DM

Integração de plataforma *low-code* com *endpoints* externos com API REST

DISSERTAÇÃO DE MESTRADO

João Carlos Alves Vasconcelos
MESTRADO EM ENGENHARIA INFORMÁTICA

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

dezembro | 2023

Integração de plataforma *low-code* com *endpoints* externos com API REST

DISSERTAÇÃO DE MESTRADO

João Carlos Alves Vasconcelos
MESTRADO EM ENGENHARIA INFORMÁTICA

ORIENTAÇÃO
David Sardinha Andrade de Aveiro

Faculdade de Ciências Exatas e da Engenharia

Mestrado em Engenharia Informática

Integração de plataforma *low-code* com *endpoints* externos com API REST

João Carlos Alves Vasconcelos

Orientado por:

David Sardinha Andrade de Aveiro

Resumo

A necessidade de partilhar dados entre sistemas de informação, nomeadamente em low-code platforms (LCPs), é cada vez maior nos dias de hoje. A interconexão e a partilha de dados entre sistemas são aspetos essenciais para muitas organizações e empresas que utilizam plataformas de desenvolvimento e gestão *low-code*. A troca de informações entre diferentes tipos de sistemas pode melhorar a eficiência operacional, facilitando a colaboração entre equipas e fornecendo uma visão mais completa dos dados.

O trabalho desenvolvido nesta dissertação, no contexto do projeto DISME, plataforma low-code para executar e modelar processos de negócio, tem como objetivo abordar os desafios atuais na transmissão de dados entre sistemas de informação. O foco está no desenvolvimento de uma componente que facilite a comunicação entre esses sistemas, superando as barreiras atuais existentes. Utilizando tecnologias REST e JSON, procura-se fornecer uma solução eficaz de integração, otimizando a troca de informações e promovendo uma interoperabilidade mais eficiente e fluida entre os sistemas de informação.

Para conduzir esta pesquisa, adotou-se uma abordagem metodológica baseada em casos de estudo. Foi utilizado o caso EU-Rent, amplamente reconhecido e utilizado em pesquisas académicas para validar ideias, protótipos e conceitos. Este caso serviu como representação da realidade de muitas empresas de aluguer de carros e permitiu adquirir conhecimento sobre os desafios atuais na transmissão de dados. Utilizando a informação disponível e os dados fornecidos pelo caso EU-Rent, foi possível compreender as necessidades de integração e restrições técnicas. Esta abordagem permitiu uma melhor compreensão do contexto organizacional representado pelo caso EU-Rent, fundamentando o desenvolvimento de um novo módulo de integração do DISME.

Com os resultados apresentados, foi possível comprovar a eficiência e eficácia da componente desenvolvida ao replicar com sucesso alguns dos cenários identificados no estudo realizado no caso EU-Rent. Os testes realizados demonstram que a solução de integração proposta proporciona uma troca de informações mais fluida e eficiente entre os sistemas de informação, superando as barreiras existentes. Estes resultados evidenciam o impacto positivo da solução desenvolvida e sua relevância.

Keywords: Engenharia Organizacional · DEMO · Sistemas de Informação · Partilha de dados · API REST · Plataforma Low-code

Abstract

The need to share data between information systems, particularly in low-code platforms (LCPs), is ever greater in current times. Interconnection and data sharing between systems are essential aspects for many organizations and companies that use low-code development and management platforms. Exchanging information between different types of systems can enhance operational efficiency, facilitating collaboration among teams and providing a more comprehensive view of data.

The work developed in this dissertation, in the context of the DISME project, a low-code platform to execute and model business processes, aims to address current challenges in data transmission between information systems. The focus is on developing a component that eases the communication between these systems, overcoming existing barriers. Using REST and JSON technologies, the goal is to provide an effective integration solution, optimizing the exchange of information, promoting more efficient and seamless interoperability between information systems.

To conduct this research, a methodological approach based on case studies was adopted. The EU-Rent case, widely recognized and used in academic research to validate ideas, prototypes, and concepts, was employed. This case served as a representation of the reality of many car rental companies, allowing us to gain insight into the current challenges in data transmission. Using the available information and the data provided by the EU-Rent case, it was possible to understand integration needs and technical constraints. This approach allowed for a better understanding of the organizational context represented by the EU-Rent case, supporting the development of a new integration module for DISME.

With the presented results, it was possible to confirm the efficiency and effectiveness of the developed component by successfully replicating some of the scenarios identified in the study conducted on the EU-Rent case. The tests carried out demonstrate that the proposed integration solution provides a more seamless and efficient exchange of information between information systems, overcoming existing barriers. These results highlight the positive impact of the developed solution and its relevance.

Keywords: Organizational Engineering · DEMO · Computer Systems · Data Sharing · REST API · Low-code Platform

Agradecimentos

É com imensa gratidão que reconheço as pessoas que tornaram possível a realização desta dissertação, que não só representa a conclusão de uma etapa académica, mas também um marco significativo na minha vida.

Queria deixar um agradecimento especial ao meu orientador, Professor David Aveiro, cuja orientação, conhecimento e apoio foram essenciais para a concretização desta dissertação. Os seus conselhos e *feedback* foram cruciais no desenvolvimento deste trabalho.

Aos meus pais, deixo um sincero agradecimento pela compreensão, apoio incondicional e incentivo constante em todo o meu percurso académico.

Um agradecimento especial à minha namorada pelo apoio incansável e motivação constante durante os momentos mais desafiadores. A sua presença foi essencial para superar obstáculos e manter o foco.

Por último, agradeço a todos os meus amigos, em especial ao Luís, e colegas de curso pelo apoio e acompanhamento ao longo da minha vida académica.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

API Application Programming Interface

DEMO Design and Engineering Methodology for Organizations

DISME Direct Information Systems Modeller and Executer

DTD Document Type Definition

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

ORM Object Relational Mapper

PHP Hypertext Preprocessor

RAC Rent a Car

REST REpresentational State Transfer

SI Sistemas de Informação

SOAP Simple Object Access Protocol

SQL Structured Query Language

XSD XML Schema Definition

1 Introdução

O presente capítulo apresenta de uma forma sucinta o contexto, o problema, os objetivos e a estrutura do projeto para uma melhor compreensão.

1.1 Contexto

A evolução dos Sistemas de Informação (SI) ao longo das últimas décadas tem desempenhado um papel crucial na transformação das organizações. O aumento da dependência dos sistemas de informação para a execução de processos e gestão de dados revela a importância da interoperabilidade e partilha de informações entre diferentes sistemas. Nesse contexto, o transporte eficiente e seguro de dados entre sistemas torna-se uma necessidade essencial para garantir a eficiência operacional entre equipas.

Desde o final da década de 1950 e o início da década de 1960, com os sistemas de informação, avançando pelos sistemas em rede nas décadas de 1970 e 1980, até a atual computação em nuvem, a evolução tecnológica tem possibilitado avanços significativos na comunicação e na troca de informações [1]. Os sistemas de informação modernos operam em ambientes complexos, onde vários sistemas coexistem e cada um desempenha um papel em específico. A comunicação entre esses sistemas permite obter uma visão global dos dados e facilita a tomada de decisões informadas. No entanto, a falta de integração e comunicação efetiva entre esses sistemas tem sido um desafio enfrentado por muitas organizações, resultando em processos fragmentados, falta de acessibilidade aos dados e ineficiências operacionais.

1.1.1 DISME e low-code platforms

Com a evolução contínua dos sistemas de informação, surgiu uma abordagem inovadora conhecida como *low-code platforms* (LCPs). Estas plataformas foram desenvolvidas para facilitar o desenvolvimento de aplicações de software, permitindo que pessoas com menos literacia digital pudessem criar soluções tecnológicas de forma mais ágil e intuitiva.

As low-code platforms disponibilizam ao utilizador uma interface visual e ferramentas de desenvolvimento simplificadas, permitindo que os utilizadores criem aplicações através de componentes pré-configurados e fluxos de trabalhos visuais. Este tipo de abordagem reduz significativamente a quantidade de código necessário para desenvolver uma aplicação, acabando por tornar o processo mais acessível e rápido para pessoas com menos experiência em programação. No entanto, a falta de integração e comunicação efetiva entre sistemas desenvolvidos em low-code platforms e outros sistemas de informação continua a ser um desafio.

O DISME (Dynami Information System Modeller and Executer) será a LCP utilizada no contexto desta dissertação. Funcionando como uma plataforma *low-code*, o DISME oferece uma abordagem inovadora para o desenvolvimento de soluções de integração de SI.

1.2 Objetivos

O transporte de dados entre sistemas de informação tornou-se uma prioridade para superar as barreiras mencionadas e promover a colaboração entre sistemas de informação, principalmente em *low-code platforms*. A partilha eficiente e segura de informações possibilita uma visão e gestão dos dados mais completa, otimiza a utilização dos recursos organizacionais e melhora a capacidade de

resposta aos desafios do mercado. Além disso, em uma realidade cada vez mais orientada para a partilha de dados, a troca ágil e precisa de informações entre sistemas de informação desempenha um papel crucial na obtenção de vantagem competitiva.

Nesta dissertação, foca-se a atenção na abordagem dos desafios atuais da transmissão de dados entre sistemas de informação em *low-code platforms (LCPs)*, com o objetivo de desenvolver mecanismos de integração do *Direct Information Systems Modeller and Executer* (DISME) com sistemas de informação existentes, através de tecnologias *Representational State Transfer* (REST) e *JavaScript Object Notation* (JSON). Ao utilizar estas tecnologias, procura-se superar as barreiras existentes na atualidade e promover uma interoperabilidade mais fluida e eficiente entre os sistemas de informação que utilizam *low-code platforms* no seu desenvolvimento. Para isso, pretende-se melhorar e desenvolver novas ferramentas no DISME de modo a que seja possível comunicar com outros sistemas de informação através de REST API.

1.3 Resultados

Com base numa abordagem metodológica apoiada em casos de estudo, explora-se as necessidades de integração e restrições técnicas de uma empresa *Rent a Car (RAC)*, fornecendo uma base sólida para o desenvolvimento de um novo módulo de integração do DISME.

Ao abordar estas questões, pretendemos contribuir para a evolução dos sistemas de informação, fornecendo uma solução eficaz para a transmissão de dados entre *low-code platforms*. Por meio da otimização de troca de informações, é esperada uma melhoria da eficiência operacional das organizações.

Além disso, é importante salientar que os resultados e descobertas deste documento contribuíram para a produção de dois artigos científicos. O primeiro artigo incorporou alguma da investigação realizada no capítulo 2, foi submetido e aceite para publicação intitulado de *"DEMO Model based Rapid REST API Management in a low code platform"* na *12th Enterprise Engineering Working Conference* [2]. O segundo artigo engloba conteúdo do capítulo 3, foi submetido, aceite e encontra-se em processo de publicação denominado de *"Rapid REST API Management in a DEMO based Low Code Platform"* na *13th Enterprise Engineering Working Conference* [3].

1.4 Estrutura do relatório

O presente relatório está estruturado em cinco capítulos principais, pretendendo facilitar a compreensão do estudo e dos resultados alcançados:

1. **Introdução:** Este capítulo oferece uma visão geral do contexto do relatório, abordando o tema das LCPs e do DISME. O problema central do relatório é identificado, seguido da solução proposta para o mesmo.
2. **Contexto do projeto:** Detalha o cenário em que o projeto está inserido, abordando a importância da interoperabilidade dos sistemas de informação nas tecnologias (REST API, XML, JSON e SOAP) e em conceitos como LCPs. O capítulo termina com uma breve introdução ao DISME.
3. **Implementação:** Este capítulo é dedicado à explicação técnica e prática da proposta apresentada. Nele serão exploradas as integrações REST em LCPs, o levantamento de requisitos, a

arquitetura DISME e os seus componentes. Será ainda abordada a ferramenta de desenvolvimento Blockly, explicando as novas peças criadas e a sua função no sistema. O capítulo encerra com uma análise do mecanismo de execução implementado.

4. **Validação:** Comprova-se a aplicabilidade da solução proposta utilizando o caso de estudo EURent para validar as ideias, protótipos e conceitos apresentados. O capítulo define o processo de validação, as tarefas e as *Application Programming Interfaces (APIs)* envolvidas e os resultados obtidos.

5. **Conclusão:** No último capítulo, serão apresentados os resultados alcançados, as contribuições para a área de estudo, possíveis sugestões de trabalhos futuros e as conclusões finais do trabalho.

2 Contexto do projeto

Neste capítulo, será realizada uma análise abrangente do contexto da interoperabilidade nos sistemas de informação e das tecnologias utilizadas nesse processo. Será ainda explorado o papel das low-code platforms, bem como o meta-modelo DEMO e a sua implementação prática, o DISME.

2.1 Interoperabilidade

A interoperabilidade nos sistemas de informação é de elevada importância para diversas áreas como a saúde, segurança pública, comércio e outros setores que dependem da partilha eficiente de dados e informações. Entendemos a interoperabilidade como sendo a capacidade de diferentes sistemas, dispositivos e aplicações colaborarem e partilharem informações de maneira coordenada, mesmo que provenham de organizações, regiões ou países diferentes [4]. Utilizando os sistemas de saúde como exemplo, a importância da interoperabilidade pode ser explorada considerando vários fatores.

A interoperabilidade permite que os profissionais de saúde e médicos consigam aceder de forma rápida ao histórico médico completo de um paciente, independentemente do local ou sistema onde foram registados.

Por exemplo, num sistema de saúde sem interoperabilidade, um paciente é diagnosticado com uma doença crónica e é prescrito um medicamento específico para o tratamento. Se o histórico médico do paciente não estiver acessível em diferentes clínicas e hospitais, cada vez que ele precisar de atendimento num local diferente, os médicos não terão acesso completo às medicações e aos tratamentos anteriores, o que pode levar a prescrições em duplicado ou medicamentos incompatíveis, resultando em desperdício de medicamentos e potenciais reações adversas.

Neste contexto, a troca de informações entre diferentes instalações médicas evita redundâncias na informação, erros médicos e auxilia na tomada de decisões informadas e precisas [5].

A capacidade de partilhar informações de saúde de forma eficiente e rápida evita repetição de exames e procedimentos médicos em vários sistemas, o que resulta na capacidade de economizar recursos financeiros e tempo.

Por exemplo, imaginemos que um paciente que foi atendido num hospital realizou exames diagnóstico importantes, como uma ressonância magnética e análises laboratoriais. Se o histórico médico desse paciente não estiver disponível para consulta em outros hospitais ou clínicas, ele pode ser submetido a novos exames, repetindo os mesmos todas as vezes que precisar de atendimento numa outra instituição. Isso pode resultar em atrasos no diagnóstico, custos adicionais para o paciente e desperdício de recursos médicos.

Neste caso, a interoperabilidade permite o acesso rápido e seguro às informações de saúde de um paciente em diferentes instituições [4].

A partilha de dados clínicos em larga escala é essencial para estudos e pesquisas médicas, descoberta de medicamentos e avanços na área da saúde. Com acesso a uma vasta quantidade de informações e dados, os investigadores podem fazer análises mais abrangentes e identificar tendências ou padrões que levem a novas descobertas, terapias e tratamentos [6].

Com o desenvolvimento de sistemas de informação que contenham uma vertente de interoperabilidade bem estabelecida, as empresas de tecnologia têm um maior incentivo para desenvolver

soluções inovadoras que se integrem cada vez mais com sistemas existentes [7]. Deste modo, é criado um ambiente favorável ao desenvolvimento de novas aplicações e uma melhoria contínua destes sistemas.

As tecnologias utilizadas na interoperabilidade de sistemas de informação são essenciais para garantir que diferentes sistemas e organizações possam trabalhar em conjunto, trocar informações e interagir de forma eficaz e eficiente. A interoperabilidade é fundamental para facilitar a comunicação e partilha de dados entre diferentes sistemas, evitando problemas como duplicação de esforços, perda de informações e aumento de dados.

2.1.1 REST API

Uma *REpresentational State Transfer* (REST) *Application programming interface* (API) é uma interface de programação de aplicações que segue os princípios de design da arquitetura REST. É uma arquitetura de software que se baseia em um conjunto de restrições de design para desenvolver sistemas distribuídos eficientes, confiáveis e escaláveis. Um sistema é considerado RESTful quando adere a todas essas restrições. A ideia fundamental do REST é que os recursos, como documentos e dados, sejam transferidos, e as operações sejam padronizadas, tornando os serviços RESTful mais auto descritivos, o que facilita a interação entre clientes e servidores de sistemas de informação [8,9].

As RESTful APIs são largamente utilizadas na construção de aplicações, devido à sua simplicidade e escalabilidade. Essas APIs utilizam métodos HTTP, como GET, POST, PUT e DELETE, para aceder a ações e realizar operações de leitura, criação, atualização e eliminação denominadas de *create, read, update and delete* (CRUD) *Operations* [10].

Na Figura 1 podemos observar o modelo de comunicação entre um cliente e um servidor através da utilização de uma REST API. Para entendermos um pouco mais de como funciona uma REST API é importante conhecer alguns conceitos:

Cliente: é a aplicação ou serviço que necessita de utilizar a REST API por meio de uma requisição aos recursos para realizar operações.

Servidor: é a aplicação ou serviço que contém os recursos e processa todas as requisições recebidas do cliente, retornando as respostas correspondentes.

Recursos: são os dados e informações disponíveis no servidor que podem ser acedidos ou manipulados através da REST API.

Representações: as respostas da REST API podem ser devolvidas em diferentes formatos, como *JavaScript Object Notation* (JSON) ou *Extensible Markup Language* (XML), que representam os dados do recurso solicitado, formatos estes que serão abordados na secção de tecnologias utilizadas.

Métodos: são utilizados para indicar a ação que deve ser realizada sobre um determinado recurso. Alguns dos métodos mais comuns são:

- GET: utilizado para obter informações ou representações de um recurso no servidor.
- POST: usado para criar um novo recurso no servidor.
- PUT: usado para atualizar um recurso existente no servidor.
- DELETE: utilizado para remover um recurso do servidor.

Stateless: Uma API REST é considerada sem estado, significando que cada requisição feita pelo cliente ao servidor deve conter todas as informações necessárias para que o servidor entenda e processe o pedido. O servidor não mantém nenhum estado entre pedidos, o que torna as requisições independentes uma da outra [11].

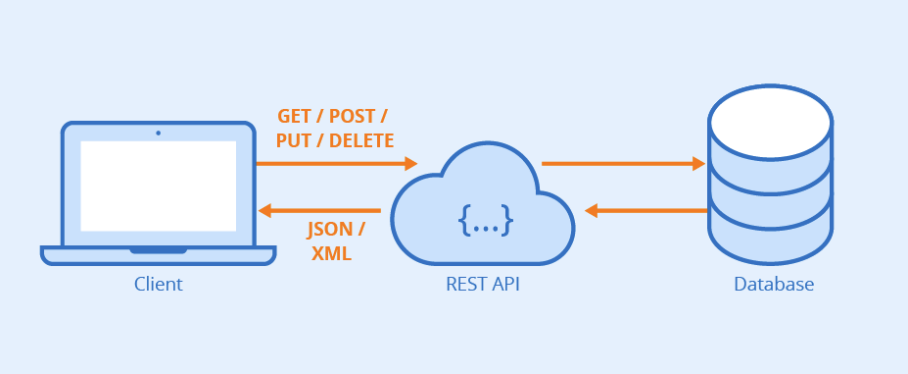

Figura 1. Arquitetura básica de um sistema com uma REST API.

2.1.2 XML

Como referido anteriormente, *Extensible Markup Language* (XML) é uma *markup language* muito utilizada para representar e armazenar dados de forma estruturada e legível por humanos e máquinas, razão pela qual é muito utilizada na representação de dados em REST APIs. É uma alternativa ao *HyperText Markup Language* (HTML) e ao *JavaScript Object Notation* (JSON).

O XML foi projetado para ser independente de software ou hardware, adequado para armazenamento e transporte de dados entre SI, como sites, bases de dados e aplicações de terceiros [12].

A sua construção é feita em torno de tags que são predefinidas pelo utilizador e projetadas especificamente para as necessidades do mesmo [13]. É uma maneira flexível e poderosa de armazenar dados num único formato que pode ser armazenado, pesquisado e partilhado entre SI. O seu formato é padronizado, o que significa que, ao transmitir e partilhar estruturas XML entre SI, localmente ou pela internet, o receptor consegue interpretar e utilizar o XML.

Esta tecnologia tornou-se importante nas REST APIs pelas seguintes razões:

Padrão estabelecido: O XML tem sido amplamente utilizado na indústria da computação há muito tempo e muitos SI já têm as suas REST APIs projetadas e suportadas com esse formato de dados. Desta forma, os desenvolvedores e sistemas estão familiarizados com XML, o que torna mais fácil a sua adoção e integração de REST APIs que utilizem este formato.

Esquemas de validação: Suporta a definição de esquemas como *Document Type Definition* (DTD) ou *XML Schema Definition* (XSD) que torna possível a validação da estrutura e conteúdo dos dados, ajudando a garantir a integridade dos dados enviados e recebidos pelas REST APIs.

Interoperabilidade: Como o XML não está diretamente relacionado com uma plataforma ou linguagem em específico, pode ser lido e interpretado por várias tecnologias diferentes, o que facilita a interoperabilidade entre SI [14].

