DEMO Model based Rapid REST API Management in a low code platform

David Aveiro [1,2,3] and Valentim Caires [1,2]

[1] *ARDITI - Regional Agency for the Development of Research, Technology and Innovation, 9020-105 Funchal, Portugal*
[2] *NOVA-LINCS, Universidade NOVA de Lisboa, Campus da Caparica, 2829-516 Caparica, Portugal*
[3] *Faculty of Exact Sciences and Engineering, University of Madeira, Caminho da Penteada 9020-105 Funchal, Portugal*

Abstract

The development of APIs for integration of different information systems can lead to repetitive work, amounting to simple translation of variables and methods, taking into account the implemented systems. Low-code platforms increase the opportunity of automatic/rapid generation of APIs, both for incoming and outgoing data and/or service actions. Based on the DEMO data models of a low-code information system, we aim to easily create endpoints for providing simple lists of data items or even the results of complex queries or operations, by simple drag and drop operations in a friendly GUI, as well as for enacting internal tasks on the system. Likewise, we can also automatically scan provided data by external APIs that our system can call and match with internal data, also in a friendly GUI, facilitating the integration of external information into our local system.

Keywords

rest api, enterprise engineering, DEMO, api management, low-code platform, EBNF grammar

1. Introduction

The use of REST APIs as the backend web service, is an increasingly popular approach for data management in enterprises. APIs (Application Programming Interface) provide programmatic access to service and/or data within an application or a database. REST (Representational State Transfer) is an architectural style and approach to communications often used in web services development and follows a few architectural constraints.

Backend software development is a demanding task that can be quite time consuming, for example, handling data integrity, confidentiality, availability and privacy as well as properly handling hundreds of requests/tasks simultaneously. Also, implementing such services requires major knowledge and qualifications on the topic.

In the context of a larger research project called Direct Information Systems Modeller and Executer (DISME) [1], in this paper we focus on the problem of how to achieve rapid and/or automatic generation and management of ingoing and outgoing REST interfaces based on DEMO models and external endpoints through a low-code platform. Furthermore, achieving this in a way that allows managers or individuals in a comparable position in organizations to generate/manage REST API endpoints, even if they have little or no prior programming experience.

Low-code platforms increase the opportunity of automatic/rapid generation of APIs, both for incoming and outgoing data and/or service actions. Using data models of a low-code information system, it should be possible to easily create endpoints for providing simple lists of data items or even the results of complex queries or operations, by simple drag and drop operations in a friendly GUI, as well as for enacting internal tasks on the local system based on calls made by external systems. Likewise, we can also automatically scan provided data by external APIs that our system can call and

CIAO! Doctoral Consortium, EEWC Forum 2022, November 2022, Leusden, The Netherlands
EMAIL: daveiro@uma.pt (A. 1); valentim.caires@arditi.pt (A. 2)
ORCID: 0000-0001-6453-3648 (A. 1); 0000-0002-0871-7212 (A. 2)

match with internal data and rules, also in a friendly GUI. The aim is to facilitate the integration of external information systems with thelocal system, based in a low-code approach.

This paper presents our approach for a Rapid REST API Management (RRAM) based on DEMO Models [1]. One of DISME's components uses Blockly[2] that, following a formal Extended Backus-Naur Form (EBNF) grammar, allows users to easily configure business rules for implementing internal business logic. Blockly is a library that adds a visual code editor to web and mobile applications. The Blockly editor uses interlocking, graphical blocks to represent code concepts like variables, logical expressions, loops, and more. It allows users to apply programming principles without having to worry about syntax or the intimidation of a blinking cursor on the command line1. We aim to take advantage of our experience with Blockly and develop a new component of DISME which will allow us to manage ingoing and outgoing API interfaces by using a user-friendly GUI. And not only configuration and data format/attributes of the APIs might be generated in a (semi-)automatic fashion, but also their documentation might be automatically generated, using Swagger UI[3].

The following section presents the literature review on this topic, section 3 briefly explains the research on DEMO's models. In section 4, our low-code platform approach for RRAM is presented. Finally, section 5, compiles the most important points and future work.

2. Literature review and related work

In this section we review some of the existing research work related to the topic of REST API generation, management and documentation.

Some approaches used code generation. Wang, et al. [2], presents a model-based approach to automatically generate code for common operations of database access, and then wrap it into RESTFul APIs, which minimizes efforts and improves flexibility and reusability. However, generating code may negatively impact the development process, because it requires huge initial efforts, loss of flexibility, code rigidity and increased technical complexity.

Other approaches suggest following Model-driven Engineering (MDE) [3, 4], an iterative and incremental software development process. Mora-Segura, et al. [3], presents a solution based on multi-level modeling to represent domain knowledge. It supports on demand data loading through domain injectors, which are described through semantic-rich query descriptions.

Hussein, et al. [5], developed a solution called REST API Automatic Generation (RAAG), based on an integrated framework that abstracts layers for REST APIs, business logic, data access, and model operations. This solution was developed applying principles, standards, and patterns focusing on re-usability, maintainability, scalability and performance. A preliminary evaluation of the RAAG solution showed very promising results in development time, which was significantly reduced compared to traditional REST API implementations, regardless of whether they are experienced or non-experienced developers. On average, the time spent with RAAG was around half the time spent using traditional technologies/frameworks. Also, a survey was presented to the participants who used the RAAG solution and the opinions on the framework quality were very positive, especially regarding ease-of-use, maintainability and productivity.

Overeem, M. et al. [6] assessed the application programming interface management maturity of four major low-code development platforms' (LCDPs). One of the identified challenges in the evaluated LCDPs regarding API management was the ability to achieve the goal of making these reachable by those without formal software engineering, i.e. inexperienced users. The intrinsic simplification necessary to develop LCDPs and the complexity of software solutions are in constant confrontation with one another, and a middle term must be met so that users without software engineering experience can really use these platforms.

