a collection, one must send a DELETE request with the primary key of the entity to be removed. DELETE requests for OData services have no content responses. The request bellow deletes the Shipper with id = 4.

DELETE `https://phoenixpressservicesptyltd-dev.outsystemscloud.com/CDS/rest/odata/Northwind/drafts/v2/Shipper(4)`

6.2.3 Querying Requests

All of the query options in Table 1 were tested for simple cases and composed two by two. The $filter query option was tested for different data types and data comparisons as well. We will instance a small portion of the unit tests over querying the data, as more than 100 tests were developed.

There are a total of 77 product instances in the previously created Northwind project. As default the products are ordered by id number. In order

to know the top 3 most expensive beverages we created a GET request with the following URL: `https://phoenixpressservicesptyltd-dev.outsystemscloud.com/CDS/rest/odata/Northwind/drafts/v2/Product?$filter=CategoryIdeq1&$orderby=UnitPricedesc&$top=3`. Out of the 8 defined categories, beverages entity has id=1 and to get the most expensive ones we ordered the results by UnitPrice descending, selecting only the top 3. To simplify the result, we only select three properties: the name of the product, the price per unit and the quantity per unit (by adding $select=ProductName,UnitPrice,QuantityPerUnit to the query options).

Navigating through relationships was also tested. Every OrderDetail entity has a corresponding product, which has a category associated with it. In order to get what is the category of the product from the order detail with id=2, the following request was made: `https://phoenixpressservicesptyltd-dev.outsystemscloud.com/CDS/rest/odata/Northwind/drafts/v2/OrderDetail(2)/Product/Category`. In this use case, two navigation properties were tested, from OrderDetail to Product and from Product to Category.

To query which products that were out of stock and awaiting order arrivals, by filtering both UnitsInStock and UnitsOnOrder properties, the following URL was requested `https://phoenixpressservicesptyltd-dev.outsystemscloud.com/CDS/rest/odata/Northwind/drafts/v2/Product?$filter=UnitsInStockeq0andUnitsOnOrdergt0`.

6.2.4 PowerBI Integration

Finally, to further validate our project's OData service and the reason why it was automatically generated, we consumed it through PowerBI. In order to complete this integration, when a PowerBI report is created the data source to get the data from has to be an OData feed. By inserting the project's service root URL we are able to load all the project's data into the report. After selecting the necessary authentication requirements, we are able to select all or a subset of the tables listed in the OData feed, as shown in Fig. 4. To ensure the data was loaded we created a simple report listing the number of orders per customer's country, shown in Fig. 5. By analyzing the report, USA had the most amount of customer orders, followed by Germany and Brazil.

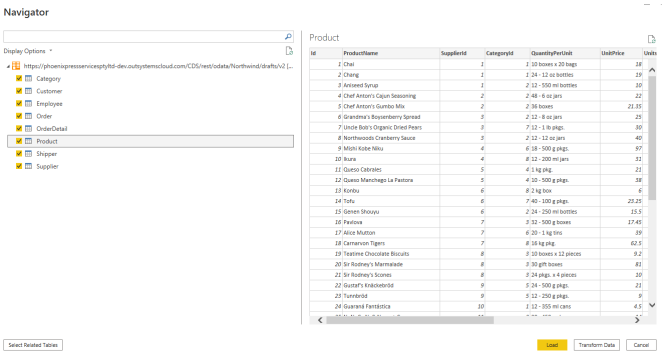

Figure 4: Getting OData feed data in a PowerBI report.

Figure 5: PowerBI report with the project's OData feed loaded data.

7. Contributions and Future Work

With the growing relevance of low-code platform applications in enterprise landscapes it is fundamental that its applicational data is made available and interoperable in the speed of low-code. This is a significant gap and as far we are concerned none of the most used LCDPs have a solution in their short to medium term roadmap. Such integration can be made possible with OData services.

We applied the Design Science Research Methodology to develop an artifact that would solve our research problem, stated in Section 3. Such artifact creates an API exposing the data retrieved from the LCDP application as an OData service. This allows end-users to use the OData query language to get the information they need in an easy and standard way as well as enables the data to be consumed by other applications. Our proposal was demonstrated using OutSystems applications and the PowerBI tool as the systems being integrated through OData services. To evaluate the efficacy of our artifact, we demonstrate its response in several use cases, checking that the stated objectives were accomplished.

