MiColEC Project - “Micro-hubs Colaborativos para a Economia Circular”

Specification of the Component for Automatic Generation of APIs for Integration with Logistics Platforms

June 2022

Cofinanciado por:

REGIÃO AUTÓNOMA DA MADEIRA

PORTUGAL 2020

UNIÃO EUROPEIA
Fundo Europeu de Desenvolvimento Regional

Document information

Delivery date:

Version: 0.2

Responsible Partner:

Dissemination level

Public

Revision history

Date	Editor	Status	Version	Changes
10 March 2022		Draft	0.1	Initial Draft
20 June 2022		Draft	0.2	Complete Draft

Contributors

Logimade Lda

ARDITI - Agência Regional para o Desenvolvimento da Investigação, Tecnologia e Inovação

Universidade da Madeira

IL Technologies, Lda

Introduction **5**

MiCoLEC - context and objectives **5**

The Solution **6**

Architecture 6

Rapid REST API Management and Generation Component 8

Literature Review and related work 10

DEMO Models 11

Fact Model 11

Action Model 13

Our low-code platform approach 14

Simple CRUD Operations 15

Query Based Data Providing 15

Matching internal action rules with external endpoints 17

Matching local endpoints with internal action rules 18

Extending the Action Meta Model's EBNF grammar 19

Automatic API documentation generation 22

Component Implementation strategy 22

Risks and mitigation **23**

Bibliography **24**

Introduction

In this project, a micro-hub of logistic companies is expected to participate in a cooperative, while still competitive, package delivery marketplace that will effectively improve their overall performance. This improvement is achieved by gains in efficiency levels (e.g. route sharing), less waste (e.g. deduplication of courier routes), and increased trust in cooperation through the platform's technology. To this end, an innovative digital platform is proposed to allow trustworthy and accountable information sharing between the participants. In particular, this platform promises to allow, for instance, package handoff between competing companies in a completely transparent and secure way, while ensuring that even if something goes wrong it is possible to trace the package movements and assess liabilities. To support the platform, MiCoLEC proposes a blockchain-based system to manage interactions between participants and allowing most of the processes to be run on smart-contracts that transparently enforce fairness and that cannot be tampered with.

Along this document, we briefly describe the main goals for the project and provide details on how the platform will be implemented. Here we provide an overview of the platform's architecture and its components as well as how they should integrate with each other. We then focus on the Rapid REST API management component describing how it fits the high-level overview and how it will be integrated with the remainder of the platform as well as the existing systems. Then, we look at some potential implementation risks and how we intend to mitigate them and end the document with some final remarks.

MiCoLEC - context and objectives

Collaborative micro-hubs are a new logistical concept in which a group of delivery companies (express) collaborate among themselves by sharing means and resources of delivery of shipments in a network of common logistical centers installed in strategic areas of urban centers, essentially with a view to operating costs reduction, service quality improvement and coverage of more geographic areas without large initial investments. On the other hand, this same concept makes it possible to reduce urban traffic, with all the resulting advantages, namely the reduction of air pollution, the number of road accidents and improvement of the quality of life within urban areas. Despite the enormous advantages of collaborative micro-hubs, their implementation is a complex challenge, whose success lies in the implementation of solutions that solve the problem of trust between different transport operators in the exchange of information and execution of shared operations between them.

The circular economy brings together a set of reuse, recycling and sustainable end-of-life processes for the massive amount of products that the consumer economy produces daily, presenting itself today as the most viable and promising way to reach the necessary levels of environmental sustainability that the planet urgently needs. The advantage of the circular economy in relation to other ecologically sustainable philosophies of life is based on the fact that the processes it promotes can be introduced without significant changes in the lifestyle of modern societies. However, the effective implementation of circular economy processes depends on solutions to a set of challenges, among which the need to create economic models to encourage the continued participation of final consumers in circular economy processes, and the availability of supply of reverse logistics services (from the consumer) with adequate levels of cost and quality of service.

This project proposes to develop a practical solution that addresses these 2 problems head on, through the application of blockchain technology (distributed ledger technologies) for the implementation of a robust digital solution to solve the collaboration and trust issues that are at the base of the limitation of the implementation of collaborative micro-hubs, since they have the potential to record all transactions of the micro-hub, TEEs, producers, end consumers and collection and recycling centers, in a verifiable, permanent and transparent way for all interested agents.