Brajesh De [7] states that REST API's adoption success depends on its documentation. It is important to create APIs with user-friendly interfaces for consumers, so API's documentation ought to make it simple for developers to get a grasp of its features and get started using it easily.

In [8] we find an algorithm for the generation of microservices expressed according to the OpenAPI standard, from the ontological model of an enterprise, that is stable by nature. However the proposed

[2] https://developers.google.com/blockly
[3] https://swagger.io/tools/swagger-ui/

approach generates services for all transaction acts which seems excessive to us. Only some acts will need to be made accessible in an API. Also they were confronted with the limitations of implementation issues such as the need to define value types and DELETE operations. From our perspective, we need to change the approach that ontologically, information/data can never be changed or deleted. Ontologically we agree that facts can never be changed or deleted; but for a particular implementation of a system, we need to have that possibility (e.g. due to European GDPR restrictions). Our proposed approach will solve these limitations.

Although several studies about REST API generation and management have been made, some of them are based on complex tools or require some kind of experience/training in order to take advantage of its full potential, some do not support different types of databases or just allow data retrieval and others focus on code generation [2] which we consider is far from ideal. We also haven't found any approach that includes the generation of the corresponding API documentation. Nevertheless, judging by the papers that have evaluated the effectiveness of their solution [2, 4] we can conclude that (semi-)automatic generation of REST APIs is a viable option and, if done properly, can turn the process into a much simpler task that can more easily be performed by people with little to no programming experience. An equally important insight gained from this literature review was the importance of an API's documentation, one that should be user-friendly so that experienced and inexperienced users can understand its characteristics for better usage.

3. DEMO Models

In order to situate the innovation of our approach, we will start by briefly introducing DEMO Models which are the base for our approach of automatic API generation. We will be introducing a recent evolution of these models presented in [9, 10], which were deemed to be more user friendly than the traditional approach [11, 12].

3.1. Fact Model

The Fact Model (FM) [9] of an organization is a model of its organizational products, in systemic terms, it is a specification of the state space and the transition space of the production world [13]. One of the artifacts that compose the FM is the Fact Diagram (FD) which can have two different views: the Concept and Relationships Diagram (CRD) and the Concept Attribute Diagram (CAD), explained below.

Regarding the CRD, arrows are used to express relationships, which will always consist, in practice, of an attribute in one concept whose instances will be a reference to instances of the other concept. In Figure 1 an example of the CRD is presented, showing the models developed for a construction licensing process [9]. The dark filled circle attached to a concept in one connector means that an instance of this concept, in order to exist, depends on an instance of the concept at the other end of the connector. Considering the example in Figure 1, an instance of Application Deliverable cannot exist without a reference to an existing instance of Type of Application Deliverable.

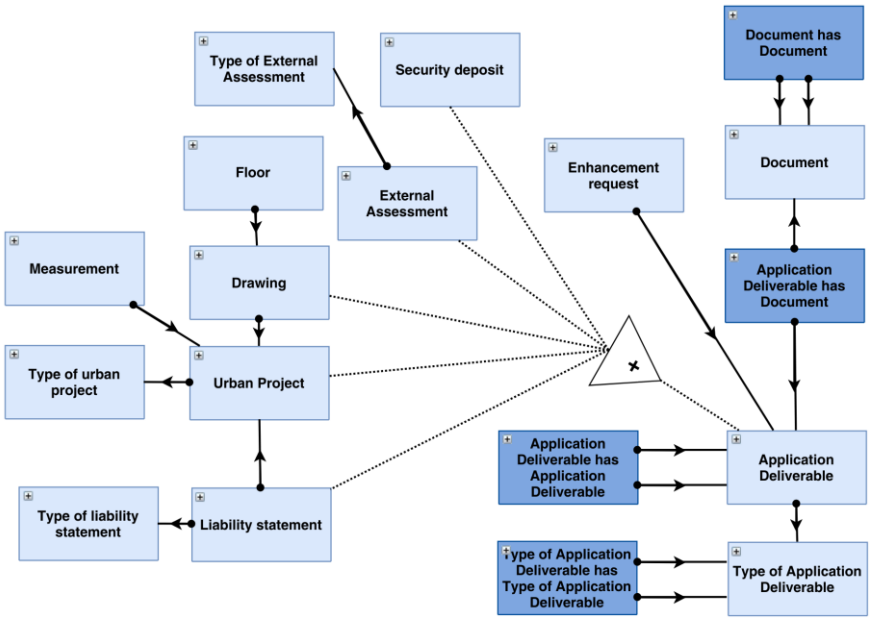

Figure 1: Concept and Relationships Diagram

As far as binary fact types are concerned, they are essentially of three types: one-to-one, many-to-one, and many-to-many, the following standard was adopted for each of these cases (with examples from a project case from a town council):

1) one-to-one relationships are represented by a connector with two arrow symbols in the middle, e.g., an urban operation can have one (and only one) permit and a permit is associated with one (and only one) urban operation (note: this example is not shown in Figure 1 for space reasons, but was taken from another part of the project case);

2) many-to-one relationships are represented by a connector with only one arrow pointing to the side "one" of the relationship, e.g., an instance of Application Deliverable has one (and only one) Type of Application Deliverable. However, the other way around, a Type of Application Deliverable can be associated with multiple (potentially zero) instances of Application Deliverable as shown in Figure 1;

3) many-to-many relationships are represented with an intermediate concept depicted with a darker color; this concept will have many-to-one relationships with the concepts participating in this many-to-many relationship; both of these relationships will have dependency laws on the side of this intermediate concept; e.g., an application deliverable can have one or more documents associated with it; and one document can be associated with one or more application deliverables as shown in Figure 1.

The specialization/generalization relationship is represented by using a connector with a pointed line (e.g., the specialization of Application Deliverable into its several more specific concepts). In practice, this specialization implies that a series of one-to-one relationships exist between the more specific concepts and the higher order concepts, with a dependency law on the side of every specific concept.