Although the OData service is created automatically with our artifact, we still had to add a security layer in our demonstration, since OData by itself is not secured in a way that feels acceptable, especially being an open data initiative. Furthermore, we haven't tested the systems integration with large datasets.

As future work, to facilitate the design and creation of more sophisticated aspects, we want to extend our mapping features to capture further OData behavioral elements such as functions and actions. We intend to expand our approach to include all features of the advanced OData conformance level.

References

[1] Appian: Low-code automation | business apps | bpm | rpa. https://appian.com/. (Accessed on 09/2021).

[2] Build applications fast, right and for the future | outsystems. https://www.outsystems.com/. (Accessed on 09/2021).

[3] Data visualization | microsoft power bi. https://powerbi.microsoft.com/en-us/. (Accessed on 09/2021).

[4] Entity data model - ado.net | microsoft docs. https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/entity-data-model. (Accessed on 09/2021).

[5] Kissflow - a unified digital workplace | all in one platform. https://kissflow.com/. (Accessed on 09/2021).

[6] Low-code application development platform - build apps fast & efficiently | mendix. https://www.mendix.com/. (Accessed on 09/2021).

[7] Odata documentation - odata | microsoft docs. https://docs.microsoft.com/en-us/odata/. (Accessed on 09/2021).

[8] Phoenixdx - transform your ideas into business value. fast. https://phoenix-dx.com/. (Accessed on 09/2021).

[9] Postman api platform | sign up for free. https://www.postman.com/. (Accessed on 09/2021).

[10] What is low-code? a full guide to low-code platforms | creatio. https://www.creatio.com/page/low-code. (Accessed on 09/2021).

[11] What is paas? platform as a service | microsoft azure. https://azure.microsoft.com/en-us/overview/what-is-paas/. (Accessed on 09/2021).

[12] Outsystems to discuss the transformational impact of low-code at gartner application strategies & solutions summit | business wire. https://www.businesswire.com/news/home/20191204005674/en/OutSystems-to-Discuss-the-Transformational-Impact-of-Low-Code-at-Gartner-Application-Strategies-Solutions-Summit, 2019. (Accessed on 08/2021).

[13] S. Burgess. Open data protocol - build great experiences on any device with odata | microsoft docs. https://docs.microsoft.com/en-us/archive/msdn-magazine/2011/september/open-data-protocol-build-great-experiences-on-any-device-with-odata, 2011. (Accessed on 06/2021).

[14] M. J. Carey, N. Onose, and M. Petropoulos. Data services. *Communications of the ACM*, 55(6):86–97, 2012.

[15] R. Cupek and L. Huczala. Odata for service-oriented business applications: Comparative analysis of communication technologies for flexible service-oriented it architectures. In *2015 IEEE International Conference on Industrial Technology (ICIT)*, pages 1538–1543. IEEE, 2015.

[16] Ed-Douibi, Hamza, Izquierdo, J. L. Cánovas, and J. Cabot. Apicomposer: Data-driven composition of rest apis. In *European Conference on Service-Oriented and Cloud Computing*, pages 161–169. Springer, 2018.

[17] H. Ed-Douibi, J. L. C. Izquierdo, and J. Cabot. Model-driven development of odata services: An application to relational databases. In *2018 12th International Conference on Research Challenges in Information Science (RCIS)*, pages 1–12. IEEE, 2018.

[18] C.-Y. Huang and S. Liang. A sensor data mediator bridging the ogc sensor observation service (sos) and the oasis open data protocol (odata). *Annals of GIS*, 20(4):279–293, 2014.

[19] M. Muntean, C. Brândaş, and T. Cîrstea. Framework for a symmetric integration approach. *Symmetry*, 11(2):224, 2019.

[20] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[21] N. Prat, I. Comyn-Wattiau, and J. Akoka. Artifact evaluation in information systems design-science research-a holistic view. *PACIS*, 23:1–16, 2014.

[22] L. Ross. Odata - visualize streaming data the easy way with odata | microsoft docs. https://docs.microsoft.com/en-us/archive/msdn-magazine/2015/april/odata-visualize-streaming-data-the-easy-way-with-odata, 2015. (Accessed on 06/2021).

[23] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178. IEEE, 2020.

[24] M. Thoma, T. Kakantousis, and T. Braun. Rest-based sensor networks with odata. In *2014 11th Annual Conference on Wireless On-demand Network Systems and Services (WONS)*, pages 33–40. IEEE, 2014.

[25] R. Waszkowski. Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine*, 52(10):376–381, 2019.