Na Figura 2 podemos observar uma estrutura de dados em linguagem XML.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
        <book category="cooking">
                <title lang="en">Everyday Italian</title>
                <author>Giada De Laurentiis</author>
                <year>2005</year>
                <price>30.00</price>
        </book>
        <book category="children">
                <title lang="en">Harry Potter</title>
                <author>J K. Rowling</author>
                <year>2005</year>
                <price>29.99</price>
        </book>
        <book category="web">
                <title lang="en">Learning XML</title>
                <author>Erik T. Ray</author>
                <year>2003</year>
                <price>39.95</price>
        </book>
</bookstore>
```

Figura 2. Estrutura de dados em formato XML.

Utilizar a linguagem XML em REST APIs traz vantagens como:

1. Estrutura hierárquica: permite a criação de uma estrutura de dados em hierarquia usando tags, tornando mais fácil a organização de informações complexas.
2. Flexibilidade: os desenvolvedores podem criar e personalizar as próprias tags, adaptando a estrutura às necessidades de cada aplicação.
3. Legibilidade: o XML é legível tanto para humanos quanto para máquinas, facilitando o entendimento dos dados que são transmitidos.
4. Suporte: é uma linguagem suportada por várias linguagens de programação, tornando-a uma opção viável a todo o tipo de SI.

Apesar de ser uma linguagem muito utilizada em REST APIs não deixa de ter desvantagens, como as seguintes:

1. Dificuldade de análise: o processamento de documentos XML pode ser mais completo e demorado para a análise de dados em comparação a formatos mais leves.
2. Menos eficiente: Devido à sua natureza mais pesada, o XML pode, por vezes, não ser a melhor opção para dispositivos com recursos limitados, como dispositivos móveis.

Por outro lado a tecnologia mais utilizada, atualmente, na representação de dados pelas REST APIs é a estrutura JavaScript Object Notation (JSON).

É um formato compacto utilizado para troca de dados simples e rápida entre SI, baseado em texto legível para seres humanos e máquinas, geralmente utilizado no formato *key-value*. Apesar

da sua sintaxe ser semelhante a uma notação de objeto em JavaScript, é um formato de dados independente da linguagem e pode ser usado de forma autónoma. Além disso, muitos ambientes de programação utilizam funções para gerar e analisar dados em formato JSON [15].

As vantagens do formato JSON são [16]:

1. Independente da linguagem: embora seja um formato derivado da notação JavaScript, é independente e pode ser utilizado em outras linguagens de programação.
2. Rápida e eficiente: é uma estrutura de dados leve e compacta em comparação às suas concorrentes.
3. Tamanho: tem uma contagem de caracteres baixa que reduz substancialmente a sobrecarga nas transferências de dados.

As desvantagens da sua utilização são [16]:

1. Segurança: a falta de suporte em alguns navegadores web faz com que exista, por vezes, falhas de segurança.
2. Tipos de dados: apesar de suportar a maior parte de tipos de dados básicos, como números, strings e arrays, o JSON tem limitações quando se trata de representar estruturas de dados mais complexas. Não suporta, por exemplo, a representação de datas e horas.

Na Figura 3 observamos uma estrutura de dados em JSON.