On the other hand, the project provides for the creation of a new digital token (cryptocurrency), called “bit circle”, whose management will also be codified in the blockchain and which will be attributed to final consumers as an incentive to participate in circular economy processes. Both the allocation of digital tokens to end consumers and the management of transactions by exchange of services between transport companies will be carried out using smart-contracts, which are also registered in a transparent and verifiable way for all stakeholders on the blockchain.

The Solution

This section describes the proposed solution that meets the project requirements in regards to the generation of API's.

Architecture

The proposed architecture, as depicted in Figure 1, is a multi-layered approach. At the center of the proposed design is the blockchain component. The blockchain serves as the data repository log where all transactions, that is deliveries, order status, etc. are recorded. This component, which will be described in more detail in the following section, provides much-needed transparency, privacy and resiliency to all platform participants.

The deployment of the blockchain component will be in permissioned private mode, where the infrastructure is owned and deployed by the project consortium. In this way, ownership and transaction fees become much more manageable in the long run. The downside is the greater cost of ownership at the beginning and possibly greater barriers to entry for new participants.

New participants can either join the consortium, meaning they must run their blockchain node helping support, maintain and manage the platform, or, on the other hand, join the platform as a simple participant and to do so, connect to the deployed infrastructure and consume the exported APIs and integrate them with their own existing infrastructure.

Coupled with the blockchain node there will be a backend instance managed by each participant of the consortium, and deployed in their own infrastructure. The role of the backend is to abstract the intricacies of the blockchain and provide an easier to integrate and a more familiar interface, further lowering the barriers to entry for new participants. The backend will be responsible for interfacing with the blockchain, for interfacing with a Distributed Data Repository where auxiliary and intermediate data can be stored. In addition, the backend will export interface APIs and Event Queues that will be consumed by the participants of the platform.

As mentioned previously, new carrier participants just need to consume and integrate the exported APIs into their own existing infrastructure. In this way, there is minimal intrusion and changes to the way they currently operate, even integrations with Web/Mobile apps should be seamless with the proposed design.