With regards to the CAD, it can be considered a variation or "expansion" of the CRD presented previously. In Figure 2 an example of the CAD is presented. With the main concepts and relationships known, one also needs to identify the attributes that belong to each concept, considering that at least one transaction in the enterprise's processes has to create values for it. Also, the ability to inspect which attributes each concept possesses is highly useful.

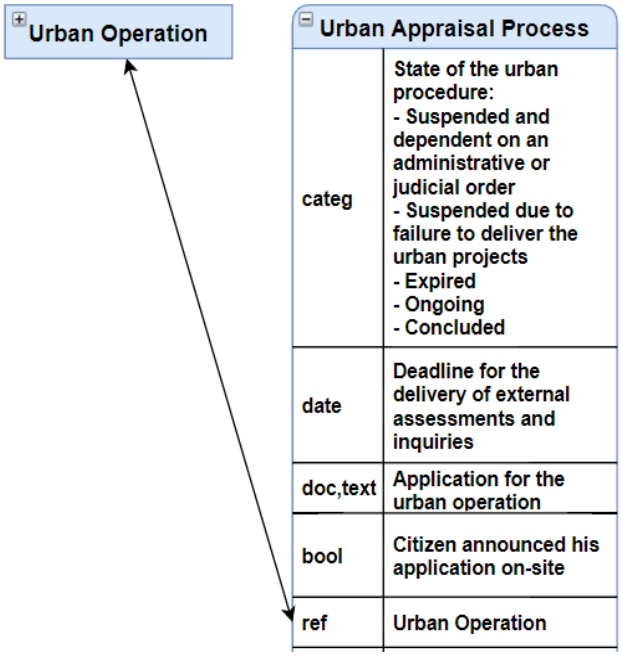

Figure 2: Concept Attribute Diagram

In the CAD, a concept is represented by a collapsible box whose expansion discloses its attributes, one per line. The value type of an attribute is specified to the left of the line, whilst to the right, the name of the attribute and eventually a list of possible values, usually for categorical value types.

3.2. Action Model

The operation of an organization is addressed in the DEMO methodology through its Action Model [11]. For each of the transaction coordination facts of the transaction pattern [9], an action rule can be produced, which specifies the guidelines that the actors must comply with whilst fulfilling their respective business roles. Actors are still allowed to deviate from expected behavior and autonomously decide and do the work from their agenda based on their professional and general knowledge [11]. The specification of these rules (see Figure 3) can be done in a visual programming language, thus allowing a more effective participation of non-technically savvy collaborators.

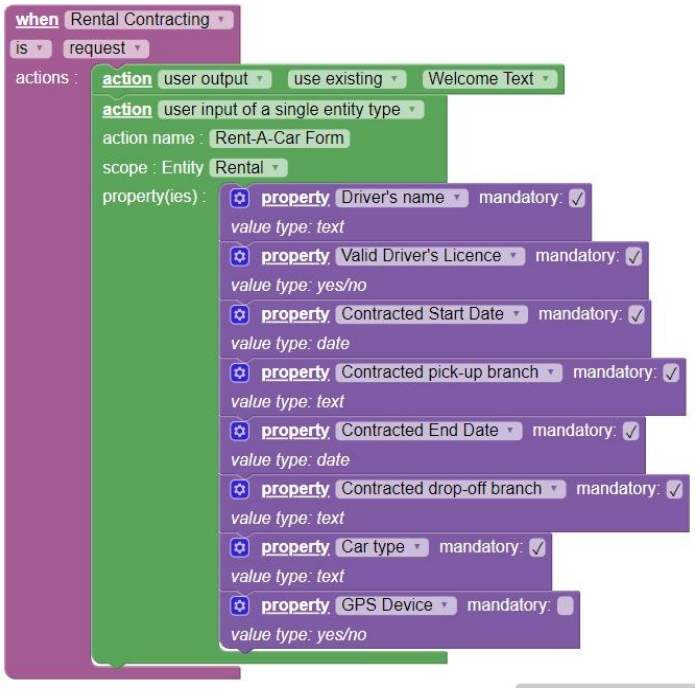

Figure 3: Action rule for the request of "Rental Contracting" transaction

Action Rule Specification (ARS) languages have evolved through time, starting with a pseudo-algorithmic language [12] and culminating, in DEMO's specification language 4.5, in a definition which adheres to the EBNF, the international standard syntactic meta language, defined in ISO/IEC 14977 [14]. In it's latest version, an ARS is tripartite: - The event part specifies which coordination events are responded to by means of a set clauses; - The assess part, by being based on Habermas' theory of communicative action, holds a set of validity claims that must be determined to be true with respect to the rightness, sincerity and truth conditions of the world; - And the final part, the response, which consists of a mandatory if clause that specifies what action has to be taken if advancing is considered to be justifiable, and, otherwise, in an optional else clause, the possible accountable actions that can be taken by the executing actor if he autonomously deems an exceptional situation to be justifiable [11].

Andrade et al [10], proposed an alternative ARS language for the Action Model, also in EBNF, whilst advocating its suitability for the implementation of an action rule engine and the execution of action models (and its ARSs) in a live production setting. According to this approach, as in DEMO's latest AM version, the execution of an AR, for a particular transaction state, is triggered by its corresponding coordination event (e.g. request, promise, etc) and multiple other actions may follow by means of causal links. However, by also considering expressions, logical conditions, validations, input forms, and templated-document outputs in the EBNF, which are constructs closer to an actual information system, it is argued that the alternative ARS language circumvents the unnecessary faults, complexities and ambiguities introduced by the so-called "structured english" sentences of DEMO's tripartite claim-based syntax [10].

We take use of this approach to allow concrete actions in the internal system, invoked through REST API endpoints made available to external systems.