```json
{
  "bookstore": {
    "book": [
      {
        "title": "Everyday Italian",
        "author": "Giada De Laurentiis",
        "year": 2005,
        "price": 30
      },
      {
        "title": "Harry Potter",
        "author": "J K. Rowling",
        "year": 2005,
        "price": 29.99
      },
      {
        "title": "Learning XML",
        "author": "Erik T. Ray",
        "year": 2003,
        "price": 39.95
      }
    ]
  }
}
```


Figura 3. Estrutura de dados em JSON.

A sua relevância na utilização nas REST APIs deve-se à sua:

1. **Legibilidade**: o JSON é uma estrutura fácil de ler e escrever tanto para seres humanos quanto para as máquinas [17].
2. **Eficiência**: as suas características eficientes de cache podem reduzir pedidos desnecessários ao servidor, otimizando os tempos de resposta para o cliente [17].
3. **Interoperabilidade**: todas as linguagens de programação modernas fornecem suporte para gerar e analisar dados em JSON [18].

4. **Tamanho**: é um formato compacto, o que ajuda a aumentar os pedidos *Hypertext Transfer Protocol* (HTTP) e reduzir o tamanho dos dados transferidos [17].

Após a análise realizada, podemos afirmar que no que diz respeito à representação e transferência de dados, especialmente num contexto de REST APIs, são destacadas duas estruturas: XML e JSON. Embora sejam amplamente utilizadas com propósitos similares, têm características distintas que influenciam na sua adoção e uso nos SI.

O XML, uma linguagem de marcação, depende essencialmente de tags para determinar e estruturar os dados. Esta estrutura oferece uma enorme flexibilidade, permitindo a representação de dados de forma complexa e hierárquica.

Por outro lado, o JSON que deriva de uma notação de objeto, adota frequentemente um formato *key-value*. Ao contrário do XML, a leitura e escrita de dados em JSON torna-se uma tarefa mais simples, especialmente para estruturas menos complexas. No entanto o JSON enfrenta limitações quando se trata de representar estruturas de dados mais complexas, acabando por compensar a sua escolha pela sua natureza compacta, que muitas vezes resulta em transferências de dados mais rápidas.

Quando se trata de interoperabilidade, tanto o XML como o JSON têm as mesmas características. São independentes de uma linguagem e amplamente suportados por diversas linguagens de programação modernas.

No contexto das REST APIs, a escolha entre XML e JSON muitas vezes depende da complexidade e antiguidade do sistema. Em sistemas mais antigos tendem a manter o XML. Em contrapartida, para sistemas mais recentes opta-se, na maioria das vezes, pelo JSON.

2.1.3 SOAP

Simple Object Access Protocol (SOAP) é um protocolo de mensagens usado para trocar informações estruturadas entre serviços web. SOAP e REST, como abordado anteriormente, são duas arquiteturas de serviço web usadas para comunicação entre SI. SOAP utiliza um conjunto de informações XML para o formato da sua mensagem e baseia-se em protocolos de camada de aplicação como o *Hypertext Transfer Protocol* (HTTP).

SOAP é um protocolo baseado em XML usado para troca de informações em ambientes descentralizados e distribuídos. A sua estrutura consiste em três partes: um envelope, que define o início e o fim da mensagem; um cabeçalho, que armazena informações de controlo; e um corpo, que contém os detalhes da mensagem em si [19].

Em comparação com a arquitetura *Representation State Transfer* (REST) o protocolo SOAP é baseado em mensagens e exige um conjunto de regras rigorosas e de padrões. Em contrapartida REST é mais flexível e usa os métodos padrão HTTP como *GET*, *POST*, *PUT* e *DELETE*.

Esta abordagem desempenhou um papel crucial na evolução dos serviços web, oferecendo um padrão rigoroso na comunicação entre SI. Embora a arquitetura REST tenha vindo a facilitar na implementação de um protocolo de mensagens para troca de informações simples e rápido, SOAP ainda é uma ferramenta indicada em certos cenários, especialmente onde a segurança, integridade da mensagem e transações são prioritárias.

2.2 Low-code platforms

Como o contexto deste projeto concentra-se no uso de *low-code platforms* (LCP), entendemos ser fundamental explorar a tecnologia inerente a este conceito, evidenciando a sua importância nos SI. Assim, conseguiremos uma compreensão mais aprofundada da LCP que será utilizada no desenvolvimento deste projeto.

Plataformas de desenvolvimento *low-code* caracterizam-se como uma abordagem de desenvolvimento, implementação, execução e gerenciamento rápidos de aplicações que prioriza o desenvolvimento visual em vez do habitual desenvolvimento em código. O tradicional ambiente técnico de código é substituído por uma interface *drag-and-drop*. Isso permite que utilizadores com diversos níveis de habilidade em desenvolvimento, desde desenvolvedores profissionais a principiantes, especialistas na área e *stakeholders*, usem plataformas de desenvolvimento *low-code* para construir aplicações [20].

A crescente popularidade das LCPs é facilmente justificável, sendo uma das suas maiores vantagens a rapidez no desenvolvimento. A necessidade de acelerar o desenvolvimento e lançamento de aplicações, que combina com a escassez de desenvolvedores qualificados, torna as LCPs uma escolha atrativa. Além disso, proporcionam uma significativa redução de custos quando comparadas ao desenvolvimento tradicional e oferecem uma flexibilidade sem precedentes, permitindo a inclusão de pessoas com menos habilidades técnicas no processo de criação de aplicações. Um artigo comparou o desenvolvimento de um sistema de informação denominado de NexusBRaNT através da utilização de desenvolvimento tradicional de *software* com uma abordagem de desenvolvimento *low-code*, os resultados demonstram que, neste caso, o tempo de desenvolvimento numa LCP foi cerca de 95% menor em relação ao desenvolvimento tradicional. Este resultado evidencia um dos motivos pelas quais as LCP têm sido cada vez mais utilizadas. [21]

No entanto, como qualquer tecnologia, as LCPs não estão livres de problemas. Uma das limitações é a restrição de personalizações avançadas. Mesmo que este tipo de plataformas possa atender a muitos casos de uso, as aplicações mais específicas e complexas podem não ser exequíveis sem o uso de código tradicional. Além disso, os custos iniciais mais baixos podem ser ofuscados por outros custos que surgem à medida que os projetos se tornam mais complexos. A dependência de um fornecedor também pode tornar-se um problema, especialmente se surgir a necessidade de migrar para outras soluções.

Atualmente o desenvolvimento de *softwares* a partir do zero envolve a contratação de uma equipa de desenvolvedores, a escolha de uma linguagem de programação, bem como a definição de prazos de desenvolvimento, que na maior parte das vezes acabam por ser prolongados. A solução encontrada para reduzir o tempo de desenvolvimento destes *softwares* é a utilização de LCP. Este tipo de plataformas fornece ferramentas como *databases*, *front-end* e *backend*, escondendo a complexidade de desenvolvimento de aplicações [22].

Na secção seguinte analisou-se algumas LCPs da indústria que contenham componentes de desenvolvimento de REST APIs que promovam a interoperabilidade dos SI desenvolvidos e mantidos em LCPs como por exemplo, Outsystems[1], Mendix[2] e Budibase[3].

[1] `https://www.outsystems.com/`
[2] `https://www.mendix.com/`
[3] `https://budibase.com/`

2.2.1 Integração REST em Low-code Platforms

Antes de passar à implementação da componente REST no DISME foi necessário entender diferentes integrações REST em outras *low-code platforms*. É essencial perceber como é que as outras plataformas abordam essa integração. Portanto realizou-se uma análise nas três plataformas mencionadas anteriormente com foco em:

1. Entender a interface do utilizador: procurou-se perceber como é que o utilizador interage com uma LCP focada na interoperabilidade, que ferramentas e funcionalidades estão ao seu dispor e como são apresentadas.

2. Pontos fortes e fracos: através de testes práticos, pretendeu-se identificar aspetos em que estas plataformas se destacam e as áreas em que podem ser melhoradas.

3. Preparação para levantamento de requisitos: a compreensão detalhada de cada plataforma e das suas capacidades e limitações serviram como base para um posterior levantamento de requisitos.

2.2.1.1 OutSystems

A *low-code platform OutSystems* é líder no mercado de desenvolvimento *low-code*. Além de ter várias ferramentas que possibilitam ao utilizador desenvolver aplicações, tem uma ferramenta de gestão própria para expor e consumir recursos de REST APIs. Esta ferramenta permite que os utilizadores do *OutSystems* possam tornar as suas aplicações interoperáveis, caso pretendam.

Para uma melhor visão de como é que uma ferramenta de gestão de REST APIs está introduzida numa LCP, realizou-se alguns testes do ponto de vista técnico como unitários, de integração, de sistema, de desempenho e usabilidade. Procurou-se perceber que requisitos são necessários para replicar uma ferramenta semelhante numa LCP.

Numa primeira fase abriu-se a interface principal do *OutSystems*, como observado na Figura 4, para criar uma aplicação teste que possibilitasse o consumo de dados externos através de uma REST API. De uma forma geral pretendeu-se, através de um botão, realizar uma leitura de dados de outro SI para que os mesmos fossem armazenados na nova aplicação e exibidos ao utilizador.

Como observado na Figura 4, numa primeira instância acedeu-se ao separador *integrations* na componente *logic* do sistema. Posteriormente foi selecionada a opção *Consume REST API*, onde surgiu uma nova janela de configuração. Nessa janela inseriu-se o método pretendido para a chamada e o URL da REST API. Na Figura 5, pode-se observar a realização automática da chamada pelo sistema, com uma pré-visualização dos dados, e após a confirmação do utilizador o sistema cria a estrutura de dados necessária para guardar os dados provenientes da REST API, Figura 6.

De seguida, configurou-se os restantes passos necessários. Através de *drag-and-drop* construiu-se a interface a ser mostrada ao utilizador (Figura 7) com a inclusão de um título, um botão e uma tabela onde os dados derivados da REST API serão inseridos.

Para finalizar configurou-se a ação do botão inserido na interface do utilizador (Figura 8). O sistema permite a inserção de várias ações ao clicar no botão. Para a demonstração foi necessário utilizar duas, a chamada à REST API e o armazenamento dos dados no sistema (Figura 8).

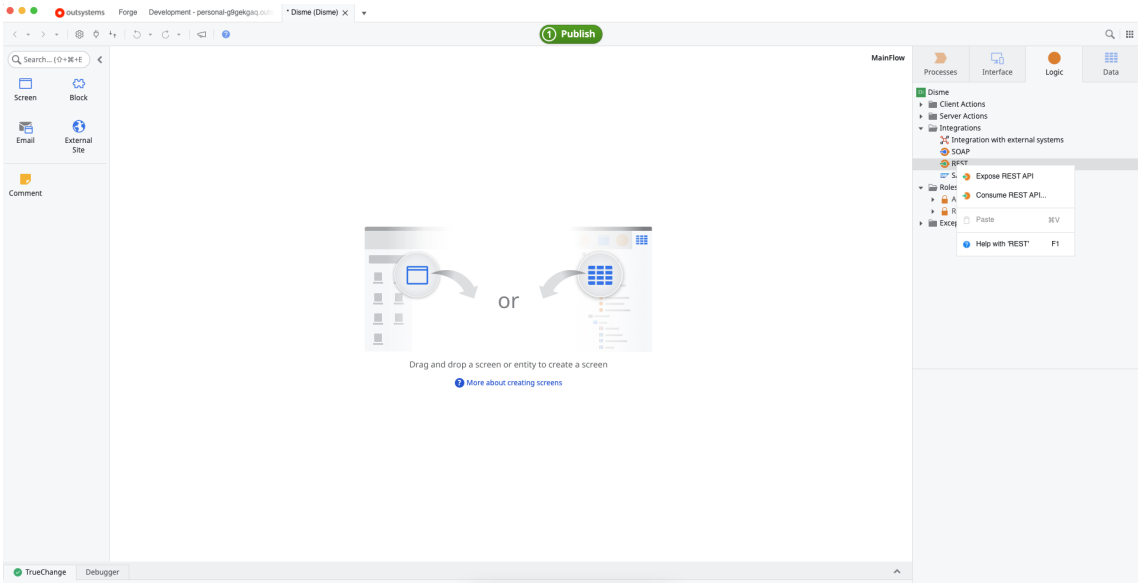

Figura 4. Interface do utilizador da plataforma OutSystems.

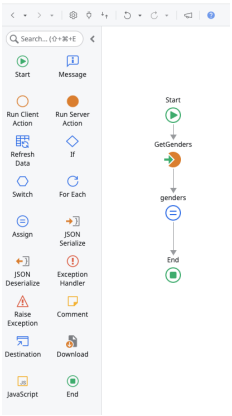

Figura 8. Configuração do fluxo de ações efetuadas no momento em que o utilizador clica no botão.

A Figura 9 mostra o resultado final da aplicação para o consumo de dados. Após o utilizador clicar no botão "Pesquisar", o sistema realiza a chamada à REST API e coloca os respetivos dados na tabela exposta ao utilizador.

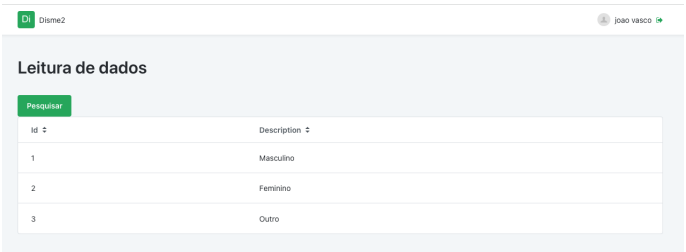

Figura 9. Interface do utilizador após o mesmo ter feito um pedido de pesquisa utilizando o botão Pesquisar.

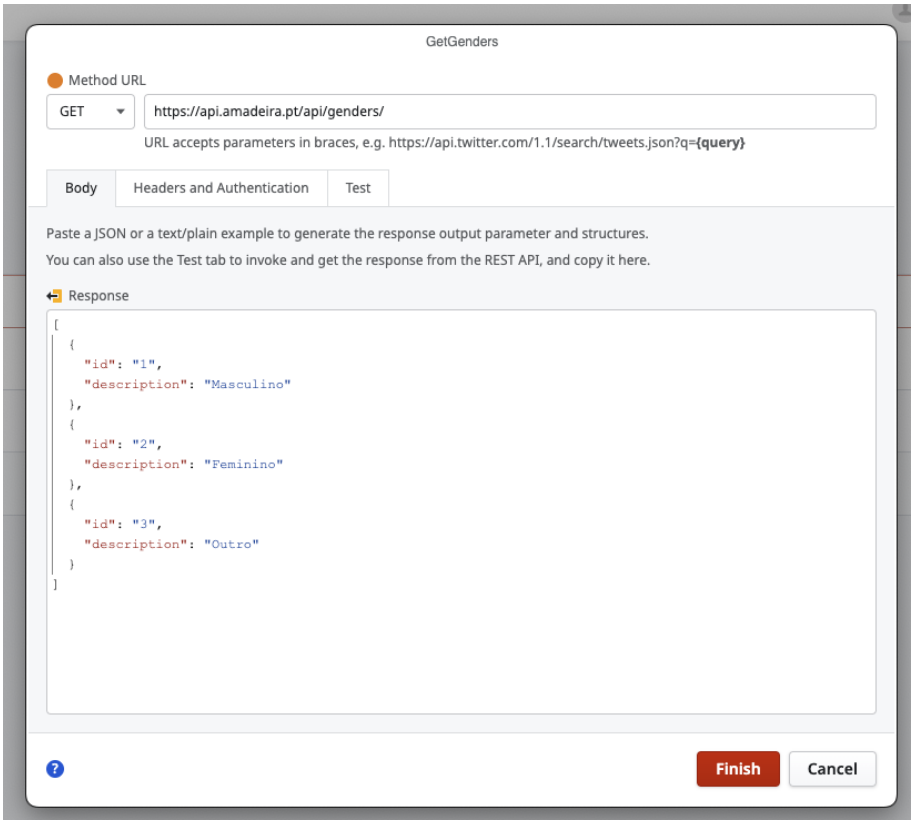

Figura 5. Introdução e configuração de pedidos à API.

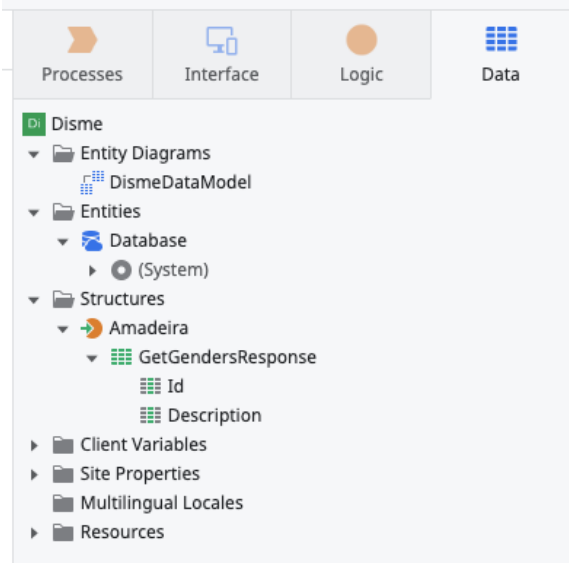

Figura 6. Identificação da estrutura da resposta da API pelo sistema.

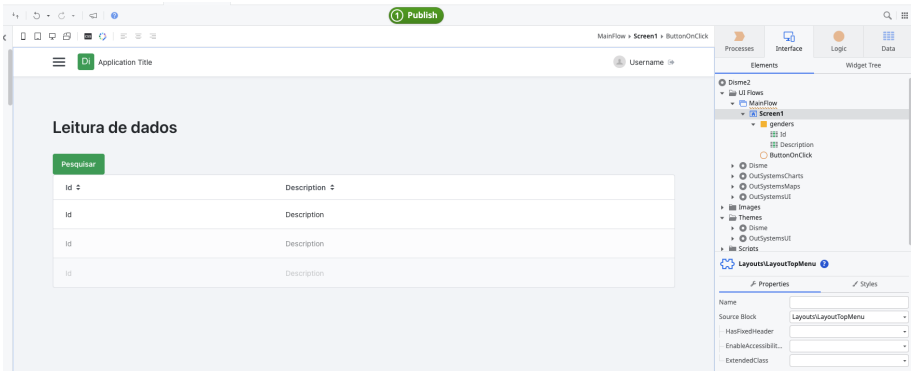

Figura 7. Construção da interface do utilizador.

Ao finalizar os testes propostos identificou-se pontos fortes como:

1. **Usabilidade e interface:** a plataforma proporciona uma fluída e confortável experiência aos utilizadores. A interface destaca-se pelo design e experiência visual para o utilizador.
2. **Facilidade de utilização:** para utilizadores com pouca experiência na plataforma, torna-se fácil explorar e realizar ações.
3. **Automação:** capacidade de realizar chamadas de teste à API automaticamente para identificar a estrutura de dados, economizando tempo.

Por outro lado, a configuração do fluxo de ações do botão "Pesquisar"foi desafiadora, pois após a sua configuração as ações desejadas não executavam conforme esperado e as mensagens de erro não continham toda a informação necessária para solucionar o problema.

2.2.1.2 Mendix