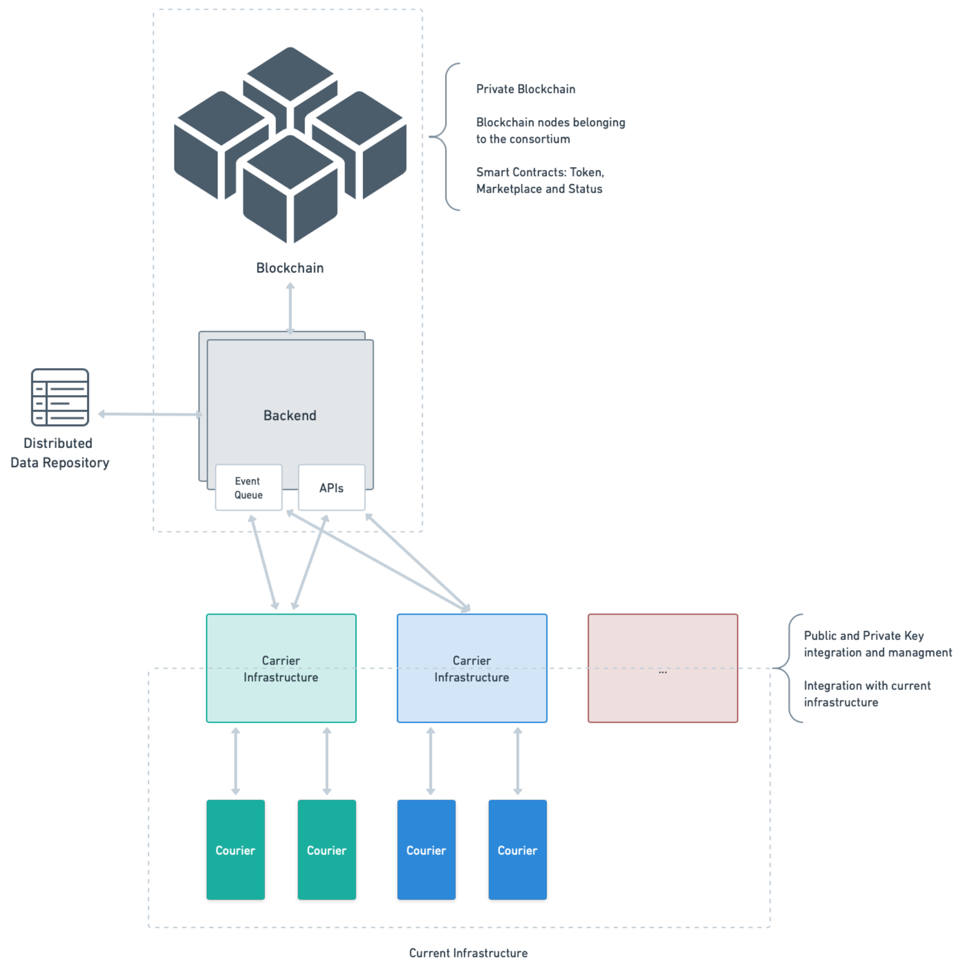

Private Blockchain
Blockchain nodes belonging to the consortium
Smart Contracts: Token, Marketplace and Status
Blockchain
Backend
Event Queue
APIs
Distributed Data Repository
Carrier Infrastructure
Carrier Infrastructure
...
Public and Private Key integration and managment
Integration with current infrastructure
Courier
Courier
Courier
Courier
Current Infrastructure

Figure 1. Proposed multi-layered architectural approach

Rapid REST API Management and Generation Component

This project foresees the specification and development of a component of automatic generation and management of APIs for integration with logistics platforms, for integration of the blockchain system to support collaborative micro-hubs, with the functionalities already implemented in traditional logistics platforms, as well as the development effort of these same functionalities directly on the micro-hub blockchain solution.

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.

Low-code platforms open the opportunity of automatic 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 match it with internal data and rules, also in a friendly GUI. The aim is to facilitate the integration of external information systems with the local system, based on our low-code approach.

This section presents our approach for Rapid REST API Management (RRAM) based on DEMO Models [1]. A DEMO based low-code platform (from here onwards referred to as DBLCP) is being developed by ARDITI. One of its components uses Blockly[1] 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 line[2]. We aim to take advantage of our experience with Blockly and develop a new component of DBLCP 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[2].

The following subsection presents the literature review on this topic, subsection ‘DEMO Models’ briefly explains the research on DEMO’s models. In subsection ‘Our low-code platform approach’, our low-code platform approach for RRAM is presented. Finally, we present our implementation strategy taking into account the given context.

Literature Review and related work

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

[1] https://developers.google.com/blockly

[2] https://swagger.io/tools/swagger-ui/

Some approaches used code generation. Wang, et al. [3], 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) [4, 5], an iterative and incremental software development process. Mora-Segura, et al. [4], 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. [6], 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. [7] 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 [8] 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.

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 [3] which we think 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 [3, 5] we can conclude that automatic generation of REST APIs is a viable option and, if done properly, can turn the process into a much simpler task that can easily be performed by people with little 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.

DEMO Models

In order to situate the innovation of this 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].

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 2 an example of the CRD is presented, showing the models developed for the 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 2, an instance of Application Deliverable cannot exist without a reference to an existing instance of Type of Application Deliverable.

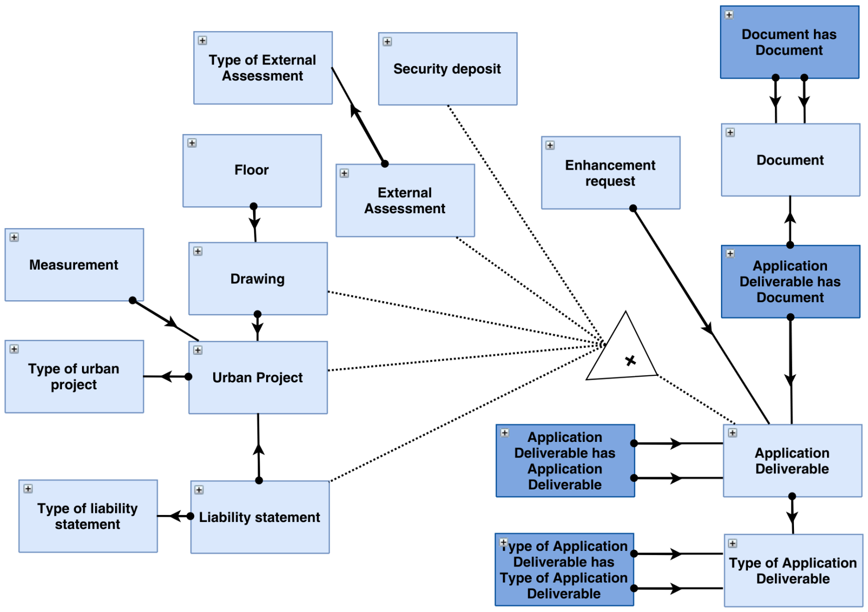

Type of External Assessment
Security deposit
Document has Document
Floor
External Assessment
Enhancement request
Document
Measurement
Drawing
Application Deliverable has Document
Type of urban project
Urban Project
Application Deliverable has Application Deliverable
Application Deliverable
Type of liability statement
Liability statement
Type of Application Deliverable has Type of Application Deliverable
Type of Application Deliverable

Figure 2. 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 2 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 2;

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 2.