4. Our low-code platform approach

Our approach for Rapid REST API Management (RRAM) is based on taking advantage of already existing DEMO Models in a local system and available external endpoints. Our goal is to add the functionality of rapid and/or (semi-)automatic REST API generation and handling to DISME which is currently being developed. This system already has features that are very helpful to the achievement of this goal, such as action rule specification and complex query modeling via user-friendly graphical user

interfaces. These were developed using Blockly and jQuery QueryBuilder[4], respectively, and are a good basis for the implementation of this new feature. Alongside the extension of these already implemented components, which will use existing information, like modeled business facts and attributes, action rules and queries, an additional component will be created to handle REST API endpoints modeling and specification. The component to be used, between the ones just mentioned, will depend on the type of operation the user wishes to perform.

To solve our research problem, we envision four main types of functionalities that should be developed to allow integration with other systems using REST APIs. These are described in greater detail in the following subsections, with them being the following: 1. simple CRUD (Create, Read, Update and Delete) operations on data from the local DISME system, 2. query based data provision by the DISME, 3. matching internal action rules and respective local data with data that needs to be fetched from external systems in their respective endpoints and 4. matching local endpoints provided to external systems with internal action rules. In this section, we demonstrate and validate our contribution using the EU-rent case from [15].

An API endpoint's information, including HTTP method, parameters and response fields, will be inferred from its specification in our system after the needed information has been selected/provided. This information will be used not only for specifying the endpoint itself, but also to automatically generate API documentation, for which we will use Swagger UI[5].

4.1. Direct Information Systems Modeller and Executer

As mentioned, we intend to develop this new component for rapid REST API management as part of the open-source low-code platform DISME, currently being developed. DISME is primarily composed of 3 components: 1) a Diagram Editor to create the higher level DEMO models in a graphical way 2) the System Manager to precisely detail and parametrize all DEMO Models, with a special attention to the Action Model, so that a complete information system can be specified according to an organization's demands; and 3) The System Executer to directly run the modeled information system in production mode.

In the System Manager, one or more users assume the administrator role and have the ability to modify each organizational process by creating and editing transactions, their relations, action rules and input forms that are associated with these transactions, in specific transactions steps, as well as by specifying entity and property types, that is, the main business objects and their attributes, or, in other words, the database of the information system. Users who model the system just need a basic understanding of enterprise engineering modeling, which is similar to the "language/representation" used within businesses, rather than requiring specific programming skills.

Users who have been granted authorization to participate in transactions in the System Executor do so in accordance with their roles and following DEMO's transaction pattern. The System Executor can be broken down into two main components: 1) the Dashboard, which serves as the user interface for users to interact with when performing organizational tasks, and 2) the Execution Engine, which controls the information and process flow in accordance with the full specification of the system.

4.2. Simple CRUD operations

Supporting the generation of API endpoints to simple CRUD operations consists of creating endpoints for Create, Read, Update and Delete operations for each of the DISME's internal concept/entity types that we wish to allow. It is pretty straightforward to generate corresponding endpoints for these four basic operations, as the only thing that changes in them is the request's targeting entity type. They are also regarded as simple because each operation only involves one concept from the system, lowering its complexity.

An interface will be developed within DISME, which will allow the user to choose the information that will be made available in the API. It will be possible to select all or a subset of an entity type's

[4] https://querybuilder.js.org/
[5] https://swagger.io/tools/swagger-ui/

properties/attributes for a particular CRUD operation, for instance. Such selections will be saved in specific DISME database tables and the respective endpoints will be automatically generated.

4.3. Query based data providing

In case a simple CRUD operation isn't enough, there is the option of associating a complex query to a certain endpoint. We have developed a component in DISME which allows the configuration of complex queries using drag and drop actions that works in the form of specification of queries and their filters, based on triplets of property-operator-value, chosen by the user selecting the relevant options in a user-friendly graphical interface and without the need of any programming experience. An example of a query configured using this component is shown in Figure 4.

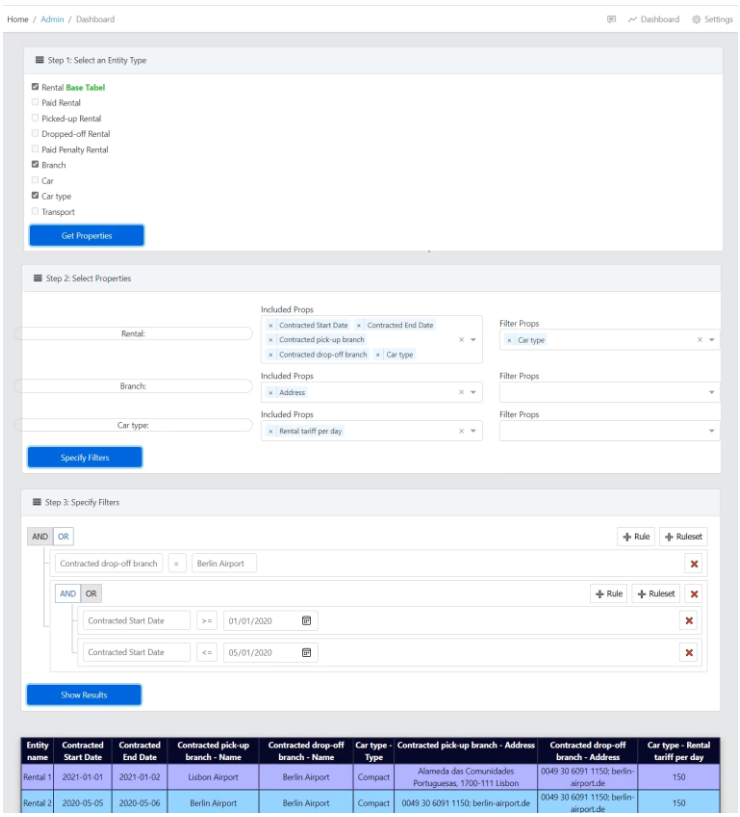

Figure 4: Complex query creation component from DISME

In the first step of this component, we have to select the entity types that our search will be focused on, with the first selected entity type serving as the Base Table, that is, the entity type where we will be searching for entity values based on the filters defined ahead. After having established the entity types that will be included in the search, it is now time for step two, where the query's properties are defined. Here, for each entity type selected, we will have two select boxes to specify the 20 properties to be included in the result as well as the properties to be used in filters. For each one, the options shown in the select box are the properties belonging to that entity type. The Filter Properties are the ones that will be used in the specification of triplets of property-operator-value. The properties selected in these select boxes will then be available for specifying filters in step three, and in case there are no properties selected, the user can still choose to see the results' table, bearing in mind that there will be no filters applied to any properties and consequently the results will include every entity of the Base Table's entity type present in the database. Finally, in step three, we can define rules and rulesets that will be applied to the main query to be run in the database. We can look at it as rules being conditions and rulesets being sub-conditions. Whether one or another is selected, we have to choose its type - and/or. We must also define the property to be filtered, the query's operator and the value that will restrict the result.