A segunda plataforma a ser analisada é a Mendix. De maneira a analisar as plataformas de forma similar, tentou-se replicar o exemplo de leitura de dados da plataforma anterior. Inicialmente, foi aberta a interface da plataforma Mendix (Figura 10).

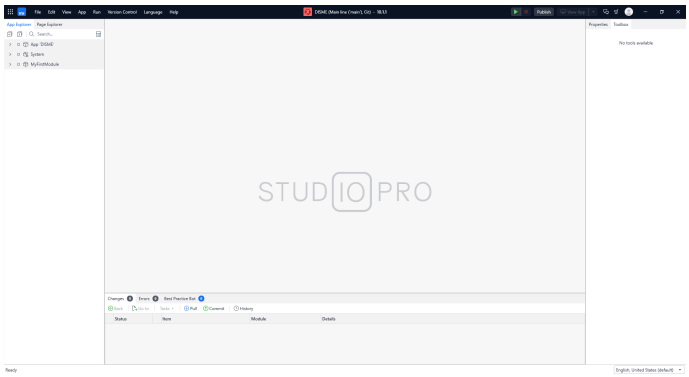

Figura 10. Interface do utilizador da plataforma Mendix.

Para configurar o serviço de chamada para uma REST API nesta plataforma foi necessário fornecer previamente a estrutura de resposta. Acedeu-se ao URL da REST API e introduziu-se a

resposta em formato JSON no sistema através da criação de uma componente denominada "*JSON Structure*". Na Figura 11 é possível observar a estrutura da resposta introduzida manualmente.

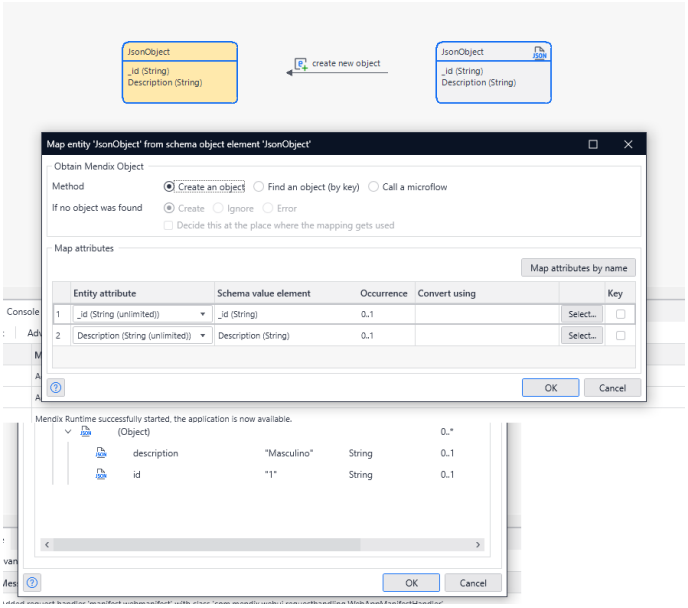

Figura 11. Criação da estrutura em JSON da resposta da API.

Após inserir manualmente a estrutura e o sistema detetar a mesma, é necessário criar uma entidade local. Para isso é possível realizar um *mapping* automático, como demonstrado na Figura 12, que transforma a estrutura JSON num objeto.

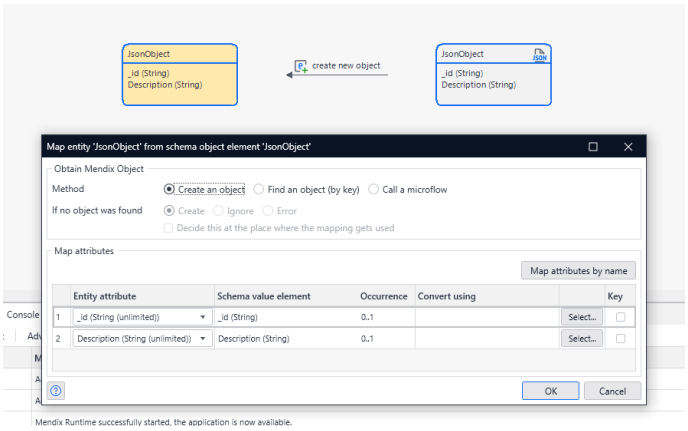

Figura 12. *Mapping* de uma estrutura JSON em uma entidade local.

Com a estrutura da resposta identificada e a entidade local que guardará os dados temporariamente criada, passou-se à configuração do serviço REST API. A configuração da componente "*Call REST Service*"dividiu-se em duas partes; *General* (Figura 13) e *Response* (Figura 14).

No separador *General* foi necessário indicar o URL da REST API, enquanto que no separador *Response* foi necessário selecionar que a resposta seria tratada pelo *mapping* que foi definido no início do processo.

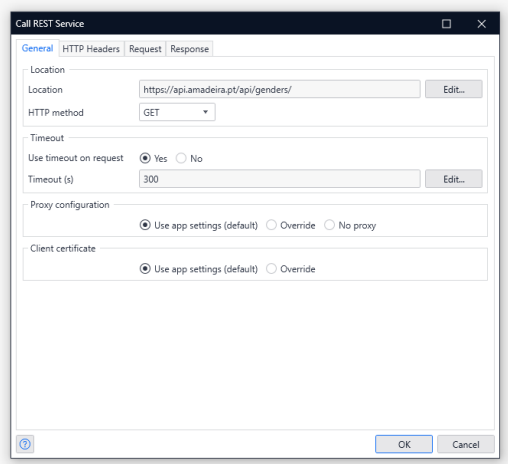

Figura 13. Configuração do serviço de chamada para REST API.

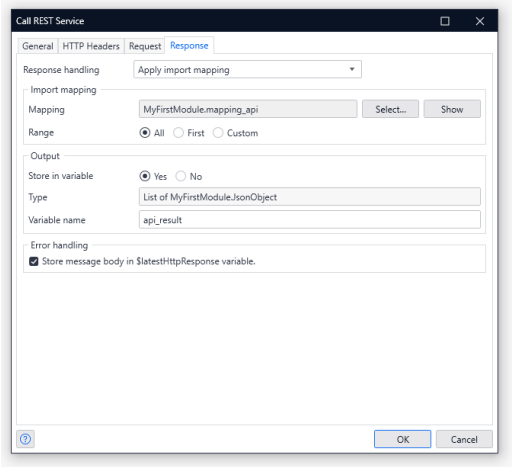

Figura 14. Configuração do serviço de resposta para REST API.

Depois do *Call REST Service* estar configurado, é necessário integrá-lo com uma ação no sistema. Para esse propósito, definiu-se um *microflow* que representasse a chamada da API como é possível verificar na Figura 15. Desta forma, ao construir a interface final, associamos o *microflow* a uma lista, conforme demonstrado na Figura 16.

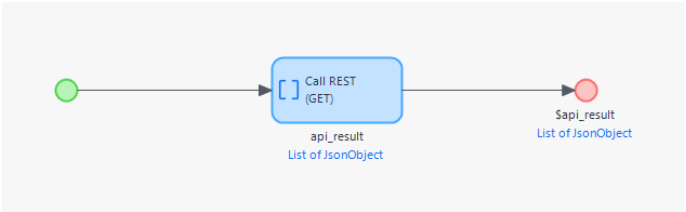

Figura 15. Configuração de um *microflow*, conjunto de ações efetuadas no momento da sua utilização.

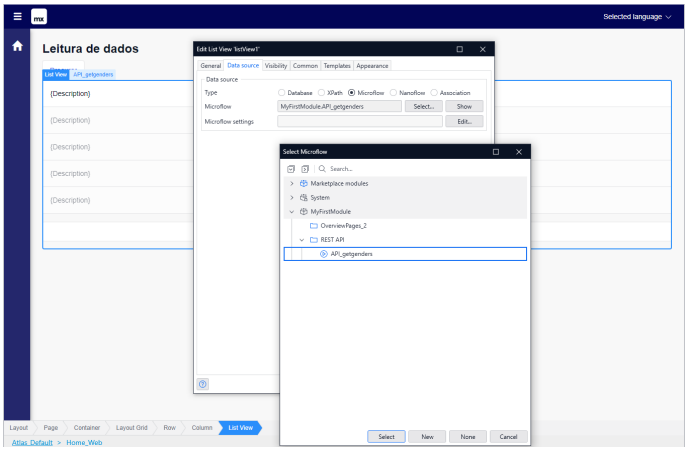

Figura 16. Construção da interface do utilizador.

Ao finalizar toda a configuração necessária, obtivemos a interface presente na Figura 17, uma página com uma lista de dados provenientes da REST API.

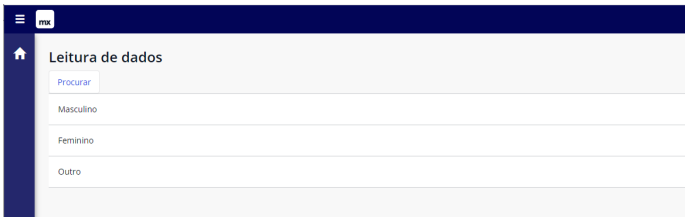

Figura 17. Interface do utilizador após o mesmo ter acedido à página que executa o pedido.

Apesar dos testes terem sido concluídos, existiu dificuldade na construção e desenvolvimento do processo. A plataforma *Mendix* falha na experiência de utilização devido ao facto de a sua interface ser confusa e, em consequência, pouco intuitiva, o que dificulta a sua utilização eficiente por utilizadores pouco experientes. Adicionalmente, a vasta gama de componentes oferecidos pela plataforma acaba por tornar a sua utilização ainda mais complexa.

Comparando com a plataforma *Outsystems*, o processo de identificação da estrutura da resposta da API na plataforma *Mendix* é inverso e manual. Enquanto que na plataforma *OutSystems* o utilizador necessita apenas de colocar o URL e o sistema identifica a estrutura, na plataforma *Mendix* é necessário definir primeiro a estrutura e posteriormente o URL.

De forma geral, para quem a utiliza pela primeira vez, a plataforma é consideravelmente complexa. Contudo, para utilizadores com alguma experiência, oferece uma variada gama de ferramentas interessantes numa LCP.

2.2.1.3 Budibase

A Budibase é uma LCP que é projetada para ajudar no desenvolvimento de ferramentas internas de forma rápida que pode ser alojada localmente ou em *cloud* [23]. À semelhança das plataformas mencionadas anteriormente, a plataforma Budibase fornece várias ferramentas ao utilizador, nomeadamente fontes de dados, *design* e automações.

A análise desta plataforma focou-se na repetição dos testes realizados nas restantes plataformas, pelo que se procurou criar uma página de consumo de dados de uma REST API.

Ao aceder à interface da plataforma, Figura 18, e ao selecionar a aba de fontes de dados podemos observar todas as opções suportadas pela mesma.

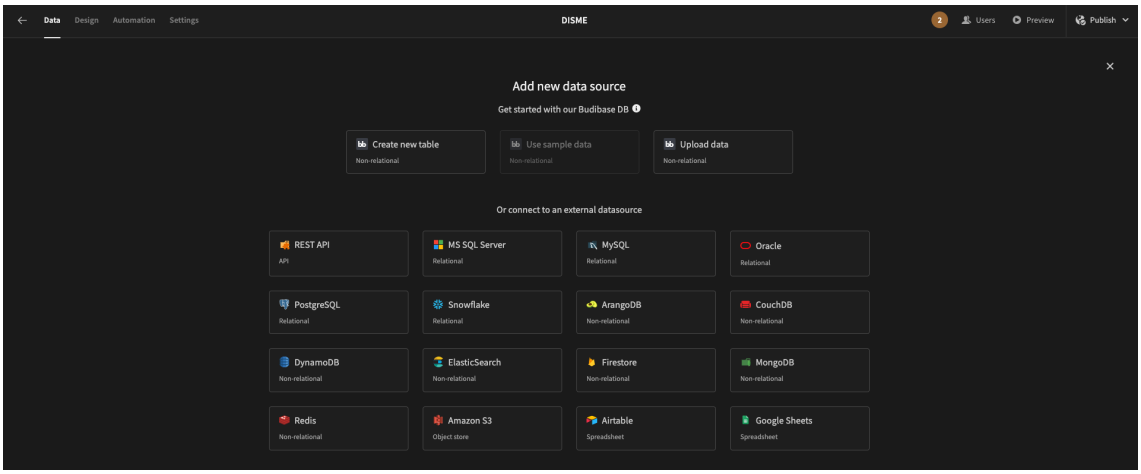

Figura 18. Interface do utilizador da plataforma Budibase.

Selecionou-se a opção REST API e de seguida inseriu-se o URL. Posto isto, a plataforma identificou automaticamente a resposta vinda do URL inserido e devolveu o JSON (Figura 19).

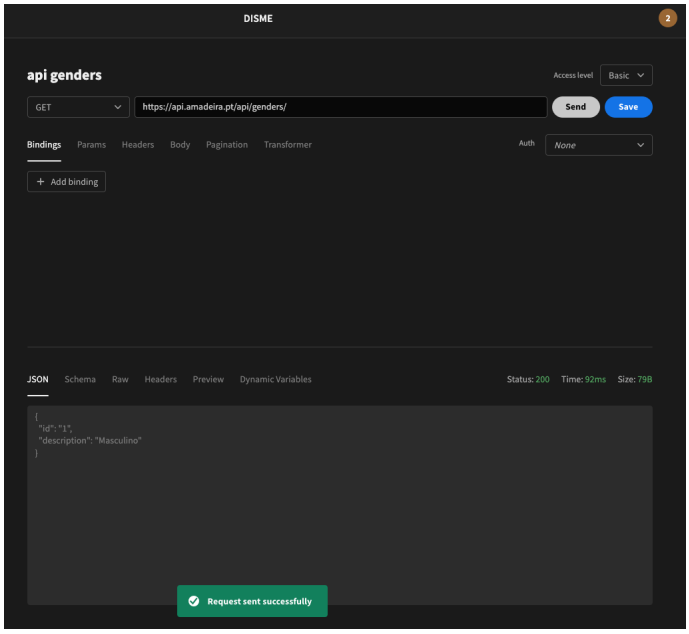

Figura 19. Configuração de uma fonte de dados utilizada no teste da ferramenta.

Na Figura 20 pode-se observar a configuração da interface do utilizador, na qual foi inserido um botão "Pesquisar"que executa a chamada à REST API e uma tabela que agrega o resultado numa lista de dados. A Figura 21 mostra a interface final após a execução da pesquisa.

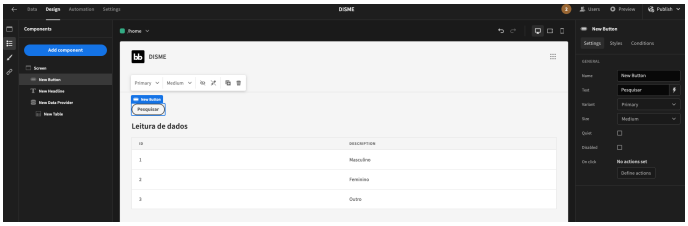

Figura 20. Construção da interface do utilizador.

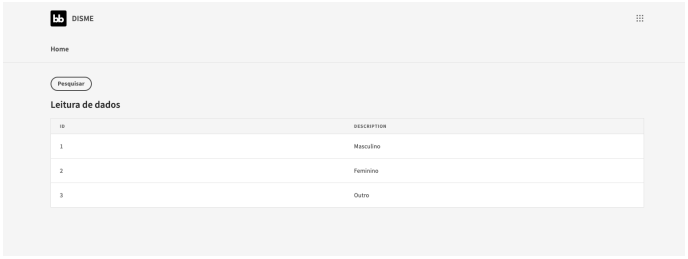

Figura 21. Interface do utilizador após o mesmo ter acedido à pagina de execução do pedido.

Comparando com as restantes plataformas, a plataforma *Budibase* foi aquela que necessitou de menos etapas e consequentemente menos tempo, para realizar o pretendido. Os pontos que se destacam nesta plataforma são:

1. **Facilidade de utilização:** utilizar a plataforma torna-se fácil e intuitivo para qualquer utilizador.
2. **Automação:** capacidade de realizar chamadas de teste à API automaticamente para identificar a estrutura de dados, evitando possíveis erros e economizando tempo.
3. ***Open-source*:** permite ao utilizador hospedar a plataforma localmente e personalizar a mesma consoante as suas necessidades específicas.

2.3 Dynamic Information Systems Modeller and Executer (DISME)

Dynamic Information System Modeler and Executer (DISME) é uma *low-code platform* construída e desenvolvida por uma equipa constituída por bolseiros e mestrandos no Enterprise Engineering Lab (eelab) que recorre a tecnologias/*frameworks* inovadoras como Hypertext Preprocessor (PHP) (Laravel[4]), Javascript (JS) (Angular[5] e Bootstrap[6]) para o seu desenvolvimento.

Na Figura 22 podemos observar a interface atual do DISME que, atualmente, não contém nenhuma componente de interoperabilidade que promova o transporte de dados entre o mesmo e

[4] `https://laravel.com/`
[5] `https://angular.io/`
[6] `https://getbootstrap.com/`

sistemas externos. Desta forma, como mencionado anteriormente, a componente desenvolvida no âmbito desta dissertação foi concebida no contexto do projeto DISME. O seu foco foi exclusivamente na recolha de dados externos e na gestão e manipulação dos mesmos na *low-code platform* DISME.

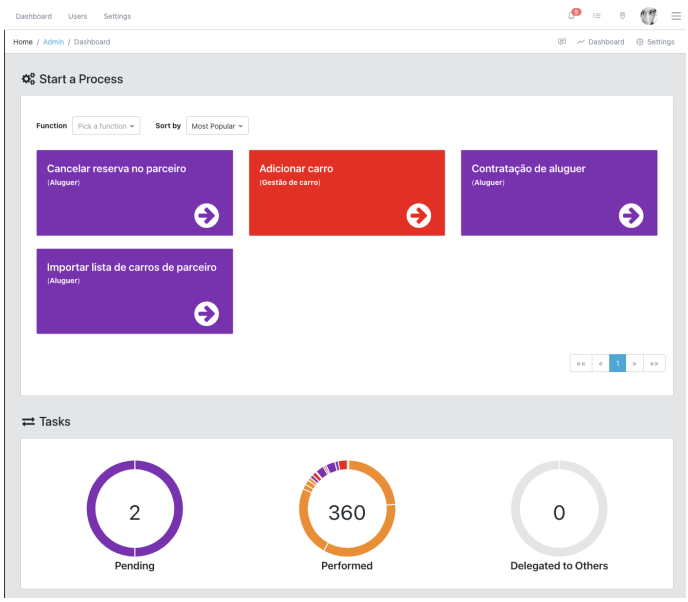

Figura 22. Interface do utilizador no DISME

2.3.1 Conceitos básicos

No que diz respeito aos conceitos fundamentais, é essencial compreender o funcionamento das transações no contexto do DISME. Uma transação é uma operação que pode ser realizada em uma ou mais instâncias de uma entidade. Pode ser composta por um ou vários passos que envolvem a introdução de dados do utilizador, produção de resultados ou operações automatizadas de negócio. As transações possuem propriedades específicas, como tipo, estado, etapa e requisitos, estas são detalhadas no DISME.

Normalmente utilizadas para criar, atualizar, eliminar ou consultar entidades no DISME, as transações são projetadas para garantirem a consistência do modelo mesmo em caso de erros ou problemas durante a execução, esta consistência é mantida pela execução integral e indivisível da transação. Assim, as transações representam uma forma essencial de modelar as operações complexas no âmbito do DISME, constituindo um dos pilares fundamentais deste modelo.

Outro conceito importante no sistema é a componente de regras de ação. Esta componente é responsável pela definição e implementação de lógica funcional num ambiente *low-code*, permitindo que os utilizadores, mesmo sem experiência prévia em programação, criem e controlem conjuntos, através de ferramentas *drag-and-drop*, de instruções designados de regras de ação. Estas regras determinam o comportamento do sistema em diferentes estados de transação, permitindo a execução de ações específicas em resposta a eventos ou condições predefinidas. A componente de regras de ação capacita os utilizadores a programar comportamentos complexos sem a necessidade de lidar diretamente com código de programação, facilitando a gestão e a execução de processos nas organizações. [24]

3 Implementação

Este capítulo foca-se em explorar e implementar as funcionalidades identificadas anteriormente nas três plataformas analisadas com o propósito de selecionar funcionalidades que devem ser utilizadas no desenvolvimento e implementação de uma nova componente de interoperabilidade no DISME. Com a análise que será realizada pretende-se identificar as características e funcionalidades essenciais para facilitar a criação e uso de REST APIs, procura-se definir requisitos e idealizar possíveis cenários práticos no contexto de um SI focado na gestão de uma rent-a-car.

3.1 Arquitetura da plataforma DISME

A *low-code platform* DISME é uma aplicação web composta por três serviços: o lado do cliente (*frontend*), o lado do servidor (*backend*) e o servidor de base de dados. O lado do cliente (*frontend*) é toda a parte que interage com o utilizador, recebe os seus *inputs*, controla as interações e lida com a lógica. Por sua vez, o *backend* inclui uma API RESTful que gere e processa todos os pedidos do utilizador. Por fim, o servidor de base de dados fornece os dados a todas as tarefas do sistema que necessitem dos mesmos, comunicando com a componente do *backend*.

Na construção da plataforma foram utilizadas diferentes tecnologias para cada uma das componentes. No *frontend*, inicialmente foi utilizado o AngularJS que, mais tarde, foi atualizado para o Angular, uma *framework open-source* baseada em *TypeScript*[7] criada pela Google[8]. No *backend*, a *framework* escolhida foi o Laravel[9], conhecida por ser uma *framework* de PHP que permite a gestão do lado do servidor, HTML e muito mais. Para o sistema de gestão de base de dados (SGBD) foi escolhido o MySQL[10] que é utilizado com o *Object Relational Mapper* (ORM) Eloquent.

O DISME é caracterizado por três componentes principais, como observado na Figura 22, que juntos fazem com que a *low-code platform* consiga executar todo o seu trabalho de maneira eficiente e eficaz. Estes componentes são:

1. ***Diagram Editor*:** este componente permite aos utilizadores criarem todos os modelos DEMO de alto nível de forma gráfica. Permite modelar toda a lógica de negócios, proporcionando uma visão clara da estrutura organizacional e das relações entre os diferentes processos de negócio.

2. ***System Manager*:** onde os detalhes necessários dos modelos DEMO são especificados e parametrizados. Permite criar e editar transações, regras de ação, formulários, especificação de objetos de negócio e os seus atributos. Garante a adaptação fácil e rápida às mudanças nos requisitos de negócios sem necessidade de alterar o código fonte.

3. ***System Executor*:** é responsável pela execução do sistema modelado em ambiente de produção. Os utilizadores participam nas transações de acordo com os seus papéis e seguem a metodologia DEMO.

[7] `https://www.typescriptlang.org/tg`
[8] `https://www.google.pt/`
[9] `https://laravel.com/`
[10] `https://www.mysql.com/`

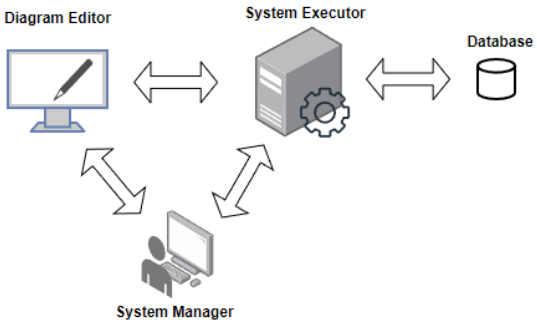

Figura 23. Arquitetura do DISME.

3.2 Levantamento de requisitos

Após a análise das três *low-code platforms* efetuada na secção 2.1 que proporcionou uma compreensão mais detalhada de cada plataforma e das suas capacidades e limitações, foram definidos requisitos importantes no desenvolvimento de uma nova componente de interoperabilidade no DISME. Para tal, começou-se com a identificação da necessidade de implementação das seguintes funcionalidades para a nova componente:

1. **Interface intuitiva**: a nova componente deve ser projetada para ser facilmente compreendida entre utilizadores mais técnicos e oferecer uma interface de fácil utilização, considerando os desafios encontrados em plataformas como a plataforma Mendix.

2. **Gestão de REST APIs**: deve ter a capacidade de criar, consumir e gerir REST APIs de outros SI.

3. **Identificação de Estruturas Automática**: deve incluir a possibilidade de entender a estrutura de resposta da API automaticamente, como observado na análise da ferramenta OutSystems.

4. **Ferramentas integradas**: funcionalidades *drag-and-drop* no design da interface para facilitar a construção e teste de interfaces e chamadas à API.

5. **Chamadas de teste à API de forma automática**: deverá realizar chamadas teste à API para verificar rapidamente se as suas chamadas à API estão bem configuradas.

6. ***Feedback* ao utilizador**: deve fornecer *feedback* imediato ao utilizador através de notificações, pop-ups ou outros métodos de comunicação visual. Estão incluidas mensagens de sucesso após execução de uma ação bem como alertas sobre erros ou falhas.

As funcionalidades referidas anteriormente foram transformadas em requisitos concretos para o desenvolvimento da nova componente de interoperabilidade no DISME. Estes requisitos são fundamentais para guiar o processo de implementação e garantir que a nova componente atenda às necessidades identificadas. Abaixo, apresenta-se a lista de requisitos resultantes da análise, organizados sob forma de uma tabela para melhor visualização e referência.

Tabela 1. Tabela de Requisitos Funcionais para a nova componente no DISME.

Identificação	Funcionalidade	Descrição
RF1	Gestão de REST APIs	O sistema deverá permitir criar, consumir e gerir REST APIs.
RF1.1	Gestão de REST APIs	O sistema deverá permitir criar REST APIs.
RF1.2	Gestão de REST APIs	O sistema deverá permitir consumir REST APIs.
RF1.3	Gestão de REST APIs	O sistema deverá permitir a especificação de parâmetros no *endpoint*.
RF1.4	Gestão de REST APIs	O sistema deverá permitir a especificação do método da chamada a ser realizada.
RF1.5	Gestão de REST APIs	O sistema deverá permitir a inclusão de conteúdo no corpo da chamada.
RF1.6	Gestão de REST APIs	O sistema deverá permitir armazenar a resposta da chamada em entidades temporárias no sistema.
RF1.7	Gestão de REST APIs	O sistema deverá permitir armazenar a resposta da chamada em entidades internas no sistema.
RF2	Identificação de Estruturas Automática	O sistema deve identificar a estrutura da resposta da API.
RF2.1	Identificação de Estruturas Automática	O sistema deve identificar a estrutura da resposta da API quando for um objeto.
RF2.2	Identificação de Estruturas Automática	O sistema deve identificar a estrutura da resposta da API quando for uma lista de objetos.
RF3	Ferramentas integradas	O sistema deve disponibilizar ferramentas de design com funcionalidades *drag-and-drop*.
RF3.1	Ferramentas integradas	O sistema deve utilizar o blockly como ferramenta de *drag-and-drop*.
RF4	Chamadas à API automáticas	O sistema deve realizar chamadas de teste à API.
RF4.1	Chamadas à API automáticas	O sistema deve permitir ao utilizador realizar chamadas de teste voluntariamente.
RF5	*Feedback* ao utilizador	O sistema deve fornecer *feedback* imediato ao utilizador.
RF5.1	*Feedback* ao utilizador	O sistema deve alertar o utilizador em caso de sucesso.
RF5.2	*Feedback* ao utilizador	O sistema deve alertar o utilizador em caso de falha.

Tabela 2. Tabela de Requisitos Não Funcionais para a nova componente no DISME.

Identificação	Funcionalidade	Descrição
RNF1	Interface Intuitiva	O sistema deve manter a interface uniforme.

3.3 Novas Funcionalidades

Percebeu-se numa primeira fase de análise de integração da nova componente que o seu desenvolvimento iria passar pelas três componentes principais do DISME: *Diagram Editor*, *System Manager* e *System Executor*. Posto isto, o principal desafio para permitir a interoperabilidade no DISME foi perceber como é que iria ser integrado com o atual sistema e como é que seria o fluxo de ações necessário ao utilizador para essa finalidade. Determinou-se que o desenho da chamada a uma

REST API seria realizado no contexto do *Diagram Editor*, mais concretamente na componente *Action Rules Management*.

A componente *Action Rules Management*, Figura 24, facilita a construção de regras de ação através de uma ferramenta de programação visual, "*drag-and-drop*", Blockly. Esta é uma solução que integra um ambiente de edição de código através de representações gráficas. Ao manipular blocos visuais que simbolizam elementos fundamentais na programação, como *ifs*, *whiles*, *loops* e variáveis, é possível construir ações no sistema.

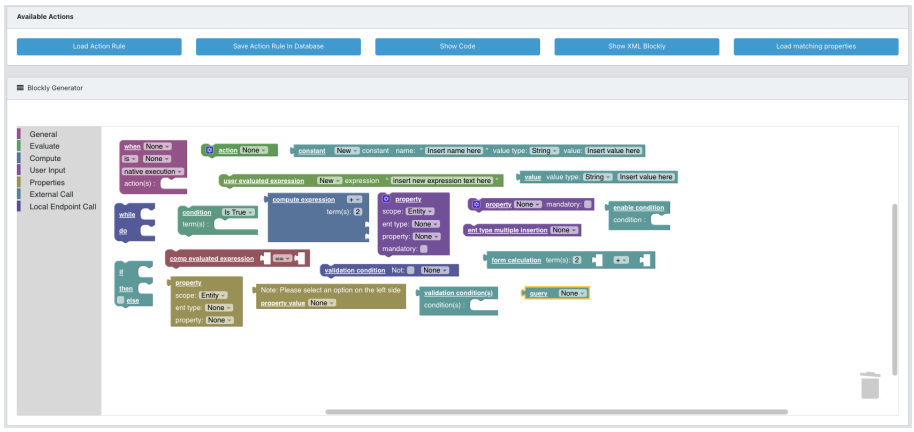

Figura 24. Interface do utilizador na componente Action Rules Management.

No contexto do DISME, esta componente pretende capacitar gestores e outros profissionais em posições semelhantes para conceber e implementar regras de ação. Uma vez desenvolvidas, as regras são armazenadas para serem executadas pelo *System Executor*, acessível através da componente *Dashboard*. Embora o Blockly tenha uma grande variedade de blocos, foi necessário criar e personalizar novos blocos para ajustar as necessidades no contexto do DISME. A ferramenta possui recursos que garantem que a sintaxe é respeitada e os blocos interagem corretamente. [24]

Iniciou-se a integração da nova componente dando continuidade ao desenho da peça "action". Esta peça permite ao utilizador selecionar, através de uma lista de opções, a ação que pretende desenhar, (Figura 25).

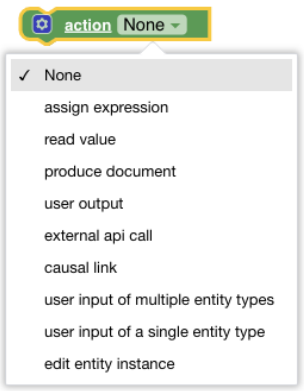

Figura 25. Lista de ações disponíveis na peça action.

Foi adicionada uma nova opção na lista de ações, a qual se designou por "external api call", como a ação responsável por realizar uma chamada a um sistema externo. Após selecionar a nova opção a peça altera o seu estado, adicionando vários *inputs* conforme definido na configuração da peça (Figura 26).

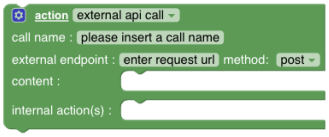

Figura 26. Configurações adicionais na peça action.

Na Figura 27 podemos observar um exemplo do novo estado da peça que incluí novos campos como:

Campos de preenchimento:

- Nome da chamada: este campo requer que o utilizador insira o nome que representa a chamada que será feita. No exemplo demonstrado: Importar lista de carros de parceiro.
- Endpoint externo: este campo é onde deve ser introduzido o URL ou endereço que a chamada será realizada. No exemplo demonstrado: `http://localhost:8001/partner/car/`.

Campos de seleção:

- Método da chamada: este campo permite ao utilizador selecionar o tipo de método HTTP a ser usado na chamada: Exemplo: POST, GET , PUT e DELETE.

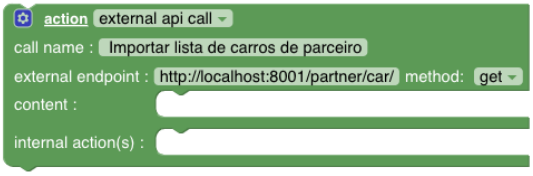

Figura 27. Exemplo de utilização dos campos adicionados no novo estado da peça action.

Além dos campos de preenchimento e seleção, que são definidos na própria peça, é possível encaixar outras peças, que serão abordadas de seguida, nos campos de junção, estas são:

- Conteúdo da chamada: este campo permite ao utilizador combinar várias peças de informação para formar o corpo da chamada. No exemplo da Figura 28, observamos a configuração de parâmetros e conteúdo no corpo que serão transmitidos no momento em que o pedido é realizado.
- Ações internas: este campo permite combinar peças que realizem ações internas no sistema. No exemplo da Figura 28, observamos duas ações distintas: mostrar resultados da realização da chamada e guardar dados resultantes da realização da chamada.

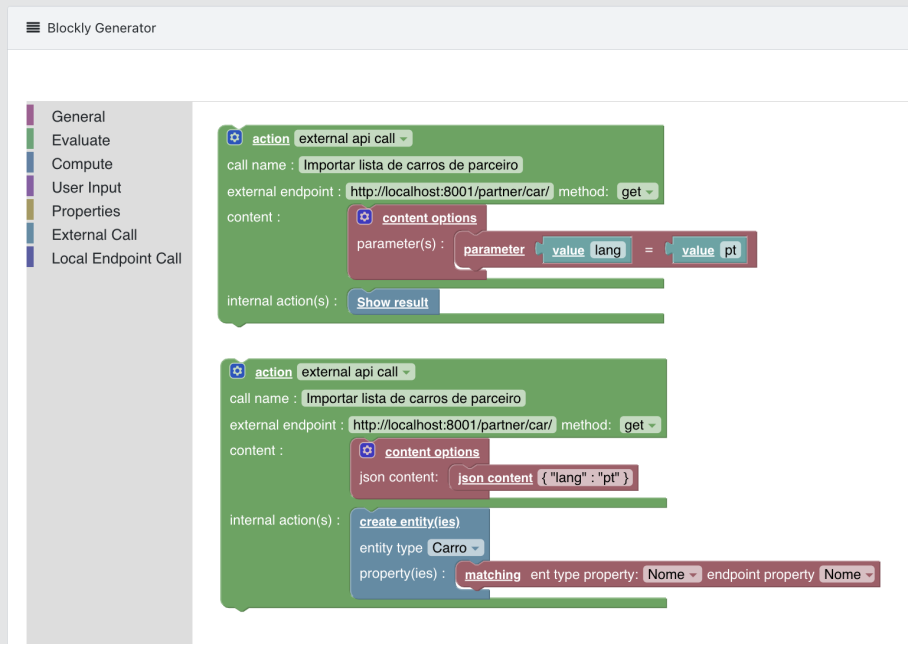

Figura 28. Exemplo de utilização dos campos adicionados no novo estado da peça action.

3.3.1 Novas peças associadas ao conteúdo da chamada

De modo a dar continuidade ao desenho da peça *action* com a nova opção "external api call"houve a necessidade de criar novas peças para ajustar as necessidades de uma chamada externa. As novas peças foram concebidas para serem incluídas no encaixe *content* da peça *action* que será explicado mais à frente.

3.3.1.1 Content options

O desenvolvimento desta peça foi pensado para ser uma ligação entre a peça *action* no espaço reservado ao *content* e os vários conteúdos que um pedido a uma API pode conter. Estes são parâmetros, (também conhecidos como parâmetros do URL ou *query strings*), autorizações, cabeçalho e corpo. A nova peça foi configurada para ser mutável, desta forma o utilizador pode decidir que tipo de conteúdo pretende ou não configurar. Nas Figuras 29, 30 e 31 é possível observar os três estados da peça.

Quando a peça sofre uma mutação, adiciona um campo de junção de outras peças na opção selecionada. As restantes peças serão abordadas de seguida.

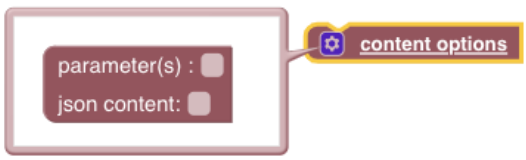

Figura 29. Estado inicial da peça *content options*.

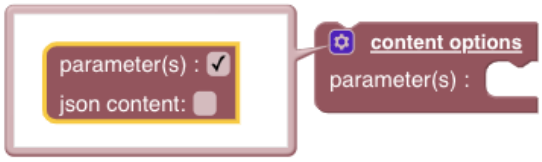

Figura 30. Peça *content options* com a opção dos parâmetros ativada.

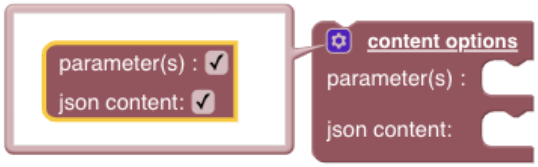

Figura 31. Peça *content options* com as opções dos parâmetros e *JSON content* ativadas.

3.3.1.2 JSON content

A peça *json content* permite ao utilizador inserir manualmente uma estrutura de dados no formato JSON. Esta peça serve de complemento à peça *content options*. Na Figura 32, observamos um exemplo de uma chamada externa em que pretendemos consultar os tipos de carro de um *endpoint* que recebe, através do corpo do pedido em formato JSON, a identificação do parceiro. A peça *json content* é utilizada para identificar no corpo do pedido do *endpoint* o ID do parceiro.

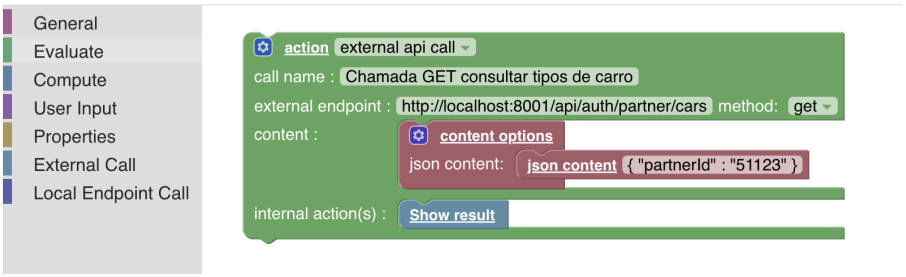

Figura 32. Exemplo de utilização da peça *json content* que permite a inserção de estruturas JSON no corpo da chamada.

3.3.1.3 Parameter

À semelhança da peça anterior, a peça *parameter* pretende auxiliar na construção de um pedido a uma chamada API com a adição de parâmetros no URL ou *query strings*. A peça permite múltiplas ligações, no caso de existir necessidade de configurar vários parâmetros como observado na Figura 33.

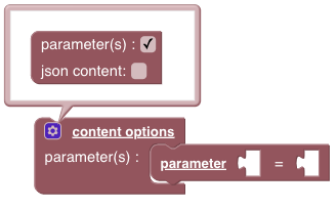

Figura 33. Peça *parameter* que permite incluir parâmetros na chamada.

A peça *parameter* recebe dois valores, à esquerda a *key* e à direita o *value*. Estes valores podem ser inseridos através de duas peças existentes na atual versão do DISME, *property* e *value*, Figura 34. A peça *property* possibilita ao utilizador escolher uma propriedade de uma entidade existente no sistema facilitando a interação com o mesmo. A peça *value* possibilita a inserção de valores pontuais e externos ao sistema.

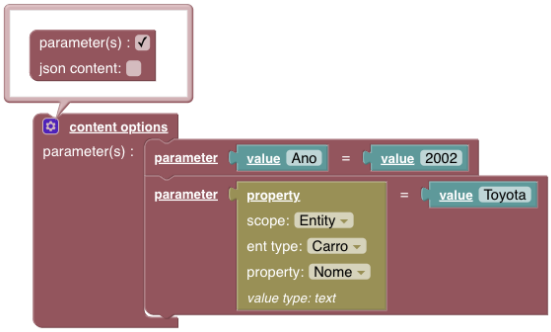

Figura 34. Exemplo de utilização da peça *parameter* com a peça *property* e *value*.

Na Figura 35, observarmos um exemplo de uma chamada externa para consultar os tipos de carro de um *endpoint* de um SI externo. Através da utilização da peça *parameter* é possível configurar a chamada de modo a incluir a identificação do parceiro.

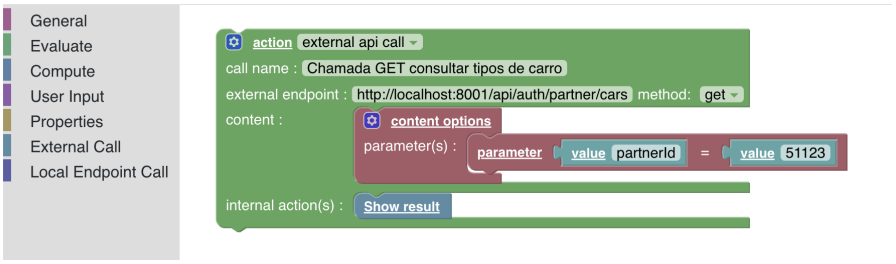

Figura 35. Exemplo de utilização da peça *parameter* juntamente com a peça *action*.

3.3.2 Novas peças associadas às ações internas

A realização de chamadas REST API a serviços de sistemas externos requerem sempre uma ação interna no sistema que executa a chamada. Estas ações são importantes pois permitem ao utilizador

dar seguimento ao conteúdo devolvido pela chamada. Nas secções seguintes serão especificadas as novas peças.

3.3.2.1 Show result

Na Figura 36 observamos a peça *show result* que foi criada com o propósito de mostrar ao utilizador, através de uma interface dedicada, o resultado proveniente da realização da chamada a uma REST API desenhada na peça *action*. A peça está apta a receber respostas nos formatos XML e JSON, sendo que estas serão sempre exibidas ao utilizador independentemente da estrutura devolvida na resposta.

Figura 36. Peça para realizar a ação interna de mostrar o conteúdo recebido da chamada a REST API.

Na Figura 37 observamos uma regra de ação que realiza uma chamada externa a um SI para visualização de uma lista de carros, esta regra de ação utiliza a peça *show result* como ação interna resultante da chamada. Esta ação irá exibir ao utilizador a resposta resultante da execução da regra de ação em formato JSON (Figura 38).

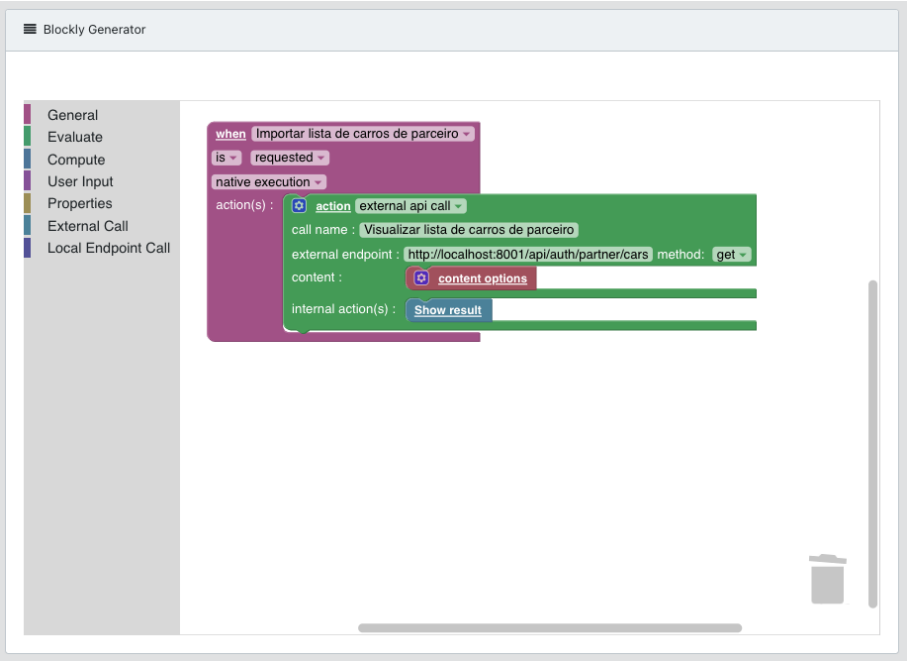

Figura 37. Utilização da ação interna *"Show result"* resultante da realização de uma chamada externa.