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 3 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.

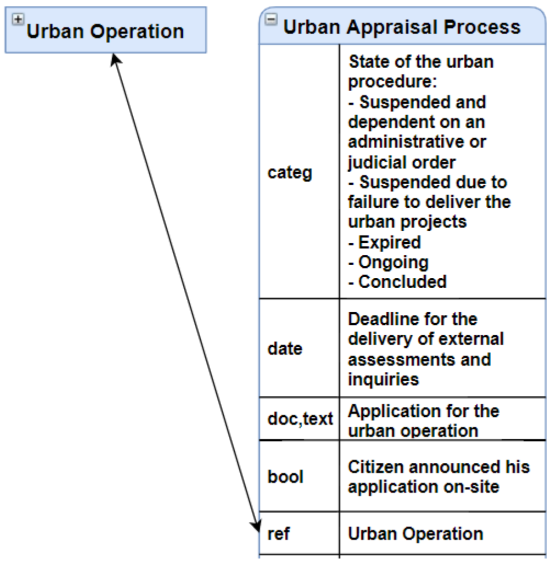

Urban Operation
Urban Appraisal Process
categ
State of the urban procedure:
- Suspended and dependent on an administrative or judicial order
- Suspended due to failure to deliver the urban projects
- Expired
- Ongoing
- Concluded
date
Deadline for the delivery of external assessments and inquiries
doc,text
Application for the urban operation
bool
Citizen announced his application on-site
ref
Urban Operation

Figure 3. 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.

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].

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.

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 the DBLCP that 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[3], 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 as follows: 1. simple CRUD (Create, Read, Update and Delete) operations on data from the local DBLCP system, 2. query based data provision by the DBLCP, 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, are inferred from its specification in our system after the needed information has been selected/provided and is used not only for defining the endpoint itself, but also to automatically generate API documentation, for which we will use Swagger UI[4].

[3] https://querybuilder.js.org/

[4] https://swagger.io/tools/swagger-ui/

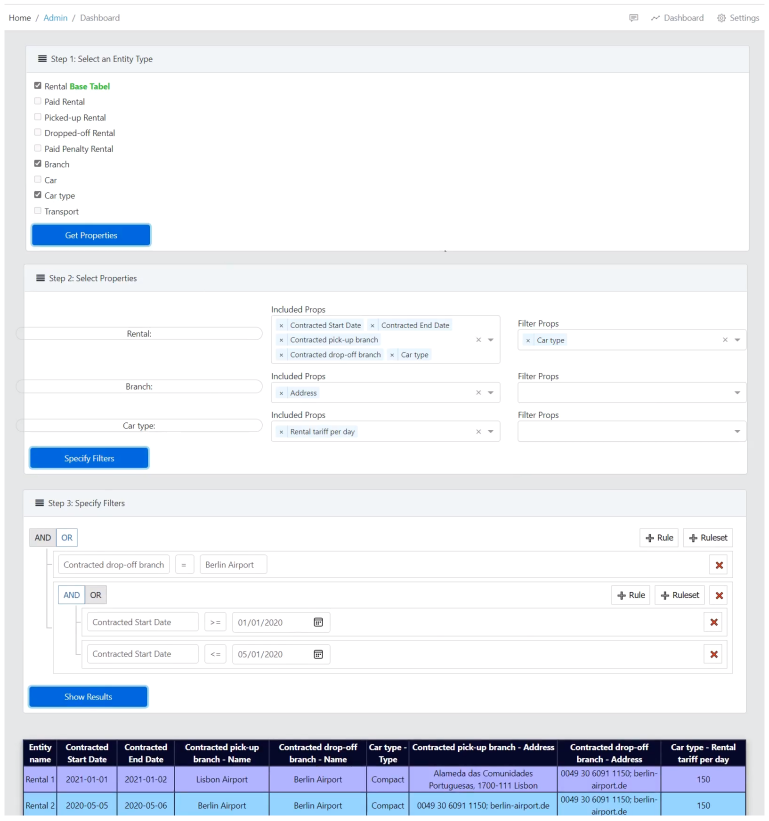
Simple CRUD Operations

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

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 DBLCP which allows the configuration of complex queries using drag and drop actions that works in the form of specification of queries 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.