As already designed queries have their result properties clearly specified, the respective configuration of an API endpoint whose response perfectly matches (part of) the result of the query is allowed. That is, the list of the properties related to that query can be shown to the user who then will be able to select the relevant properties that should be part of the response of the API call. The same reasoning applies to the filters. One or more of them can be specified as parameters of the query, with no pre-filled value and with the value being assigned in run-time, for example, when an endpoint is called with certain parameter values. Note that DISME's database has tables with names: entity type and property for specification of the data model of an enterprise which are still used in the query management interface. But in DISME's user interface we will only use the names: concept and attribute, from the adapted DEMO's FM meta-model, which correspond to entity type and property, respectively.

4.4. Matching internal action rules with external endpoints

DISME also aims to support external calls to third party APIs, allowing the local system to both receive external information relevant for its functions, as well as actively supplying information or triggering actions in external systems. With this, a number of challenges arise, such as information in external systems being most likely structured in a way that needs to be matched and/or slightly adapted to local information. An internal action rule, after making a request to an external API, might process the received response and properly update some properties/instances in the internal system. If we retrieve meteorological information from an external API, for example, it is very likely that 1) the response we receive has more attributes than we use/need in our system and 2) the same attributes have different names in each system.

This type of operation will be supported through the extension of the Action Rules Management component that implements a user-friendly visual programming editor, through Blockly, for the specification of an organization's action rules, with a new action type named 'external api call', as can be seen in Table 2. The Blockly block that will allow us to specify an external get (call action) in this component will mutate into a block where we can specify an external API endpoint, with it automatically parsing the possible attributes of the response given by that endpoint and adding them to an internal list. If the desired action to perform when fetching the data is to create entities in our system, we can then attach, internally to this block, an entity input block, where we specify that a particular instance of an entity type will be created - selecting the entity type in a dropdown menu. Afterwards, blocks that specify the matching that must be done between the entity's properties and the API call response's properties (out of the internal list mentioned previously), whose value will be assigned to the property mentioned beforehand, are attached to this entity input block. Blockly can automatically validate if the value type of the internal property is compatible with the value type of the selected property of the response and not allow incompatible matches/selections. Similarly, an action can be defined so that it is possible to update an entity that is already in the system. For this, the user needs to specify the corresponding entity type and establish an incoming api call's parameter for the stipulation of the entity to be updated. An example of an external get call could be our local DISME requesting, to an external system of a partner car rental company, a list of all available cars, where the response would include properties like: car type, car model, number of doors, if it has AC, car engine capacity, etc. and we would only need the first two properties for the purpose of this call. This external api call action type definition example using DISME's blockly component is shown in Figure 5.

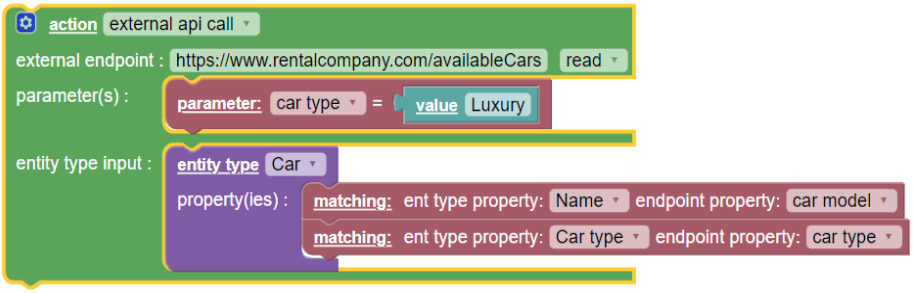

Figure 5: External_api_call action type definition example using DISME's Blockly component

This external api call block also includes a slot to specify possible parameters for the call. A similar reasoning applies to the parameters as the one applied to the response properties. Blockly would automatically parse the possible parameters of the external endpoint, and include them in an internal list. Then one could select a certain parameter (e.g. a particular Car Type) to be sent in the call and assign a value to the parameter: either a constant, the value of some property in the scope of the process running such a call, or a free value. Following the example above in Figure 5, after executing the external api call we would receive a response with a list of cars, only of the particular Luxury Car Type, in order to add them to our system for later use (e.g., showing to a potential client in a user output). While performing the action of creating entities in our system, endpoint properties would automatically be matched with the corresponding internal properties, as specified in the Blockly action.

The cases of external post, put and delete api calls are a simplification of the previous case, where only the parameters apply.

4.5. Matching local endpoints with internal action rules

Another type of operation supported through the extension of the Action Rules Management component is the creation of endpoints that can lead to the execution of internal action rules. This association of an endpoint with an internal action rule will be done through the introduction of the property: action rule execution type which doesn't currently exist in the DISME. It will have as possible values: native execution, which is the normal execution of action rules in the local system through user interaction in the prototype's dashboard; and local endpoint call execution, that is, the execution of local actions invoked by an external system's call to a locally configured api endpoint, as can be seen in Table 1's EBNF.

In order to accomplish this, we must first create a new action rule (with local endpoint call type) that will be responsible for processing the execution of the desired internal action rule, as well as defining the required parameters and a response type/format for it.