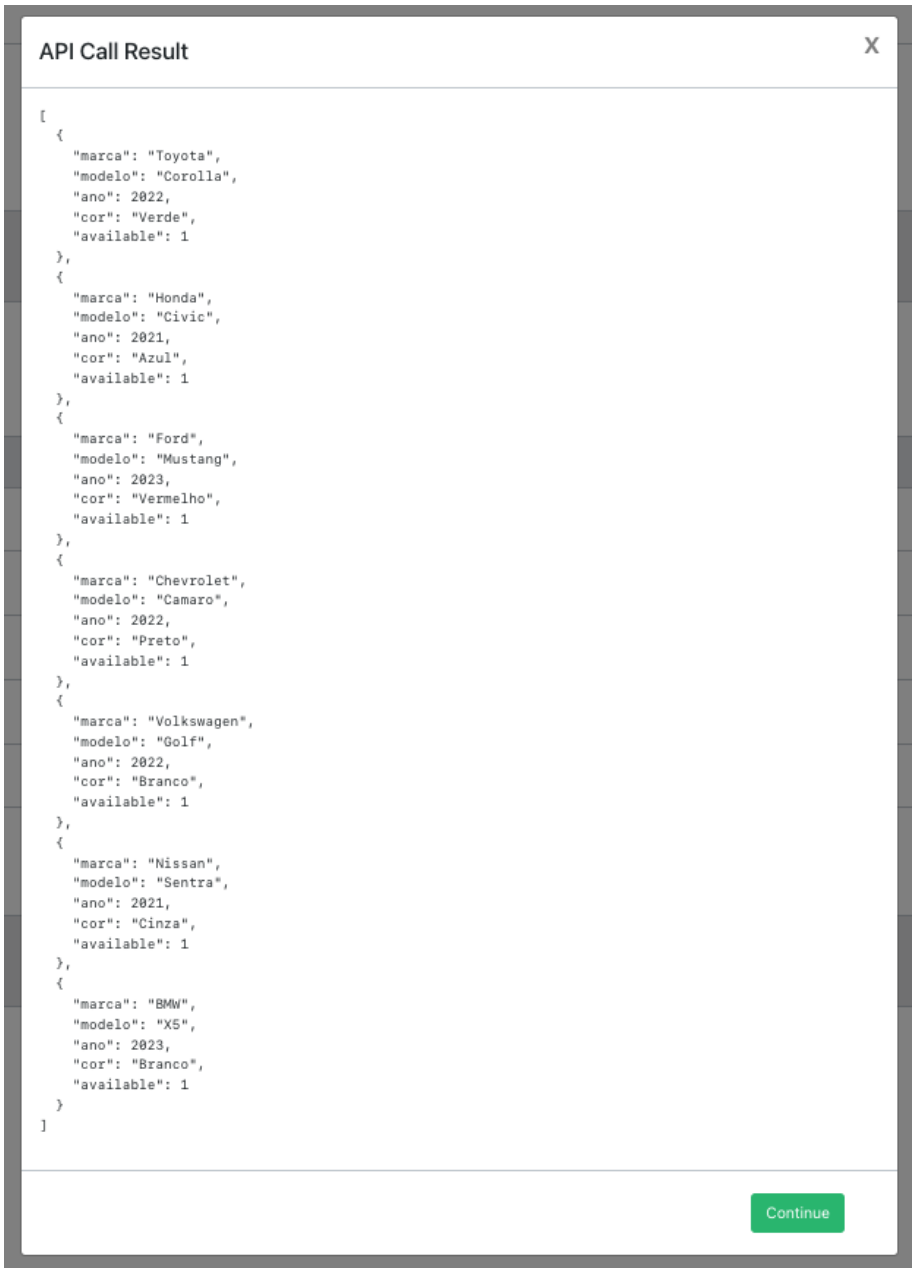

Figura 38. Interface resultante da ação "Show result".

3.3.2.2 Create entity(ies)

A peça *create entity(ies)* é composta por um campo de seleção e um campo de junção de peças (Figura 39). A sua utilização requer uma peça adicional que será mencionada de seguida.

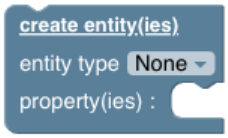

Figura 39. Peça create entity(ies).

Esta peça disponibiliza uma lista de tipos de entidades já existentes, como podemos observar na Figura 40, presentes no sistema. Desta forma é possível criar entidades do tipo selecionado e, posteriormente, guardar valores nas suas propriedades.

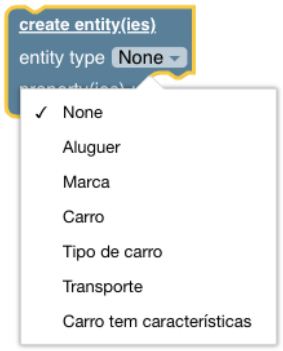

Figura 40. Lista de tipos de entidades disponíveis na criação de entidades.

3.3.2.3 Matching

A peça *matching*, presente na Figura 41, permite ao utilizador criar uma correspondência entre as propriedades da entidade selecionada na peça *create entity(ies)* e as propriedades existentes no conteúdo do resultado da chamada a REST API.

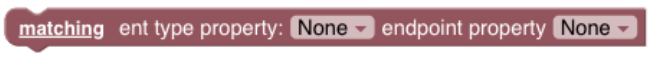

Figura 41. Peça matching.

A Figura 42 ilustra um exemplo isolado de uma correspondência de propriedades utilizando a peça matching. A lista disponibilizada no campo *"ent type property"* contém as propriedades da entidade selecionada na peça *create entity(ies)*. Ao utilizar várias peças matching e ao realizar a

correspondência de todas as propriedades existentes com as propriedades retornadas pela REST API é possível criar uma entidade interna ao sistema.

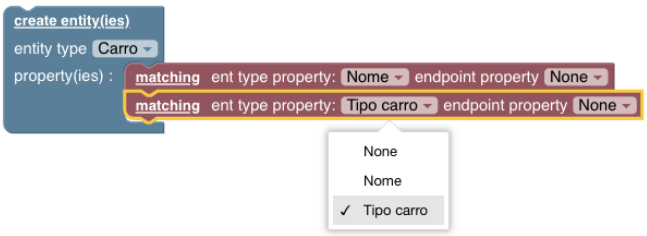

Figura 42. Lista de propriedades da entidade interna na peça *matching*.

Desta forma, no contexto de uma regra de ação podemos utilizar a peça *create entity(ies)* para criar uma entidade do tipo marca e posteriormente realizar o *matching* da propriedade externa com a propriedade interna *name*. Deste modo, ao executar a regra de ação presente na Figura 43 o sistema cria uma nova entidade do tipo marca e adiciona os valores na propriedade interna em que a correspondência foi realizada.

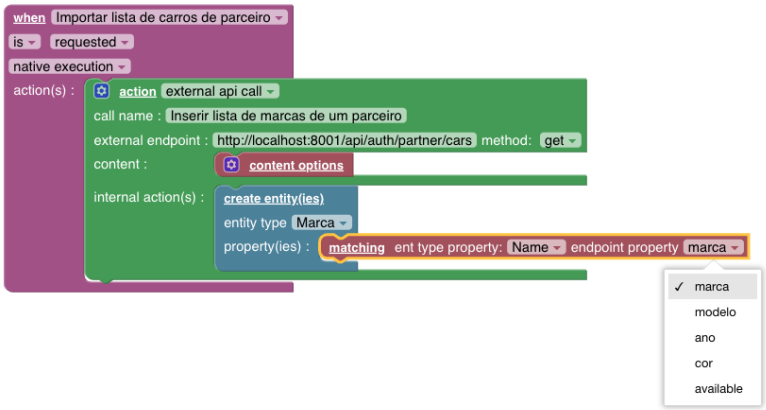

Figura 43. Utilização da ação interna *"create entity(ies)"* resultante da realização de uma chamada externa.

3.3.2.4 Create temporary entity(ies)

À semelhança da peça *create entity(ies)* mencionada anteriormente, a peça *create temporary entity(ies)* permite criar entidades no sistema. Porém com a particularidade de serem criadas entidades temporárias, ou seja, sem que as mesmas interfiram com as restantes entidades existentes no sistema. Desta forma, é possível guardar no sistema dados externos temporariamente a partir da criação de entidades temporárias onde os dados são armazenados, isolando-os apenas ao contexto da regra de ação atual.

A ação interna *create temporary entity(ies)* não necessita de nenhum tipo de campo adicional (Figura 44), servindo apenas para indicar ao sistema que deve criar entidades temporárias a partir do conteúdo do resultado da REST API.

Figura 44. Peça create temporary entity(ies).

Na Figura 45, observamos um exemplo de utilização da ação interna *create temporary entity(ies)*. Neste exemplo o sistema irá realizar uma chamada utilizando o método *GET* e obterá uma lista de carros através de um SI externo. Depois da realização da chamada, o sistema recebe a indicação que deve criar as entidades temporárias com os dados provenientes da mesma. Nesse momento o sistema tem armazenados temporariamente todos os valores resultantes da chamada.

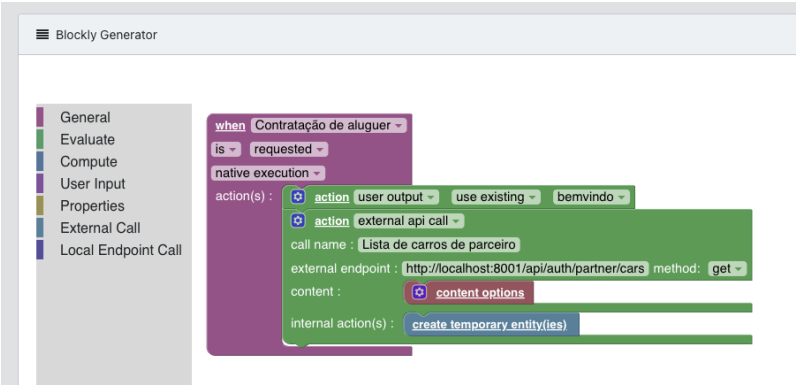

Figura 45. Exemplo de utilização da nova peça de ação interna do tipo *create temporary entity(ies)*.

3.3.2.5 temporary matching

Como verificado na Figura 46, a peça *temporary matching* é idêntica à peça *matching* na responsabilidade de fazer a correspondência entre as propriedades do sistema com as propriedades retornadas pela REST API.

Figura 46. Peça temporary matching.

Contudo, esta nova peça deve ser utilizada juntamente com outro tipo de regra de ação, *"user input of a single entity type"* como verificado na Figura 47. Isso permite ao utilizador, no momento em que a regra de ação é executada pelo *System Executor*, obter em tempo de execução os valores temporários para auxiliar na inserção de novas entidades.

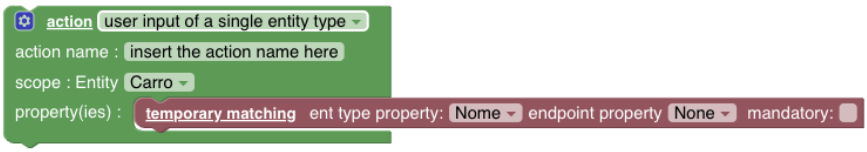

Figura 47. Utilização da peça temporary matching com a ação de inserir dados num tipo de entidade.

Na Figura 48 observamos uma regra de ação que representa o início de uma contratação de aluguer utilizando as peças *create temporary entity(ies)* e *temporary matching*. A primeira regra definida, do tipo *external api call*, indica ao sistema que os valores resultantes da execução da chamada devem ser armazenados temporariamente. A segunda regra, do tipo *"user input of a single entity type"*, indica ao sistema que serão inseridos valores pelo utilizador. As quatro primeiras propriedades serão inseridas manualmente pelo utilizador, no entanto ao realizar um *temporary matching* indicamos ao sistema que deve fornecer ao utilizador, sob a forma de lista, os valores temporários previamente armazenados da propriedade "marca".

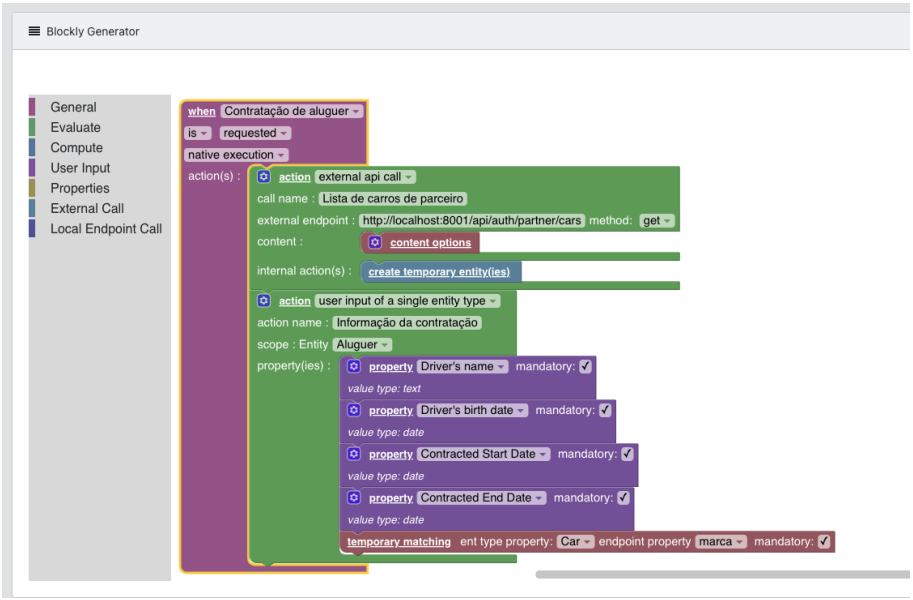

Figura 48. Exemplo de correspondência entre propriedades numa ação interna do tipo *create temporary entity(ies)*.

O sistema, ao terminar a execução da regra de ação, apaga todos os valores temporários que foram criados no contexto de execução da mesma.

As duas ações internas abordadas anteriormente, *create entity(ies)* e *create temporary entity(ies)*, permitem incluir no sistema mecanismos de *caching* pois possibilitam criar cópias localmente ou armazenar os dados em tabelas temporárias.

3.4 Sistema de desenho

Nesta fase de implementação, depois de concluído o desenvolvimento das novas peças, o DISME está preparado para o *design* de novas regras de ação que permitem a realização de chamadas externas a outros SI. As regras podem ser guardadas, editadas ou eliminadas pelo sistema. A componente *API Management* entra em ação de forma transparente e otimiza o processo do utilizador atualizando a lista de REST APIs na nova componente.

Nesta secção, iremos aprofundar o sistema de desenho atual de uma regra de ação que contenha uma chamada externa a um SI no DISME. Iremos demonstrar as etapas que um utilizador deve realizar para desenhar e guardar uma regra de ação, bem como algumas funcionalidades específicas implementadas para reduzir a complexidade do sistema.

3.4.1 Etapas e funcionalidades

Ao iniciar o processo de *design* de uma nova regra de ação é necessário que o utilizador aceda à componente *Action Rules Management*. A Figura 49 mostra como é a interface da componente. A primeira peça a ser utilizada deve ser a *"when"*, uma vez que esta peça permite ao utilizador identificar que transação pretende desenhar a regra de ação correspondente. No exemplo da Figura 49 utilizou-se a transação *List car from API*.

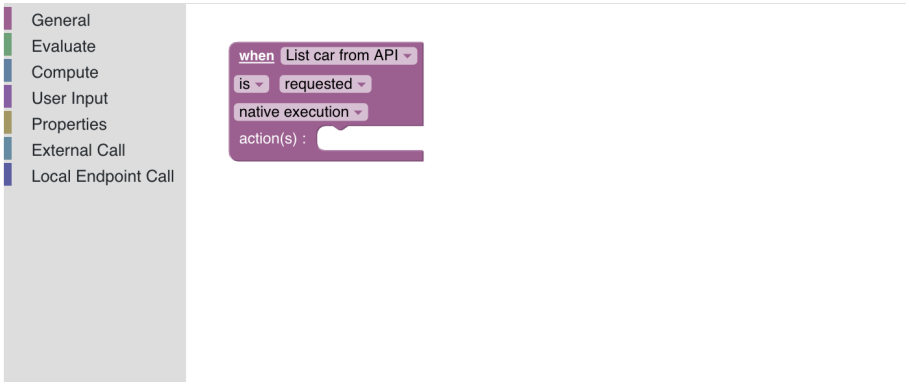

Figura 49. Peça *"when"* utilizada no inicio do desenho de uma regra de ação.

Em seguida o utilizador deve inserir a peça *"action"*, como podemos ver na Figura 50 e selecionar a opção *"external api call"* que significa executar uma regra de ação de chamada a uma REST API externa ao DISME, ou seja, a outro SI.

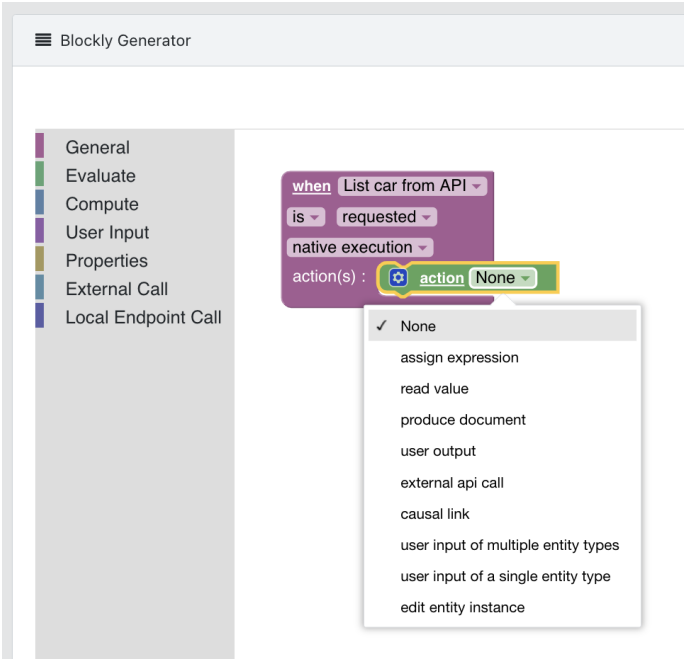

Figura 50. Exemplo de utilização da peça *"action"* com a peça *"when"*.

Com as peças *"when"* e *"action"* juntas (Figura 51), o utilizador pode configurar a sua chamada externa preenchendo o nome da chamada, o *endpoint* que deseja utilizar, o método a ser utilizado bem como o conteúdo adicional à chamada.

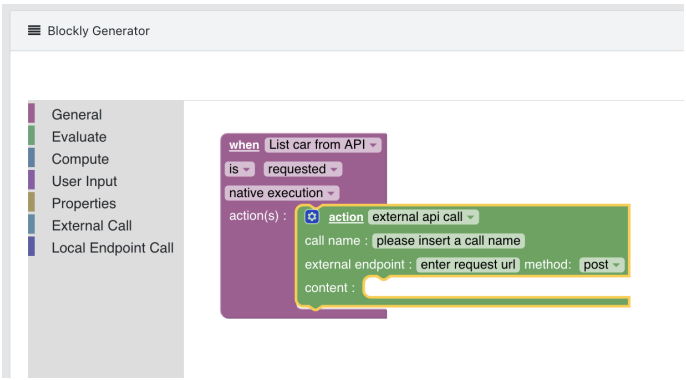

Figura 51. Peça *"when"* junta com a peça *"action"* para configuração do utilizador.

Na Figura 52, podemos observar um exemplo de configuração de uma chamada externa onde o objetivo é importar para o DISME uma lista de carros de um determinado parceiro. Para isso, configurou-se a ação com o método *"get"*, utilizou-se um *endpoint* e configurou-se os parâmetros de modo a obter apenas os carros disponíveis do parceiro.

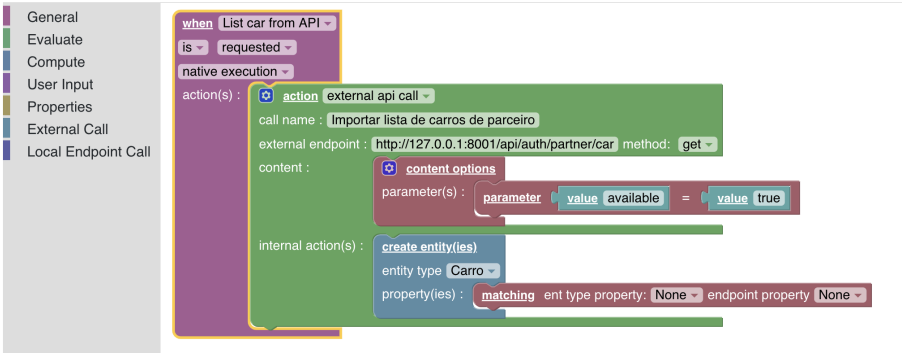

Figura 52. Exemplo de configuração de uma *external api call*.

A necessidade de desenvolvimento de uma nova funcionalidade surgiu no resultado da execução de uma REST API. Isto é, o resultado da execução nos casos em que o utilizador pretende ler e guardar dados externos obriga a que o DISME saiba previamente que tipo de dados irá receber. Esta nova funcionalidade tem como objetivo identificar a estrutura de dados do *endpoint* de modo a identificar os valores a serem colocados na propriedade *"endpoint property"*, sob a forma de lista, na peça *"matching"*.

De modo a contornar este problema criou-se um botão *"Load matching properties"*. A sua funcionalidade é realizar uma chamada de teste ao *endpoint* inserido pelo utilizador e verificar a estrutura presente na REST API. A Figura 53 mostra a estrutura de dados do exemplo utilizado onde cada carro é composto por 5 propriedades: marca, modelo, ano, cor e available.

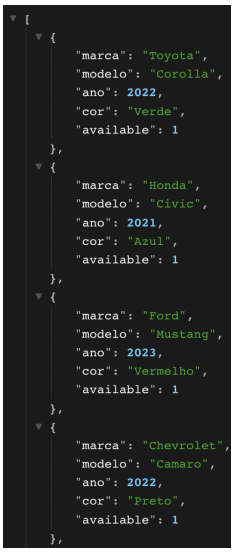

Figura 53. Exemplo de uma estrutura de dados de uma lista de carros.

Durante o desenho da regra de ação, o botão deve ser utilizado para que o sistema notifique o utilizador através de notificações *push* se o *endpoint* possui propriedades e, em caso de existência, importa-as para a lista na peça "matching"como observado na Figura 54.

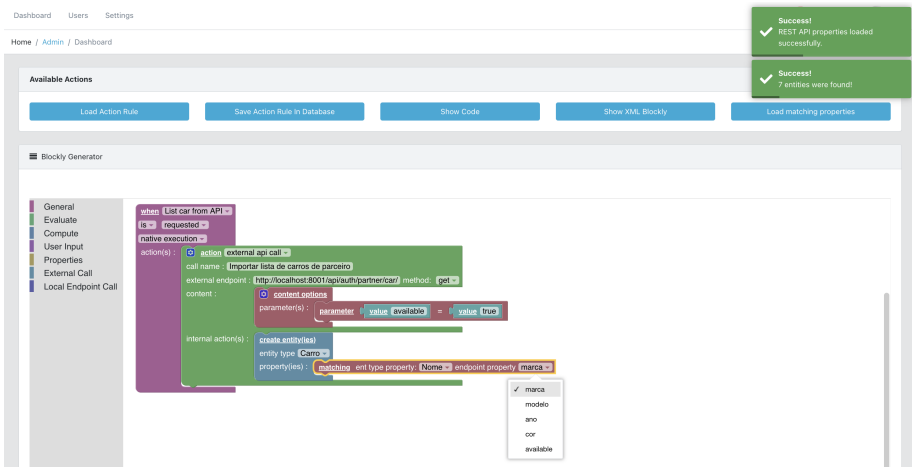

Figura 54. Novo botão *"Load matching properties"* para identificar a estrutura de dados.

Posto isto, o utilizador deve escolher a propriedade do *endpoint* que pretende realizar a correspondência com a propriedade interna ao sistema.

Após a conclusão do desenho da regra de ação, o utilizador deve finalizar o processo clicando no botão *Save Action Rule In Database*. Este botão permite que a regra de ação seja armazenada na base de dados. Para garantir a execução livre de erros das regras de ação através da componente *Dashboard*, o sistema realiza uma validação na estrutura da regra de ação antes de guardá-la na base de dados. Essa validação verifica se todas as peças estão presentes e se foram colocadas nos espaços corretos. Isso garante uma maior robustez no DISME.

A Figura 55 ilustra o momento em que uma regra de ação é validada e armazenada na base de dados sem qualquer erro. O sistema notifica o utilizador sobre esse sucesso por meio de uma notificação *push*.

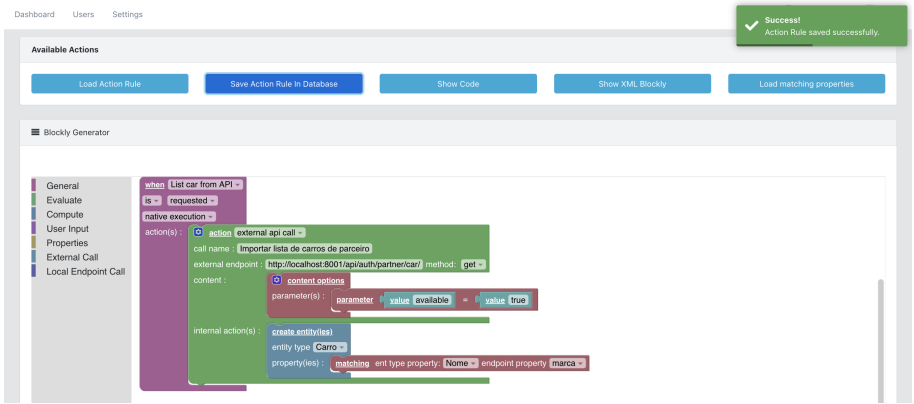

Figura 55. Notificação *push* ao armazenar uma regra de ação com sucesso.

Por outro lado, na Figura 56, podemos observar um cenário oposto, onde o sistema falha ao armazenar a regra de ação devido à falha de peças necessárias. Nesse caso, o sistema também notifica o utilizador por meio de uma notificação *"push"* informativa.

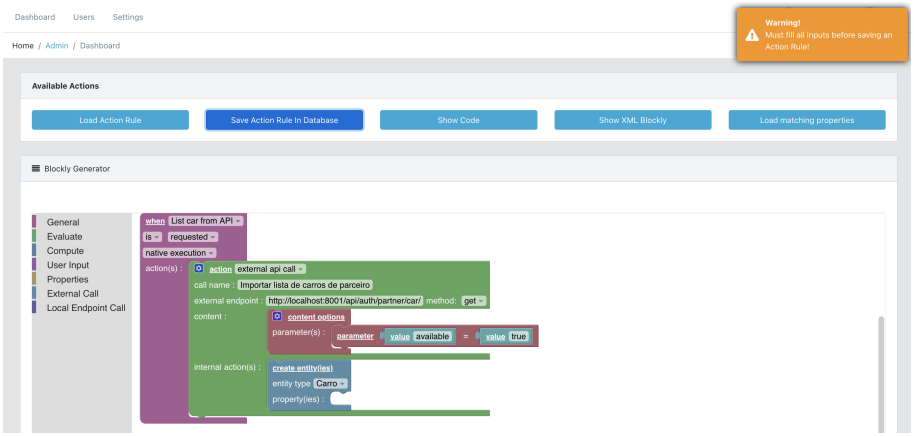

Figura 56. Notificação *push* ao armazenar uma regra de ação inválida.

3.5 Sistema de execução

Nesta secção, aprofunda-se o funcionamento interno do DISME, abordando detalhadamente o sistema de execução tendo como objetivo alterar o seu funcionamento com o propósito de incluir as alterações necessárias para que o sistema interprete e execute chamadas externas a outros SI.

Enquanto que anteriormente o foco do desenvolvimento foi maioritariamente a interface do utilizador com o desenvolvimento da nova componente *API Management* e alterações na componente *Action Rules Management*, é essencial entender e explorar de que forma o DISME coordena e executa as regras de ação definidas pelos utilizadores.

3.5.1 Identificação da ação

No âmbito da implementação, o *System Executor* é uma componente crucial do DISME responsável por interpretar e executar as regras de ação do modelo DEMO. Esta funcionalidade foi integrada anteriormente, pela equipa do DISME, com sucesso, na *dashboard* que facilita a gestão de tarefas e processos entre os vários utilizadores da plataforma.

O *System Executor* possui duas componentes principais: a *dashboard*, como podemos observar na Figura 57, que serve como interface do utilizador para interações organizacionais, e um *Execution Engine* que funciona como um controlador e executa todo o fluxo de informações e processos de acordo com a especificação completa do sistema.