Home / Admin / Dashboard

Step 1: Select an Entity Type

- [x] Rental **Base Tabel**
- [ ] Paid Rental
- [ ] Picked-up Rental
- [ ] Dropped-off Rental
- [ ] Paid Penalty Rental
- [x] Branch
- [ ] Car
- [x] Car type
- [ ] Transport

Step 2: Select Properties

	Included Props	Filter Props
Rental:	Contracted Start Date; Contracted End Date; Contracted pick-up branch; Contracted drop-off branch; Car type	Car type
Branch:	Address	
Car type:	Rental tariff per day	

Step 3: Specify Filters

AND | OR
- Contracted drop-off branch = Berlin Airport
- AND | OR
  - Contracted Start Date >= 01/01/2020
  - Contracted Start Date <= 05/01/2020

Entity name	Contracted Start Date	Contracted End Date	Contracted pick-up branch - Name	Contracted drop-off branch - Name	Car type - Type	Contracted pick-up branch - Address	Contracted drop-off branch - Address	Car type - Rental tariff per day
Rental 1	2021-01-01	2021-01-02	Lisbon Airport	Berlin Airport	Compact	Alameda das Comunidades Portuguesas, 1700-111 Lisbon	0049 30 6091 1150; berlin-airport.de	150
Rental 2	2020-05-05	2020-05-06	Berlin Airport	Berlin Airport	Compact	0049 30 6091 1150; berlin-airport.de	0049 30 6091 1150; berlin-airport.de	150

Figure 4. Complex query creation component from DBLCP

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 define the Included Properties and the Filter Properties. For each one, the options shown in the select box are the properties belonging to that entity type. The Included Properties are the properties that will be shown in the results table, with the selected properties defining the result table's rows, whereas the Filter Properties are the properties that will be used in the definition of queries based on 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.

Matching internal action rules with external endpoints

Our DBLCP 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 DBLCP 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 DBLCP's blockly component is shown in Figure 5.

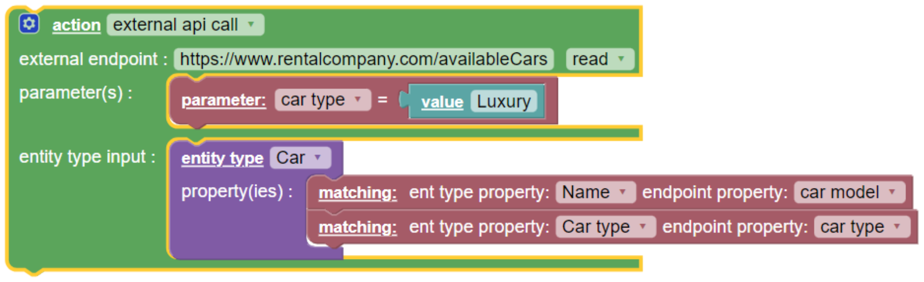


Figure 5. External_api_call action type definition example using DBLCP's Blockly component.

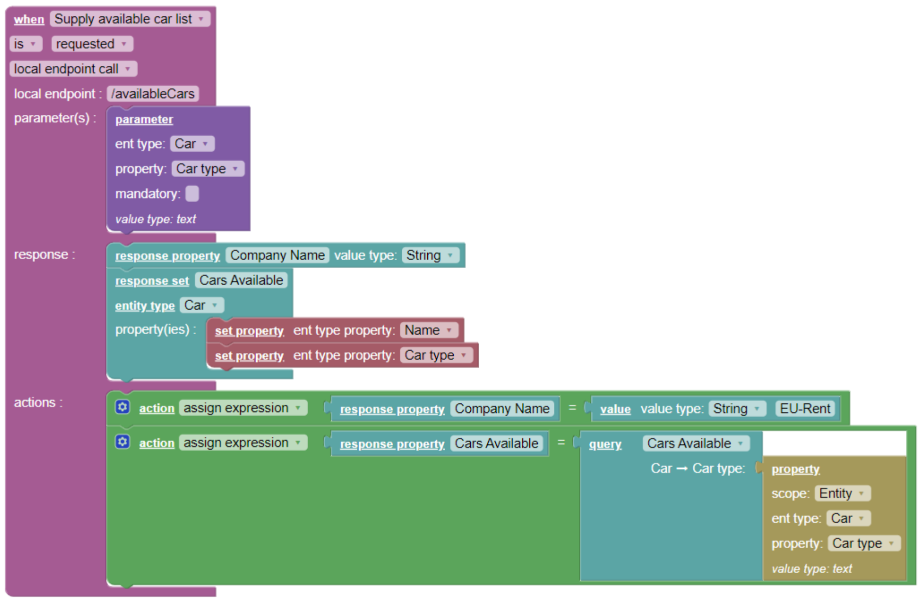
This external api call block also includes a slot to specify possible parameters for the call. A similar reasoning applies to the parameters as the one applied to the response properties. Blockly would automatically parse the possible parameters of the external endpoint, and include them in an internal 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.