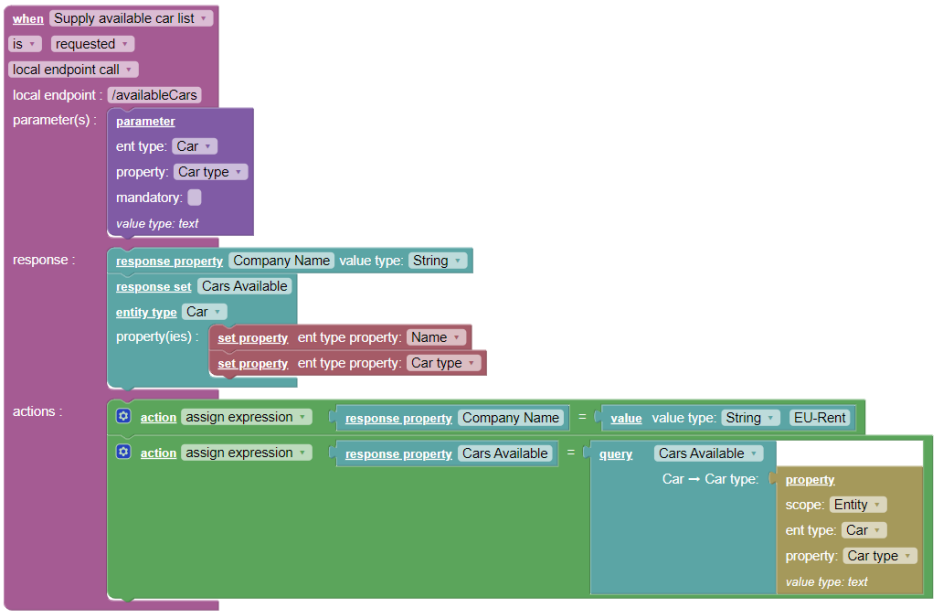

Figure 6: Definition example of an action rule with execution type 'Local endpoint call' using DISME's Blockly component

This definition of a 'local endpoint call' execution type action rule through DISME's Blockly component can be seen in Figure 6. For this example, let's look at the opposite of the one given in Figure 5. In this case, the goal is to enable an external system of a partner car rental company requesting a list of all available cars in our system. Thus, an action rule, with the 'local endpoint call' execution type has to be defined for the transaction type 'Supply available car lists' in order to create and link the

endpoint that will allow the system to accept this kind of requests. In this action rule, an optional parameter is defined so that incoming requests for available cars can explicitly define that only cars of that type should be checked for availability. The request's response must also be specified, being that one can include solo properties or a set of them in it, with them being assigned by the system in the actions section of the action rule.

Then, based on the configurations specified in the action rule, the standard information for making the endpoint available can be automatically deducted: HTTP method, URI, parameters and response.

4.6. Extending the Action Meta Model's EBNF grammar

Another contribution of this paper is the proposition of extending the current DISME's action rules' EBNF grammar. The changes introduced in this subsection are directly related to subsections 4.3. and 4.4., as the api call's operation types specified in them are the ones that use action rule specification to achieve its goal.

The current grammar foresees the action of calling external APIs, with the action type 'external call', but didn't detail it. By the extension observed with subsection 4.3., a proposition for its definition is presented in Table 2, and in Table 1 is the introduction and respective definition of an execution type to action rules, influenced by subsection 4.4.'s new operation.

In the tables displayed below, only new/updated concepts on DISME's action rule's EBNF grammar are specified, and soon after explained, as presenting the entire grammar definition would prove to be space-intensive and wouldn't add much value to the topic in discussion. Rows that have updated definitions of already existing concepts have those respective updates in bold lettering, and every other row is a new entry in the EBNF grammar.

Table 1
Action rule EBNF table with added/updated concept specification for internal api calls

Term	Rule
when	WHEN transaction_type IS|HAS-BEEN transaction_state **execution_type** { action } -
execution_type	NATIVE_EXECUTION | local_endpoint_call
local_endpoint_call	local_endpoint call_action { local_endpoint_parameter } - { response_property | response_set } -
local_endpoint	STRING
local_endpoint_parameter	property [MANDATORY]
response_property	value_type STRING
response_set	STRING entity_type { property } -

An action rule occurs in the context of a transaction type, among those specified in the system, in the activation of a particular transaction state. Our new proposition extends this definition of action rule to include an execution type that is divided into: native execution, that is,or the normal execution of action rules in the system through user interaction in the prototype's dashboard; and local endpoint call execution -, that is, the execution through an external system's api call of the defined system endpoint, as stated before. When defining an action rule with the 'local endpoint call' execution type, as is illustrated in Figure 6, one must specify the local endpoint with which external systems will interact, followed by the api call's parameters - mandatory or not system properties. Finally, one needs to define the call's response, with the chance to explicit solo properties, by defining their name and value type, and/or a set of them, by also defining its name firstly, but then choosing an entity type from the system and selecting properties from it.

Table 2
Action rule EBNF table with added/updated concept specification for external api calls

Term	Rule

action	causal_link | assign_expression | user_input | edit_entity_instance | user_output | produce_doc | if | **external_api_call**
external_api_call	external_endpoint_url call_action { api_call_parameter } api_call_entity_input
call_action	CREATE | READ | UPDATE | DELETE
external_endpoint_url	STRING
api_call_parameter	external_endpoint_parameter "=" (property | constant | value)
external_endpoint_parameter	STRING
api_call_entity_type	entity_type { matching_property } -
matching_property	MATCHING property external_endpoint_property
external_endpoint_property	STRING

An action rule can lead to the execution of one or more actions of a specific type. For example, an action may imply a causal link - changing the state of any transaction - or it may simply assign a value to a property in the system. We can have a sequence of one or more actions. For each action, one needs to specify the action type that will imply what concrete operations/instructions will be executed by the action engine and then define its parameters, specific to the corresponding action type, required for its execution. The set of available actions in this grammar is also extended by our approach, with the specification of the 'external api call' concept. When defining an action of the 'external api call' type, as can be seen in Figure 5, one starts by declaring the external api's endpoint url that is to be accessed. Then, the call action, between those supplied, is selected, followed by the definition of parameters in an explicit manner to be included in this request. These parameters can be assigned a free value or a system's constant or property value that is evaluated at run-time. Finally, if the desired action to perform when fetching data, or when the call action is read, is to create entities in our system, we must then specify the entities' type and the linkage that must be done between the entities' properties and the API call response's properties.