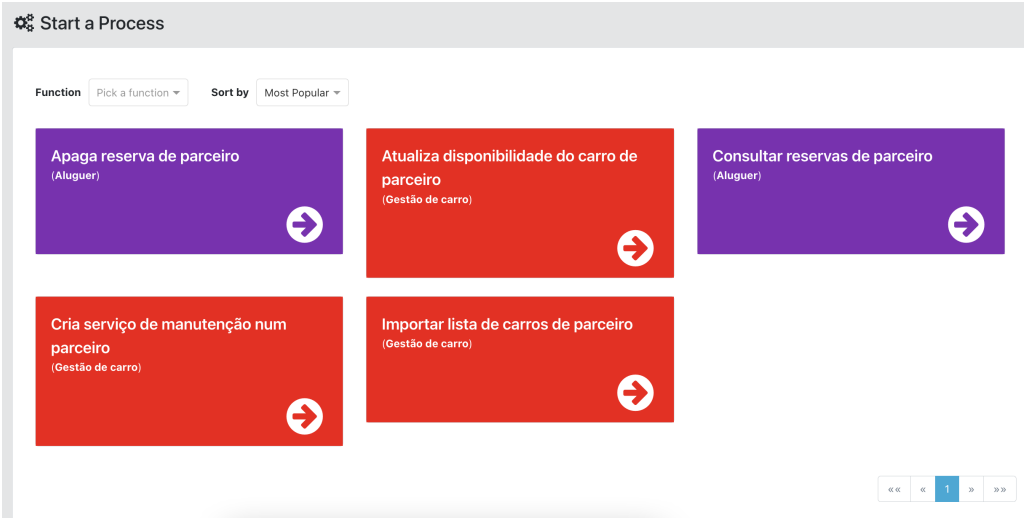

Figura 57. Interface do utilizador na *Dashboard*.

Na fase inicial do desenvolvimento, realizou-se uma análise detalhada do processo de execução do *Execution Engine* e da forma como o mesmo lida com o fluxo de informações e processos inerentes às regras de ação especificadas pelo sistema. Durante essa análise, identificou-se que a implementação da execução das regras de ação envolve a separação de todas as regras de ação existentes no sistema por meio de uma estrutura *switch case*. A expressão utilizada para avaliação no *switch case* é o nome da regra de ação selecionada na componente *"action"*. Isto é, para cada tipo de ação presente na lista de ações disponíveis pelo DISME, o *Execution Engine* interpreta essas ações de formas distintas.

Essa diferenciação na interpretação das ações resulta da comunicação contínua entre o *backend* e a base de dados, permitindo ao DISME, por meio do *Execution Engine*, gerir o fluxo de informações e processos em modo de produção. Conforme mencionado anteriormente, as regras de ação são projetadas no sistema através da componente *Action Rules Management* e posteriormente armazenadas. A interpretação de uma regra de ação torna-se possível devido à combinação das suas diferentes peças, permitindo ao DISME compreender a ordem de execução de cada regra de ação de forma precisa.

Essa comunicação eficaz entre o *backend* do DISME e a base de dados é fundamental para o funcionamento do sistema. Isso é alcançado através da utilização do *Object Relational Mapper* (ORM) do Laravel, que simplifica a consulta das informações necessárias para a interpretação das regras de ação. Estas consultas são executadas pelo *Execution Engine*, que atua como intermediário entre a interface do utilizador e os processos internos do sistema.

Esta estrutura de comunicação e execução permite que o DISME processe as regras de ação de forma eficiente, assegurando que as ações sejam realizadas na ordem correta e que os dados sejam manipulados de acordo com as especificações do utilizador.

3.5.2 Ações internas

Neste sentido, passou-se à integração da nova ação no *Execution Engine*. Inicialmente inseriu-se a nova ação do DISME *external api call* na estrutura *switch case*, permitindo dar seguimento ao tratamento de ações deste tipo.

Pelo facto do *Execution Engine* processar ações que resultem em alterações internas ao sistema, com a implementação da nova componente surgiu a necessidade de desenvolver as três novas ações internas associadas ao resultado da ação *external api call*, sendo estas: *show result*, *create entity(ies)* e *create temporary entity(ies)*.

Nas próximas secções abordaremos em detalhe a implementação das novas ações aprofundando as necessidades que existiram na sua integração, nomeadamente as principais alterações ao atual DISME.

3.5.2.1 *Show result*

A primeira ação a ser implementada foi a *"Show result"*. O objetivo desta ação é mostrar ao utilizador o conteúdo devolvido pela realização de uma chamada externa ao DISME. Como referido anteriormente, o *Execution Engine* ao executar este tipo de ação exibe uma interface dedicada ao utilizador com o resultado proveniente da chamada.

Para tratar de todos os pedidos externos ao DISME foi necessário criar um serviço adicional, *API Request Service*, que facilitasse a execução dos pedidos. Este serviço permitiu definir configurações importantes e comuns em todas as chamadas como os *headers*, o URL, o conteúdo pertencente ao corpo do pedido e o tipo de chamada a ser realizado. Foram desenvolvidas quatro funções correspondentes aos quatro métodos de chamada existentes: *POST, GET, PUT e DELETE*, com as configurações comuns para que pudessem ser utilizadas no tratamento de ações do tipo *external api call*. Este serviço permitiu criar uma camada de abstração e reduzir a duplicação de código no sistema.

Juntamente com o novo serviço, para implementar esta ação no *Execution Engine* foi necessário desenvolver uma nova API de comunicação entre a base de dados e o *backend* com o propósito de identificar a *external api call* correspondente, ou seja, através do *id* da regra de ação que está a ser executada o sistema consegue identificar em tempo real a ação *external api call* associada à mesma. Desta forma é possível obter em tempo de execução as informações da REST API na base de dados tais como o *endpoint*, o método a ser utilizado, os parâmetros do URL e o conteúdo adicional em *JSON*.

Após os dados serem recolhidos pela API o *Execution Engine* constrói o *endpoint* da chamada através da junção do *endpoint* com os parâmetros do URL, em caso de existência, e através de um *switch case* verifica qual é o método que deve ser executado e chama a função correspondente do serviço *Api Request Service*. No momento em que a função é executada, a chamada a outro SI é realizada e a sua resposta é exibida ao utilizador através de uma interface criada para o efeito.

Na Figura 58 podemos observar um exemplo de uma regra de ação que executa uma chamada externa utilizando a nova ação *external api call*. Neste exemplo em concreto chamou-se ao tipo de ação *external api call* de "Visualizar lista de carros de parceiro". O *endpoint* utilizado foi criado para simular um SI externo que disponibiliza uma lista de carros, o método utilizado foi o de leitura, *get*. Não foram adicionados conteúdos extra à chamada e a ação interna esperada foi mostrar os resultados provenientes da chamada.

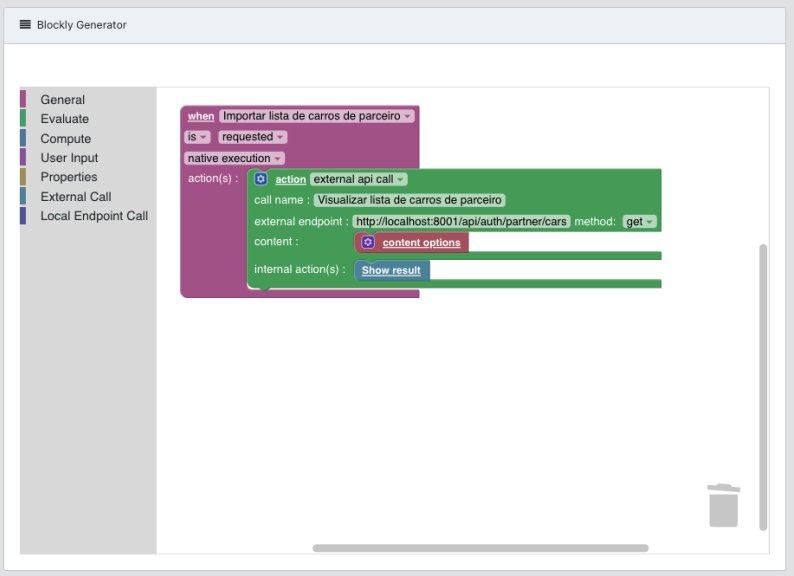

Figura 58. Exemplo de uma regra de ação que utiliza a ação *"Show result"*.

Após desenhar e armazenar a regra de ação o utilizador deve aceder à *dashboard* onde irá iniciar o processo que corresponde à regra de ação desenhada. Na Figura 59 podemos observar o botão em forma de seta que inicia o processo "Importar lista de carros do parceiro".

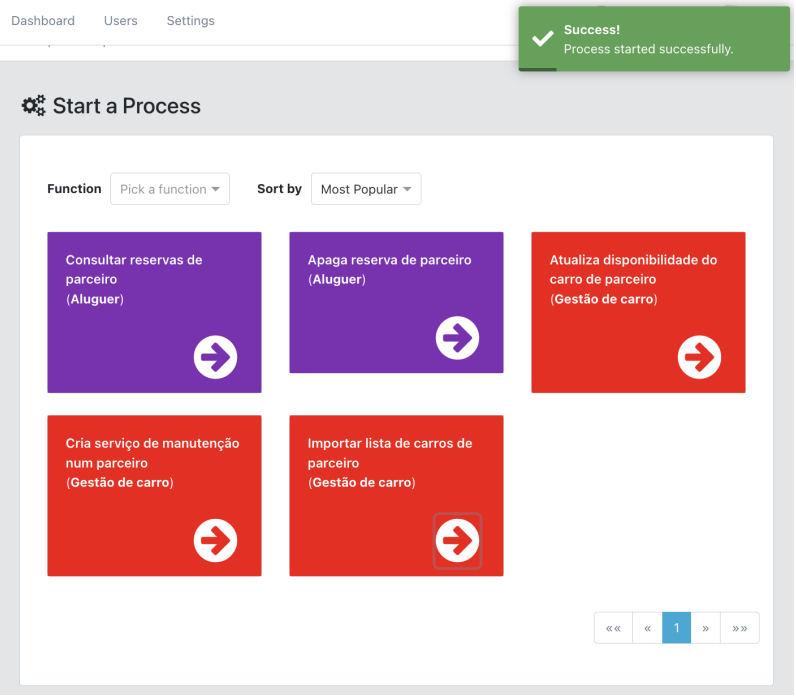

Figura 59. Interface da dashboard com o exemplo de regra de ação que utiliza a ação *"Show result"*.

Assim que o processo é iniciado o utilizador recebe uma notificação de sucesso e é criado um processo na lista de tarefas pendentes ao utilizador. Ao clicar na seta correspondente ao processo,

como observado na Figura 60, o *Execution Engine* inicia o tratamento do fluxo de informações e processos de acordo com a especificação da regra de ação.

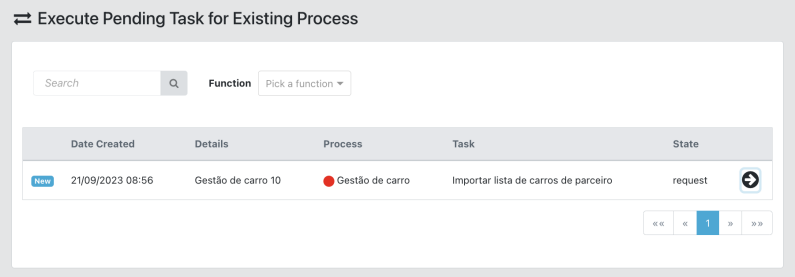

Figura 60. Interface com a tarefa exemplo de regra de ação que utiliza a ação *"Show result"*.

Na Figura 61 podemos observar o resultado final da execução do processo. Tal como referido, o conteúdo resultante da execução da chamada externa é exibido ao utilizador numa interface dedicada. Esta interface pode ser personalizada utilizando a componente *Templates Management* que permite ao utilizador tornar a interface mais apelativa e amigável ao utilizador final, neste caso utilizou-se a interface padrão da peça. No exemplo dado, a execução do processo termina após a ação de mostrar resultados, pois não foram definidas mais ações.

3.5.2.2 *Create entity(ies)*

A ação *create entity(ies)* foi desenvolvida com o objetivo de importar valores de SI externos para o DISME. Durante a fase inicial do desenvolvimento dessa ação, foi crucial compreender como o *Execution Engine* lidaria com esta ação no sistema.

Esta ação necessitou de uma atenção especial durante o seu desenvolvimento, uma vez que todos os valores no DISME estão associados a uma entidade interna ao sistema, através das suas propriedades. Armazenar os valores de SI externos na tabela *"values"*, que contém todos os valores internos do DISME, sem nenhuma entidade associada, afetaria a integridade e funcionamento do restante sistema.

Desde o início da implementação desta ação interna, ficou claro que seria necessário associar valores devolvidos pela realização da chamada a um tipo de entidade interno ao sistema. Por esse motivo, houve a necessidade de incorporar, na peça *"create entity(ies)"*, uma caixa de seleção que lista todas as entidades internas ativas ao DISME como observado na Figura 62. O objetivo é identificar a entidade na peça e associar todos os valores resultantes da chamada a essa entidade.

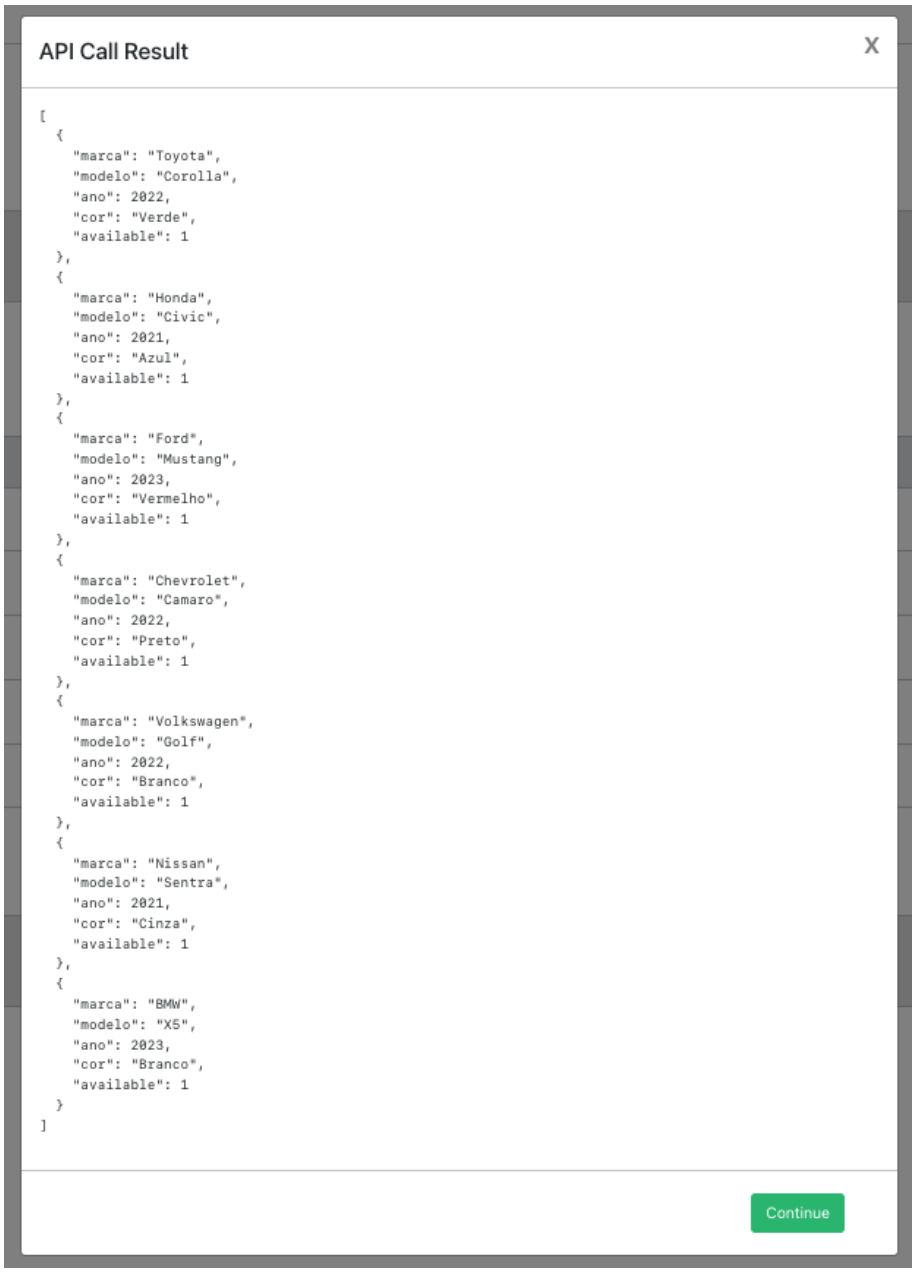

Figura 61. Interface resultante da ação *"Show result"*.

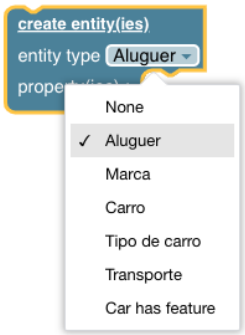

Figura 62. Caixa de seleção com a lista de entidades internas adicionada na peça *"create entity(ies)"*.

Posteriormente, esta abordagem inicial revelou-se incompleta, uma vez que houve a necessidade de identificar qual propriedade da entidade escolhida na caixa de seleção iria armazenar os valores retornados pela realização da chamada e quais valores retornados deveriam ser armazenados. Foi então que surgiu o desenvolvimento de uma nova peça, como observado na Figura 63, que encaixa na peça *"create entity(ies)"*, sob a forma de *property(ies)*. Esta nova peça chamada de *"matching"* serviu para garantir e indicar ao *Execution Engine* que existe uma correspondência entre as propriedades externas e as propriedades da entidade interna ao sistema. Isso permite garantir que apenas os valores desejados pelo utilizador de uma determinada propriedade interna ao DISME sejam armazenados.

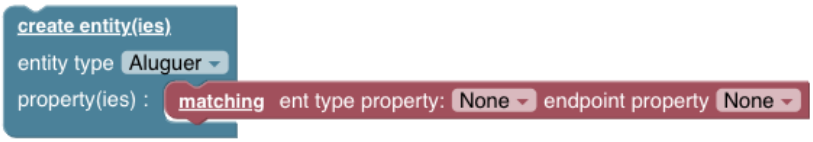

Figura 63. Peça *"matching"* que faz a correspondência entre propriedades externas e propriedades internas ao DISME.

Depois do desenvolvimento das duas peças passou-se ao desenho da ação que armazena dados externos ao DISME utilizando uma *external api call* a um SI. Utilizou-se o exemplo do processo da ação interna anterior, "Importar lista de carros do parceiro"para desenvolver uma ação que executa uma chamada externa ao DISME e armazena os dados internamente. A Figura 64 ilustra a utilização das peças *"create entity(ies)"* e *"matching"* integradas na peça *"action"* como ação interna resultante da execução da regra de ação.

No seguimento do tratamento das ações internas deste tipo, o *Execution Engine* utiliza a API de comunicação entre a base de dados e o *backend*, mencionada na secção anterior, para recolher todos os dados e informações necessárias da *external api call* a fim de realizar a chamada e tratar do seu resultado.

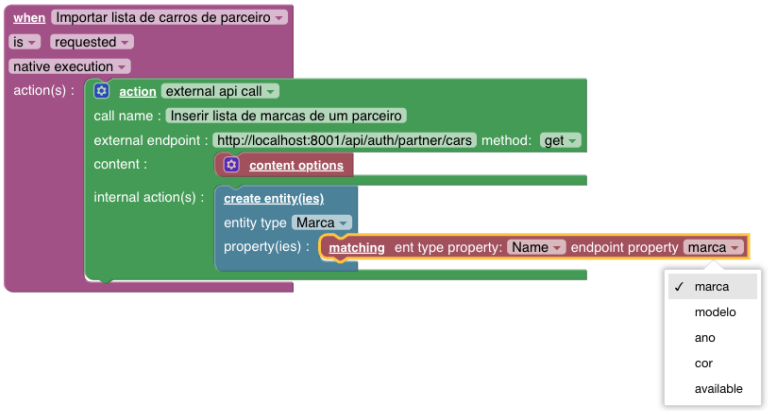

Figura 64. Exemplo de utilização das peças *"create entity(ies)"* e *"matching"* integradas na peça *"action"*.

Posteriormente à realização da chamada, executada pelo *API Request Service*, o *Execution Engine* recebe o resultado da chamada que poderá ser do formato *JSON* ou *XML*, e armazena em *cache* antes de prosseguir com o restante tratamento da ação. Ainda no decorrer da execução da ação, o sistema interpreta os dados inseridos pelo utilizador no momento em que a regra de ação foi desenhada. Esses valores indicam ao *Execution Engine*, no exemplo utilizado, que o mesmo deve armazenar valores da entidade "Marca"e fazer a correspondência entre a sua propriedade *"Name"* e a propriedade externa "marca".

Após o sistema ter a indicação de que correspondência deve ser feita e a estrutura resultante da chamada externa, o mesmo utiliza uma função criada para identificar a estrutura devolvida. Isso significa que, através da estrutura resultante, é possível determinar quantas entidades devem ser criadas. No exemplo utilizado na Figura 65, podemos observar a presença de sete entidades do tipo carro com cinco propriedades. Nesta situação a função identifica a necessidade de criar sete novas entidades no DISME, sendo que para cada uma delas, uma propriedade do tipo "marca"deve ser criada, conforme determinado no desenho da regra de ação.

Para executar a regra de ação, o utilizador deve aceder à *dashboard*, como podemos observar na Figura 66, onde inicia o processo "Importar Lista de carros do parceiro".

Em seguida, o sistema cria um novo processo na lista de tarefas pendentes ao utilizador, o qual é iniciado quando o utilizador clica na seta exibida na Figura 67.

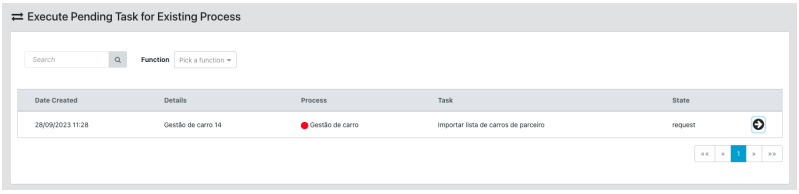

Figura 67. Interface com a tarefa exemplo de regra de ação que utiliza a ação *"Create entity(ies)"*.

Esta é uma ação que, ao ser executada, notifica o utilizador por meio de uma notificação *"push"* de sucesso. Na notificação, o utilizador recebe informações sobre a quantidade de entidades