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 DBLCP. 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.



```
when Supply available car list
is requested
local endpoint call
local endpoint : /availableCars
parameter(s) :  parameter
                ent type: Car
                property: Car type
                mandatory: 
                value type: text
response :      response property Company Name value type: String
                response set Cars Available
                entity type Car
                property(ies) : set property ent type property: Name
                                set property ent type property: Car type
actions :       action assign expression  response property Company Name = value value type: String EU-Rent
                action assign expression  response property Cars Available = query Cars Available
                                                                              Car → Car type: property
                                                                                              scope: Entity
                                                                                              ent type: Car
                                                                                              property: Car type
                                                                                              value type: text
```

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

This definition of a 'local endpoint call' execution type action rule through DBLCP'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.

Extending the Action Meta Model's EBNF grammar

Another contribution of this paper is the proposition of extending the current DBLCP's action rules' EBNF grammar. The changes introduced in this subsection are directly related to subsections 'Matching internal action rules with external endpoints' and 'Matching local endpoints with internal action rules', 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 ‘Matching internal action rules with external endpoints’., 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 ‘Matching local endpoints with internal action rules’’s new operation.

In the tables displayed below, only new/updated concepts on DBLCP’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.

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.

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.

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 dDocumentation, 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 (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 is 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.

Component Implementation strategy

We intend to leverage on the current experience we have with our DBLCP, to use it to directly implement parts of the features and requirements elicited as necessary for the backend component of our platform. Eventually the majority of them, if feasible. We foresee two scenarios:

1. Using DBLCP model editor to design the services and format of data so that we can generate, both the API specifications and (parts of) the backend code that will be needed to be programmed in order to make the necessary API's available and functional.
2. If time and encountered complexity allows, implementing the previous step and, additionally, using the low-code features of DBLCP to implement the needed business logic of the backend so that the needed programming is reduced to a minimum and the APIs can run on top of the logic implemented in a low-code fashion

Risks and mitigation

Risk	**Mitigation strategy**
Unexpected technical limitations	Extensive technological mapping and testing; Early proof-of-concept;
Limitations in the integration with legacy systems	Modularized architecture that allows proxy or translator components; Non-intrusive design where integration is done considering legacy systems as a black-box;

Bibliography

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)
2. Introduction to Blockly | Google Developers, https://developers.google.com/blockly/guides/overview?hl=pt
3. Wang, B., Rosenberg, D., Boehm, B.: Rapid realization of executable domain models via automatic code generation. Presented at the September 1 (2017)
4. 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)
5. Gonçalves, R., Azevedo, I.: RESTful Web Services Development With a Model-Driven Engineering Approach. Presented at the January 1 (2019)
6. Hussein, S., Zein, S., Salleh, N.: REST API Auto Generation: A Model-Based Approach. Presented at the September 17 (2020)
7. 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)
8. 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)
9. Gouveia, B., Aveiro, D., Pacheco, D., Pinto, D., Gouveia, D.: Fact Model in DEMO - Urban Law Case and Proposal of Representation Improvements. Presented at the April 14 (2021)
10. Andrade, M., Aveiro, D., Pinto, D.: Bridging Ontology and Implementation with a New DEMO Action Meta-model and Engine. Presented at the January 3 (2020)
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. Presented at the April 14 (2021)
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)
13. Dietz, J., Mulder, H.: Enterprise Ontology: A Human-Centric Approach to Understanding the Essence of Organisation. (2020)
14. Skotnica, M., Pergl, R.: Das Contract - A Visual Domain Specific Language for Modeling Blockchain Smart Contracts. Presented at the January 3 (2020)
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)