4.7. Automatic API documentation generation

As far as REST APIs are concerned, the existence of a corresponding documentation is equally important for the development teams as well as end consumers of the API, so they can visualize and interact with its resources without having any of the implementation logic in place. Therefore, our proposition of RRAM includes the automatic generation of the API documentation, based on the API properties specified with our low-code approach. For that, we will be using Swagger UI, an open source tool which generates a web page that documents the API, following the OpenAPI Specification[6] (OAS, formerly known as Swagger Specification). This documentation of the API is simple, user friendly and is also easy to change/update.

The generation of the documentation with Swagger UI will be based around the use of annotations for defining the information that shows up in the web page. Most of that information, such as HTTP method, parameters, response fields and so on, can be deduced based on the specifications made with the components mentioned in the previous subsections. Human readable information, like names and descriptions, will already be defined when complex queries and/or specific action rules are being created, or can be automatically generated based on the mentioned specifications/names. Without any implementation logic in place, Swagger UI enables anyone, including development teams and end users, to visualize and interact with an API's resources. Its latest release enables users to use the Swagger UI to visualize and automatically generate documentation of an API defined in OAS 3.0.

4.8. Rapid REST API Management Component

[6] https://swagger.io/specification/

The main component for our proposed solution will be developed in DISME for Rapid REST API Management. It will serve as an uniting control center for all features involving api management and documentation. Not only will it be the place where all API endpoints used, whether it be internal or external, are listed, but it will, for the aforementioned operation types that involve the action rules management component that uses Blockly, enable the possibility for users to directly load the action rule component with the corresponding action rule already loaded and ready for updates. It will also display the matching made between internal endpoints and their respective queries, allowing their preview and edition. The ultimate key feature in this component will be the possibility of activating/deactivating internal endpoints for external usage. On the listing of internal endpoints, whether they're from subsections 4.1, 4.2 or 4.4, the user will have the option to trigger their activation or deactivation. In the future, monitoring and api usage statistics is also planned to be included in this component.

4.9. DEMO meta-model extensions for API management

In order to develop the proposed solution, the DEMO's meta-model will have to be extended, so that it can provide the necessary management concepts to make the REST API automatic generation possible. The GOSL (General Ontology Specification Language) based diagram with these additions is shown in Figure 7. It is based on the EBNF grammar additions presented in section 4.5.

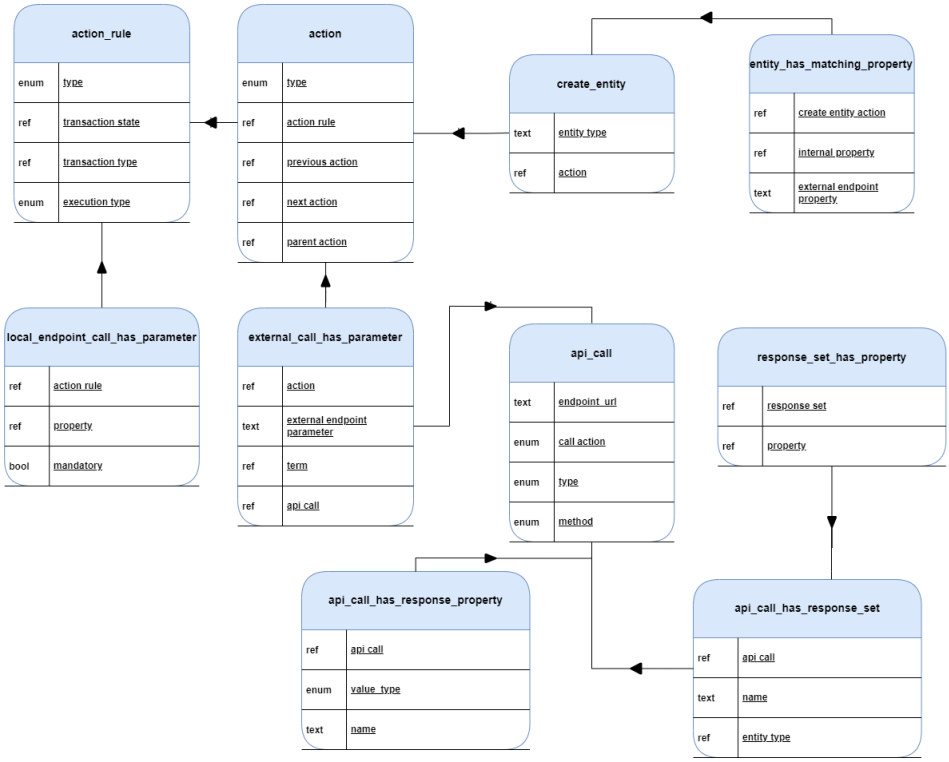

Figure 7: GOSL diagram for DEMO's meta-model extension

As it can be seen in Figure 7, the tables we're adding to DISME will enable the data for the general configuration of each endpoint, for the specific information for local endpoints that match internal action rules (action_rule_internal_call) and external action rules (external_api_action_rule_call). Additionally, in order to store the information about which properties will be used as parameters and which properties should be returned as a response to the API calls some other tables were added, providing support to many-to-many relationships (internal_call_has_response_property, internal_call_has_endpoint_parameter, external_call_has_response_property and external_call_has_endpoint_parameter). Finally, we represented the 'property' table in red, which represents a table that already exists in the DEMO's specification.

5. Conclusions and future work

In this paper we present a conceptual solution for rapid REST API management from DEMO Models with a low code approach. This solution will lead to the development of new components of the low-code platform being developed, DISME. This solution implies changes to DEMO's Meta Model in order to support this very important aspect of integration with other systems. We will support the creation of endpoints for both incoming and outgoing data and/or service actions using a friendly drag and drop GUI. Existing research work on this topic shows that rapid REST API management is a viable option which improves many development and usability aspects of software systems when compared to traditional technologies/frameworks. We believe our approach will improve many aspects of software systems like development time, productivity, ease-of-use and maintainability, not only because it is based on already existing DEMO Models and DISME, which provides friendly GUI features for configuring Action Rules and Complex Queries, but also because the corresponding API Documentation will be generated automatically. Also, we chose to build on a different diagrammatic representation of DEMO's FM because for the users that will use the low-code platform, diagrammatic representations with more usability are better, as it was proved in [9, 11] this is indeed more usable.

That said, we should make it clear that this paper represents a first iteration of Design Science Research methodology and there is a lot more work to be done. Furthermore, we aimed to debate these aspects at the conference, which led to a lively discussion especially regarding the DEMO's metamodel as well as the possibility/need to update and/or delete facts. At the end of the discussion it was agreed that it is really necessary to include more implementation aspects in DEMO, as well as the possibility to erase data in real systems due to GDPR restrictions.

As far as future work is concerned, there is still a lot to be done, given that the goal of this paper is to plan and identify the conceptual structure and requirements for such an approach, as well as getting some feedback on our approach. This functionality is still going to be implemented as an extension of our low-code platform DISME, and due to DEMO's principles we will also have to implement data restrictions for exchanging data via REST interfaces: length of fields/tables, specific datum, time formats, etc.

6. Acknowledgements

This work was supported by the program PROCiência 2020, funded by the European Regional Development Fund (ERDF), project MiColEC (M1420-01-0247-FEDER-000072).

7. References

[1] Andrade, M., Aveiro, D., Pinto, D.: DEMO based Dynamic Information System Modeller and Executer: In: Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management. pp. 383–390. SCITEPRESS - Science and Technology Publications, Seville, Spain (2018). https://doi.org/10.5220/0007230003830390.

[2] Wang, B., Rosenberg, D., Boehm, B.: Rapid realization of executable domain models via automatic code generation. Presented at the 2017 IEEE 28th Annual Software Technology Conference (STC) September 1 (2017). https://doi.org/10.1109/STC.2017.8234464.

[3] Mora-Segura, Á., Sánchez Cuadrado, J., Lara, J.: ODaaS: Towards the Model-Driven Engineering of Open Data Applications as Data Services. Presented at the Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOCW September 1 (2014). https://doi.org/10.1109/EDOCW.2014.55.

[4] Gonçalves, R., Azevedo, I.: RESTful Web Services Development With a Model-Driven Engineering Approach. In: Code Generation, Analysis Tools, and Testing for Quality. pp. 191–228 (2019). https://doi.org/10.4018/978-1-5225-7455-2.ch009.

[5] Hussein, S., Zein, S., Salleh, N.: REST API Auto Generation: A Model-Based Approach. Presented at the 19th International Conferences on New Trends on Intelligent Software Methodologies, Tools and Techniques September 17 (2020).

[6] Overeem, M., Jansen, S., Mathijssen, M.: API Management Maturity of Low-Code Development Platforms. In: Augusto, A., Gill, A., Nurcan, S., Reinhartz-Berger, I., Schmidt, R., and Zdravkovic, J. (eds.) Enterprise, Business-Process and Information Systems Modeling. pp. 380–394. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-79186-5_25.
[7] De, B.: API Documentation. In: De, B. (ed.) API Management: An Architect's Guide to Developing and Managing APIs for Your Organization. pp. 59–80. Apress, Berkeley, CA (2017). https://doi.org/10.1007/978-1-4842-1305-6_4.
[8] Krouwel, M., Op 't Land, M.: Business Driven Microservice Design - An Enterprise Ontology Based Approach to API Specifications. In: Advances in Enterprise Engineering XV. pp. 95–113. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-11520-2_7.
[9] Gouveia, B., Aveiro, D., Pacheco, D., Pinto, D., Gouveia, D.: Fact Model in DEMO - Urban Law Case and Proposal of Representation Improvements. In: Advances in Enterprise Engineering XIV. pp. 173–190 (2021). https://doi.org/10.1007/978-3-030-74196-9_10.
[10] Andrade, M., Aveiro, D., Pinto, D.: Bridging Ontology and Implementation with a New DEMO Action Meta-model and Engine. In: Advances in Enterprise Engineering XIII. pp. 66–82 (2020). https://doi.org/10.1007/978-3-030-37933-9_5.
[11] Pinto, D., Aveiro, D., Pacheco, D., Gouveia, B., Gouveia, D.: Validation of DEMO's Conciseness Quality and Proposal of Improvements to the Process Model. In: Advances in Enterprise Engineering XIV. pp. 133–152 (2021). https://doi.org/10.1007/978-3-030-74196-9_8.
[12] Pacheco, D., Aveiro, D., Pinto, D., Gouveia, B.: Towards the X-Theory: An Evaluation of the Perceived Quality and Functionality of DEMO's Process Model. (2022). https://doi.org/10.1007/978-3-031-11520-2_9.
[13] Dietz, J., Mulder, H.: Enterprise Ontology: A Human-Centric Approach to Understanding the Essence of Organisation. (2020). https://doi.org/10.1007/978-3-030-38854-6.
[14] Skotnica, M., Pergl, R.: Das Contract - A Visual Domain Specific Language for Modeling Blockchain Smart Contracts. In: Advances in Enterprise Engineering XIII. pp. 149–166 (2020). https://doi.org/10.1007/978-3-030-37933-9_10.
[15] Bollen, P.: SBVR: A Fact-Oriented OMG Standard. In: Meersman, R., Tari, Z., and Herrero, P. (eds.) On the Move to Meaningful Internet Systems: OTM 2008 Workshops. pp. 718–727. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88875-8_96.