```
[
  {
    "marca": "Toyota",
    "modelo": "Corolla",
    "ano": 2022,
    "cor": "Verde",
    "available": 1
  },
  {
    "marca": "Honda",
    "modelo": "Civic",
    "ano": 2021,
    "cor": "Azul",
    "available": 1
  },
  {
    "marca": "Ford",
    "modelo": "Mustang",
    "ano": 2023,
    "cor": "Vermelho",
    "available": 1
  },
  {
    "marca": "Chevrolet",
    "modelo": "Camaro",
    "ano": 2022,
    "cor": "Preto",
    "available": 1
  },
  {
    "marca": "Volkswagen",
    "modelo": "Golf",
    "ano": 2022,
    "cor": "Branco",
    "available": 1
  },
  {
    "marca": "Nissan",
    "modelo": "Sentra",
    "ano": 2021,
    "cor": "Cinza",
    "available": 1
  },
  {
    "marca": "BMW",
    "modelo": "X5",
    "ano": 2023,
    "cor": "Branco",
    "available": 1
  }
]
```


Figura 65. Exemplo de uma estrutura de dados com sete entidades.

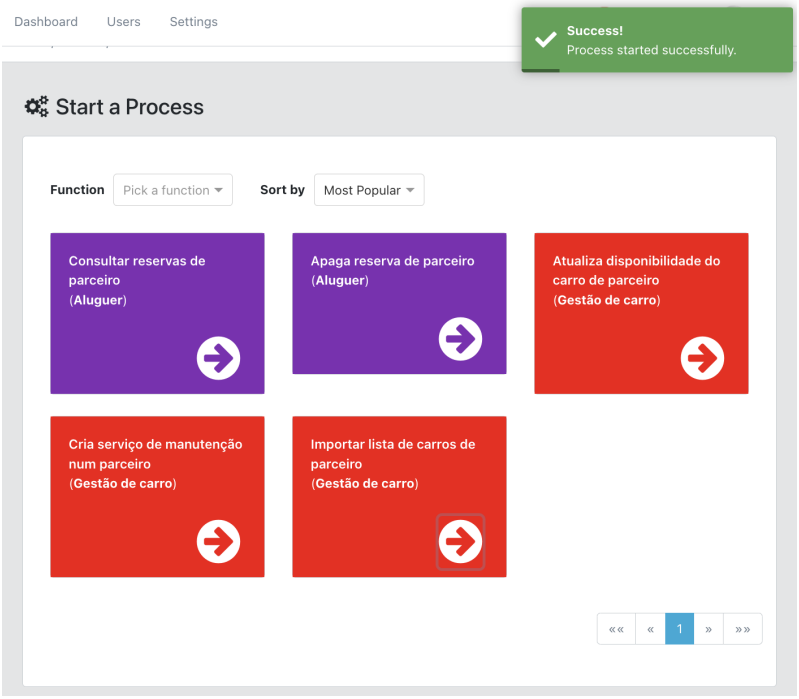

Figura 66. Interface da dashboard com o exemplo de regra de ação que utiliza a ação *"Create entity(ies)"*.

encontradas no SI externo e importadas para o DISME (Figura 68). A partir do momento em que a regra é finalizada os valores externos passam a existir no DISME e podem ser utilizados normalmente.

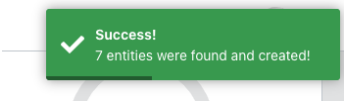

Figura 68. Notificação *push* de execução de uma regra de ação que utiliza a ação *"Create entity(ies)"*.

3.5.2.3 *Create temporary entity(ies)*

Esta ação foi desenvolvida com o objetivo de ser uma variante da ação anterior, *"create entity(ies)"*. O seu principal objetivo é realizar uma chamada externa a outro SI e guardar os dados para serem utilizados como *inputs* do utilizador no momento em que a regra de ação é executada.

O desenvolvimento da ação torna-se um grande desafio, pois pretende-se que os valores devolvidos pela chamada sejam armazenados por um curto espaço de tempo no DISME, apenas no contexto da execução da regra de ação. Numa primeira fase de desenvolvimento do tratamento desta ação considerou-se armazenar todos os valores externos como valores internos ao DISME, na tabela *"value"*. Esta primeira solução resultou, mas rapidamente percebeu-se que estavam a ser importados valores desnecessários e que estava a afetar a integridade dos dados no sistema.

Posto isto, foi necessário desenvolver uma alternativa ao armazenamento de dados no DISME que contornasse o problema da primeira solução. A alternativa proposta passou por duplicar a tabela *"value"*, mas num contexto diferente. O objetivo desta duplicação seria utilizar a tabela

duplicada apenas para valores temporários ao DISME, desta forma a integridade dos dados do restante sistema é salvaguardada.

Numa primeira instância de desenvolvimento da nova proposta, percebeu-se que, devido às relações na base de dados, outras tabelas deveriam ser duplicadas, essas tabelas são: *entity*, *ent_type*, *ent_type_name*, *property* e *property_name*. A duplicação destas tabelas permite assim criar entidades com propriedades com valores temporários. A nomenclatura utilizada para identificar as novas tabelas foi adicionar *"_temp"* às tabelas existentes, desta forma foi possível identificar as tabelas associadas ao armazenamento de valores temporários no sistema. Consequentemente, foram criadas as seguintes tabelas:

- ***entity_temp***
- ***ent_type_temp***
- ***ent_type_name_temp***
- ***property_temp***
- ***property_name_temp***
- ***value_temp***

Na Figura 69, observamos a utilização da peça de ação interna *"create temporary entity(ies)"* que, com a duplicação das tabelas, permite criar nas mesmas um conjunto de entidades e as suas respetivas propriedades, temporariamente, para serem utilizadas no decorrer da execução da regra de ação.

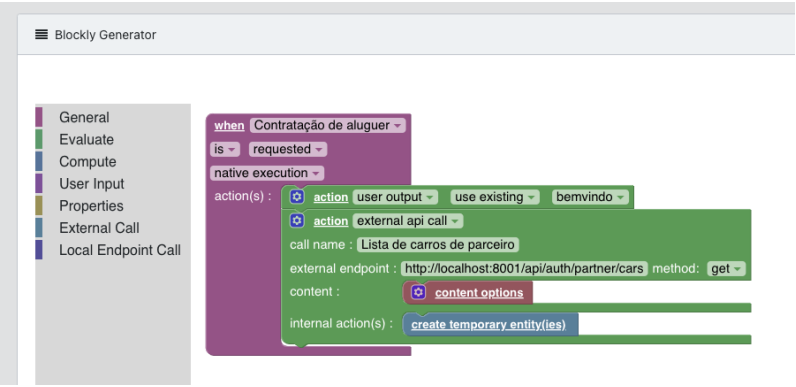

Figura 69. Exemplo de utilização de uma ação interna do tipo *create temporary entity(ies)*.

Pelo facto dos valores temporários serem utilizados no contexto de execução de uma regra de ação é necessário disponibilizar ao utilizador os mesmos para preenchimento auxiliar de formulários.

O processo natural de uma ação do tipo *"external api call"* que guarde valores temporários é a sequência da ação *"user input of a single entity type"*. Esta ação permite criar um formulário de preenchimento para o utilizador no contexto de execução de uma regra de ação. Por exemplo, numa regra de ação em que o utilizador deve inserir informações sobre uma contratação de aluguer de carros, e que seja necessário indicar a marca do carro que o utilizador deseja escolher, o sistema permite que o utilizador insira manualmente a marca ou, utilizando uma chamada com valores

temporários para obter uma lista de nomes de marcas de carros de um SI externo, disponibiliza ao utilizador uma caixa de seleção com uma lista de marcas de carro.

À semelhança da ação interna anterior, *"create entity(ies)"*, é necessário criar uma correspondência entre a propriedade externa e a propriedade temporária interna ao DISME (Figura 70). Desta forma, o *Execution Engine* ao executar a primeira ação, cria e armazena, nas novas tabelas, os valores e, na ação que se segue em que o utilizador deve preencher um formulário, existe uma correspondência utilizando a peça *"temporary_matching"* para indicar ao sistema em que propriedade devem ser utilizados os valores temporários.

No DISME, as ações do tipo *"user input of a single entity type"* requerem o desenho de um formulário que permite ao utilizador que está a desenhar a regra de ação solicitar *inputs* no contexto de execução da regra de ação. Esses *inputs* são definidos no momento em que a regra de ação é desenhada ao inserir a peça do tipo *"property"*. Nesse momento o utilizador pode escolher a propriedade que deseja incluir no formulário bem como a sua obrigatoriedade.

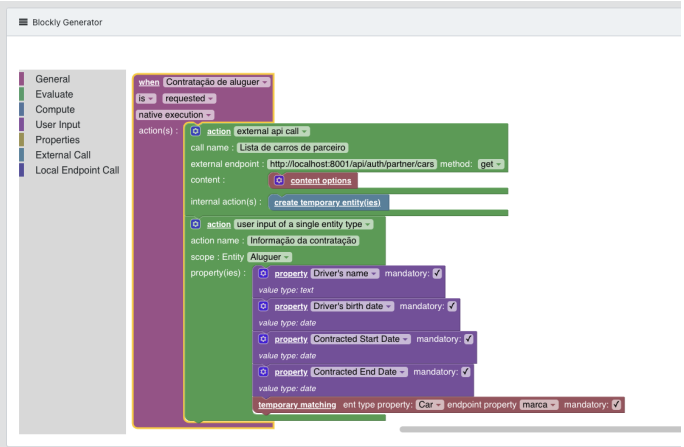

Figura 70. Exemplo de correspondência de propriedades numa ação interna do tipo *create temporary entity(ies)*.

Ao incluir valores temporários no sistema foi necessário alterar a componente responsável pela gestão dos formulários,*"Forms Management"* (Figura 71), nas funções de criar e armazenar o formulário.

Figura 71. Exemplo de criação de um formulário de inserção para o utilizador através da componente *"Forms Management"*.

Na Figura 72 podemos observar a criação de um formulário de inserção do utilizador para o exemplo de contratação de aluguer. Nesse formulário através de uma *interface* onde existe a possibilidade de arrastar os *inputs* é possível criar um formulário para ser utilizado no contexto de execução de uma regra de ação com as propriedades previamente definidas. O tipo de *input* utilizado está definido como atributo da propriedade da entidade. Ou seja, em propriedades do tipo data o *input* será do tipo data, em propriedades do tipo *text* o *input* será do tipo texto e em propriedades do tipo *number* o *input* será do tipo número.

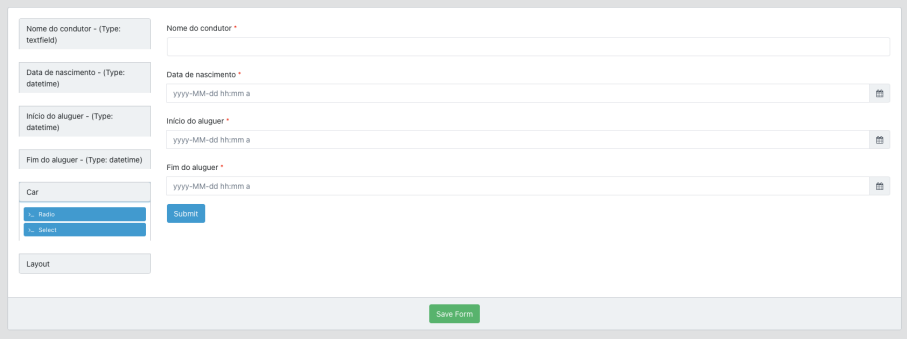

Figura 72. Exemplo da criação de um formulário de inserção para o utilizador.

A primeira alteração a ser realizada na criação do formulário é no momento em que o utilizador desenha o formulário. O DISME identifica, através da comunicação de APIs, a existência de ações do tipo *"external api call"* com a ação interna de criar valores temporários na regra de ação atual. Se existir, o sistema realiza uma chamada prévia e armazena os valores nas tabelas temporárias criadas para o efeito. Nesse momento, o sistema carrega na construção do formulário uma caixa de seleção (Figura 73) com os valores temporários na propriedade em que o *matching* foi definido no momento do desenho da regra de ação. Neste exemplo, utilizou-se a propriedade *"car"*.

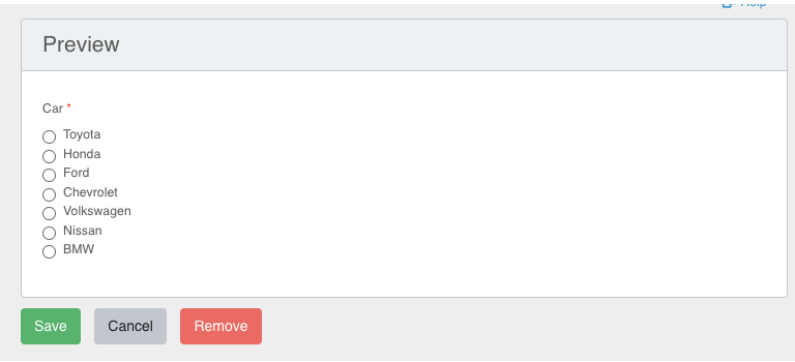

Figura 73. Lista de marcas de carro importadas utilizando valores temporários.

Na Figura 74 observamos o formulário finalizado com a existência de um *input* para cada propriedade. Após armazenar o formulário o desenho do processo "Contratação de aluguer"encontra-se finalizado e pronto a ser utilizado pelo sistema.

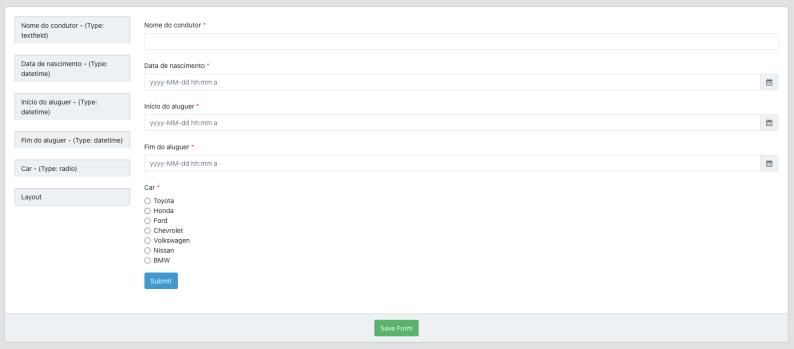

Figura 74. Exemplo de um formulário de inserção para o utilizador utilizando valores temporários e internos ao DISME.

Após desenhar e armazenar a regra de ação o utilizador deve aceder à *dashboard* onde irá iniciar o processo que corresponde à regra de ação desenhada. Na Figura 75 podemos observar o botão em forma de seta que inicia o processo "Contratação de aluguer". Ao pressionar o botão o *Execution Engine* inicia o tratamento do fluxo de informações e processos inerentes à especificação da regra de ação, inserindo a tarefa na lista de tarefas pendentes ao utilizador (Figura 76).

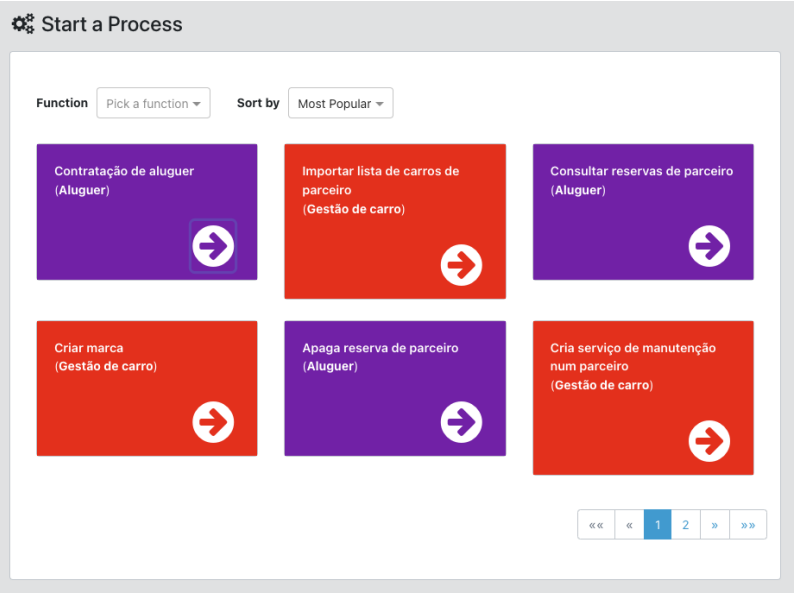

Figura 75. Interface da *dashboard* com o exemplo de regra de ação que utiliza a ação *"Create temporary entity(ies)"*.

Ao iniciar o processo, o *Execution Engine* executa cada ação individualmente de acordo com a especificação da regra de ação. No exemplo utilizado, o sistema efetua sempre a chamada externa ao SI e guarda os valores retornados nas novas tabelas. Posteriormente, o formulário carrega esses valores na propriedade escolhida pelo utilizador para a correspondência em forma de lista (Figura 77).

Apesar do formulário ser desenhado uma única vez, os valores temporários utilizados para o desenho servem apenas de referência ao utilizador para o desenho do formulário. Numa situação

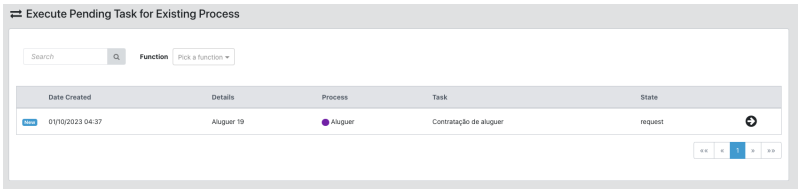

Figura 76. Interface com a tarefa exemplo de regra de ação que utiliza a ação *"Create temporary entity(ies)"*.

em que os valores do SI externo sejam alterados, a regra de ação não necessita de ser atualizada pois o DISME, ao iniciar o processo inerente à regra de ação, realiza sempre uma chamada e, consequentemente, armazena os valores temporariamente para serem apresentados ao utilizador.

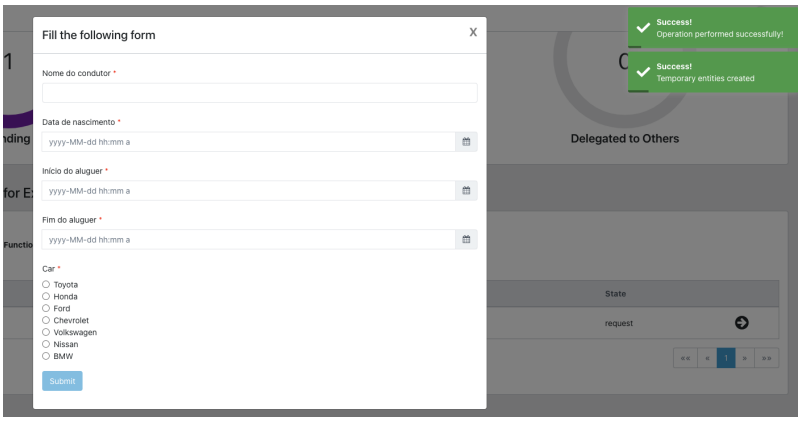

Figura 77. Execução da ação *user input of a single entity type* utilizando valores temporários.

A finalização do processo, com o preenchimento dos dados por parte do utilizador, foi alterada. Após os dados serem inseridos pelo utilizador, o DISME realiza uma verificação para a existência de valores temporários selecionados pelo utilizador e realiza uma cópia dos mesmos para a tabela *"values"*. Deste modo, os valores selecionados passam a ser valores internos ao DISME e os restantes valores temporários são eliminados do sistema.

3.6 Nova componente API Management

Após a compreensão da arquitetura da plataforma DISME e de todas as suas componentes de utilização, conforme apresentado na Figura 78, iniciou-se o desenvolvimento de uma nova componente destinada a gerir todas as REST APIs utilizadas pelo sistema, denominada de *"API Management"* (Figura 79).

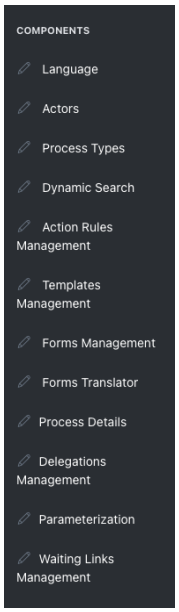

Figura 78. Todas as componentes disponíveis no DISME.

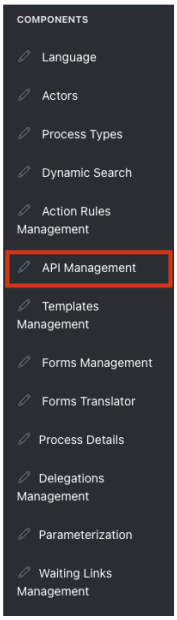

Figura 79. Adição da componente nova "API Management"no DISME.

O principal objetivo desta nova componente foi disponibilizar ao utilizador as *CRUD operations* como criar, ler, editar e eliminar, em forma de lista, permitindo assim a consulta de todas as REST APIs ativas no sistema. Na Figura 80, é possível observar a nova interface da componente, que inclui exemplos REST APIs, que possibilitam a interoperabilidade da plataforma DISME com plataformas externas.

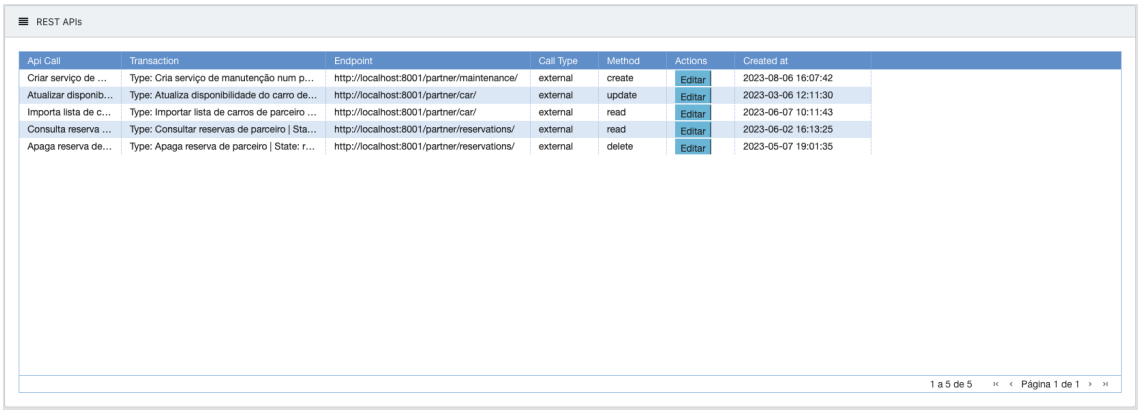

Figura 80. Interface do utilizador na componente API Management.

Numa fase inicial do desenvolvimento da nova componente, foram implementadas as quatro operações, cada uma com um botão correspondente:

1. **"Criar REST API"**: Utilizado para criar novas chamadas a REST APIs;
2. **"Exibir"**: Permite visualizar a configuração de uma chamada a uma REST API;
3. **"Editar"**: Utilizado para editar uma chamada REST API previamente definida;
4. **"Eliminar"**: Permite eliminar permanentemente uma REST API;

O botão "Criar REST API"foi inserido no topo da página da componente, e as outras opções foram inseridas na coluna "*Actions*"permitindo manipular cada REST API individualmente.

No entanto notou-se que a nova componente funcionaria melhor apenas com o botão "Editar", visto que as outras opções estão integradas em outras componentes existentes no DISME, as quais serão mencionadas na secção seguinte.

3.6.1 Integração com o restante sistema

A abordagem presente nesta secção tem como objetivo otimizar a usabilidade global do sistema, permitindo que os utilizadores possam realizar as suas tarefas essenciais concentrados, sem serem sobrecarregados por tarefas administrativas.

A componente *API Management* é integrada com o restante sistema por meio de uma das principais componentes do DISME, *Action Rules Management*. Esta integração tem um impacto direto e positivo na gestão das REST APIs, simplificando significativamente a experiência do utilizador. A existência da nova componente está ligada ao *design* e à construção de regras de ação na componente *Action Rules Management* que utiliza o Blockly como ferramenta de programação visual, "*drag-and-drop*". O ambiente proporcionado pela ferramenta é flexível e permite configurar diversos campos e inputs que facilitam a interação entre os blocos e o utilizador.

Quando um utilizador cria uma regra de ação no sistema para executar uma chamada externa a outro SI, a componente *API Management* entra em ação de forma transparente. Automaticamente, uma REST API responsável por realizar essa chamada é criada e incorporada à lista de REST APIs da componente *API Management*. Isso não só simplifica o processo, mas também elimina a necessidade de existência do botão "Criar REST API"que, anteriormente, exigia uma ação manual por parte do utilizador.

No sentido de simplificar ainda mais a experiência do utilizador, decidiu-se remover o botão "Exibir,"mantendo apenas o botão "Editar."No contexto do DISME, as ações de "Editar"e "Exibir"realizam essencialmente a mesma tarefa. Quando o utilizador clica no botão "Editar"de uma REST API na lista da *API Management*, é redirecionado para a componente *Action Rules Management*. Nesse momento, o sistema carrega a regra de ação por completo e exibe-a ao utilizador. O mesmo pode optar por visualizar a regra de ação ou, se necessário, editá-la. Essa simplificação e consistência na experiência do utilizador garantem que a navegação seja intuitiva e que o utilizador possa realizar as suas tarefas de forma eficaz.

3.6.2 Armazenamento da componente

A componente *System Executor* desempenha um papel fundamental no DISME, sendo responsável pela execução de todas as regras de ação criadas pelos utilizadores na componente *Action Rules Management*. Essa execução só é possível graças à integração com APIs internas que facilitam a comunicação entre o *System Executor* e a base de dados que armazena as regras de ação criadas pelos utilizadores no *Diagram Editor*.

Para garantir uma execução precisa das regras de ação, o *System Executor* utiliza uma abordagem baseada numa estrutura XML. Essa estrutura é obtida pela combinação das peças da ferramenta Blockly no momento em que o utilizador cria a regra de ação. A estrutura XML representa a lógica das ações a serem executadas e serve como referência para o *System Executor*.

A implementação da nova componente no DISME, *API Management*, segue um princípio semelhante ao utilizado no armazenamento de regras de ação. Isto garante uma consistência eficaz entre a lógica da regra de ação definida pelo utilizador e a forma como é interpretada e executada pelo sistema. A utilização da estrutura XML resultante da ferramenta Blockly permite a tradução dessa lógica para a ação real.

O desenvolvimento da nova componente começou pela alteração da ação de guardar a regra de ação. Para isso, foi necessário desenvolver uma função que fosse capaz de identificar todos os blocos do tipo *"external call api"* individualmente e analisar a estrutura XML da regra de ação por completo, separando cada bloco do tipo ação com a opção *"external call api"*. Isso permitiu a criação de uma lista de todas as ações *"external call api"* utilizadas nas regras de ação com o auxílio da nova componente.

Durante a análise da estrutura para as novas tabelas da base de dados responsáveis por armazenar o conteúdo das ações do tipo *"external call api"*, ficou claro que era essencial preservar as informações fundamentais para construir uma chamada externa, como parâmetros e conteúdo JSON. Esses dados são inseridos nas tabelas referentes às peças mencionadas anteriormente, *"parameters"* e *"json content"*, respetivamente.

O conteúdo inserido nessas tabelas desempenha um papel crucial na interpretação e execução eficaz da componente *"System Executor"*, auxiliando na compreensão das ações e na realização das chamadas externas de maneira precisa e confiável.

3.6.2.1 Novas tabelas

Após a análise, decidiu-se criar duas novas tabelas na plataforma responsáveis por todo o armazenamento das chamadas REST API externas. Essas tabelas são:

1. **api_call_type**

   A tabela presente na Figura 81 é responsável por armazenar os dados das chamadas externas realizadas pelo DISME a outros SI. A Tabela 3 apresenta os atributos da nova tabela (api_call_type).

Tabela 3. Tabela de atributos da tabela api_call_type.

Atributo	Descrição
action_rule_id	Chave estrangeira para a tabela action_rule que permite estabelecer uma relação entre a chamada e uma regra de ação.
name	Do tipo *varchar*, armazena o nome indicado pelo utilizador para identificar a chamada no sistema.
description	Do tipo *varchar*, armazena a descrição da chamada.
endpoint	Do tipo *varchar*, armazena o *endpoint* utilizado na realização da chamada.
type	Do tipo *enum*, esta propriedade guarda os tipos de chamada existentes no DISME, sendo elas *internal* e *external*.
method	Do tipo *enum*, esta propriedade guarda os métodos existentes numa chamada. Os métodos são *GET*, *POST*, *PUT* e *DELETE*.
json_content	Do tipo *longtext*, guarda o conteúdo em formato JSON.
blockly_xml	Do tipo *longtext*, guarda o formato da peça Blockly em XML.
blockly_code	Do tipo *longtext*, guarda o formato da peça em código com o formato de interpretação pelo *System Executor*.
update_by	Chave estrangeira para a tabela utilizadores, permite referenciar o utilizador que editou pela última vez a chamada.
deleted_by	Chave estrangeira para a tabela utilizadores, permite referenciar o utilizador que eliminou a chamada.
created_at	Do tipo timestamp, guarda o dia e a hora em que a chamada foi criada.
updated_at	Do tipo timestamp, guarda o dia e a hora em que a chamada foi editada.
deleted_at	Do tipo timestamp, guarda o dia e a hora em que a chamada foi eliminada.

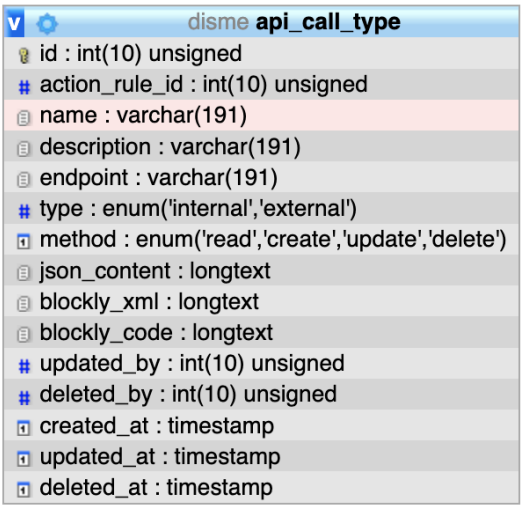

Figura 81. Nova tabela "api_call_type"adicionada na base de dados.

2. **api_call_type_has_parameter**

A nova tabela adicionada no sistema designada de "api_call_type_has_parameter"presente na Figura 82 permite ao utilizador complementar a informação da REST API com a adição de um ou vários parâmetros no URL na realização da chamada. A Tabela 4 apresenta os atributos da nova tabela (api_call_type_has_parameter).

Tabela 4. Tabela de atributos da tabela api_call_type_has_parameter.

Atributo	Descrição
api_call_type_id	Chave estrangeira para a tabela api_call_type que permite estabelecer uma relação entre um parâmetro e REST API.
key	Do tipo *varchar*, armazena a *key* do parâmetro na construção da REST API.
value	Do tipo *varchar*, armazena o *value* do parâmetro na construção da REST API.
created_at	Do tipo timestamp, guarda o dia e a hora em que o parâmetro foi criado.
updated_at	Do tipo timestamp, guarda o dia e a hora em que o parâmetro foi criado.
deleted_at	Do tipo timestamp, guarda o dia e a hora em que o parâmetro foi criado.

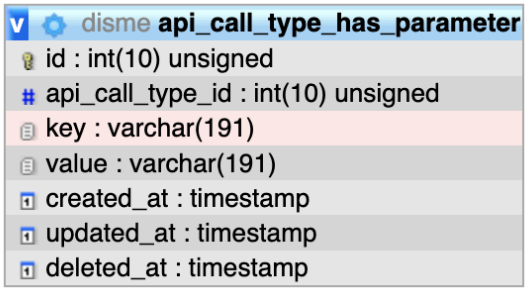

Figura 82. Nova tabela "api_call_type_has_parameter"adicionada na base de dados.

Após definir e criar as tabelas, houve a necessidade de desenvolver na componente do *backend* um recurso de armazenamento para comunicar com a componente *Action Rules Management*. O recurso utilizado denomina-se "*api resource*"e é uma prática comum na implementação de *backend* utilizando a *framework laravel*. Desta forma, ao guardar uma regra de ação que contenha uma chamada externa, o sistema, através da comunicação com o *backend*, armazena a informação nas tabelas recém criadas.

4 Validação

O presente capítulo, representa um marco importante no desenvolvimento da presente dissertação. Após a implementação e desenvolvimento de conceitos e ferramentas, descritos nas secções anteriores, é imperativo submeter esses elementos a testes mais rigorosos e esclarecedores.

A validação desempenha um papel fundamental na confirmação da eficácia, utilidade e relevância das soluções apresentadas. Para atingir esse objetivo, este capítulo utiliza o caso de estudo amplamente reconhecido na área, denominado EU-Rent. Este caso de estudo serve como ferramenta de avaliação essencial, proporcionando um ambiente desafiador e representativo para avaliar a implementação realizada.

Ao longo deste capítulo, iremos explorar a utilização de cenários reais do caso EU-Rent, utilizando as novas ferramentas implementadas. Estes cenários serviram como critério para avaliar a eficácia das soluções desenvolvidas, permitindo medir o desempenho, a precisão e a capacidade de adaptação dos processos construídos. Procura-se não apenas verificar se as implementações cumprem as especificações técnicas, mas também avaliar a sua aplicabilidade.

4.1 Processos e métodos de validação

Nesta secção, são apresentados os processos e métodos de validação escolhidos para medir a eficácia da solução proposta, usando o caso de estudo EU-Rent como um ambiente de testes desafiador e representativo. O caso de estudo EU-Rent fornece uma oportunidade única para testar as implementações em condições de utilização real e analise de vários cenários que ocorrem em SI.

Cada um dos cenários será apresentado em detalhe, explicando o contexto em que ocorre e fornecendo informações sobre a sua implementação no DISME. Com esta abordagem pretende-se medir a eficácia da solução proposta em lidar com os desafios e requisitos específicos de cada cenário, bem como avaliar a sua aplicabilidade em situações reais.

4.2 Cenários de validação

Os cenários selecionados para validação, permitem testar todas as funções implementadas na nova componente de interoperabilidade do DISME, nomeadamente os quatro métodos: *GET, POST, PUT e DELETE*, para realizar chamadas externas, bem como as ações internas no sistema: *Show result, Create entity(ies), Create temporary entity(ies).*

Nas secções seguintes, iremos detalhar a construção e execução de regras de ação para cenários de validação no caso de estudo EU-Rent. Para uma melhor interpretação, consideraremos o DISME como sendo um EU-Rent. Além disso, serão utilizados *endpoints* fictícios que permitam a simulação de todos os casos.

4.2.1 Criar relatório de vendas

O cenário de interoperabilidade que utilizaremos envolve a ação de criar um relatório de vendas num SI externo. Ou seja, através da realização de uma chamada de API a um SI externo à EU-Rent, iremos criar um relatório de vendas na mesma.

Imaginemos que EU-Rent tem uma parceria com uma *rent-a-car* (RAC) e é necessário gerar um relatório de vendas no SI da RAC semanalmente. Este relatório permite que a EU-Rent receba, via *e-mail*, um ficheiro com todas as vendas de aluguer associadas à parceria.

Neste caso, é possível criar uma regra de ação na EU-Rent que permita aos utilizadores realizar uma chamada de API externa para esse efeito. Para executar esta tarefa, utilizando a nova componente, devemos aceder à componente *Action Rules Management* (Figura 81) e desenhar a regra de ação que permita alcançar o resultado desejado.

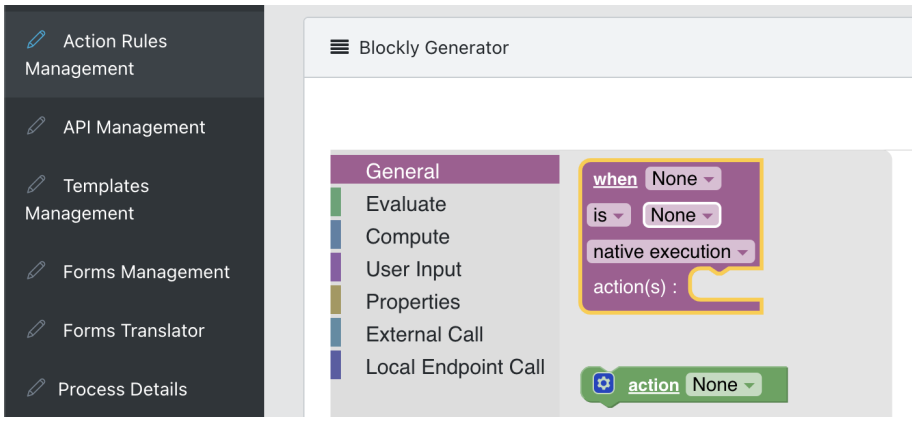

Figura 83. Utilização da peça *"when is do"* na criação de uma regra de ação para o exemplo "Criar relatório de vendas".

Para isso, começaremos por utilizar a peça *"when is do"*, disponível no separador *"General"*, e selecionar o processo "Criar relatório de vendas"no estado *is requested* (Figura 84). Isto permitirá ao sistema identificar quando a ação "Criar relatório de vendas"for iniciada as ações definidas na regra devem ser executadas.

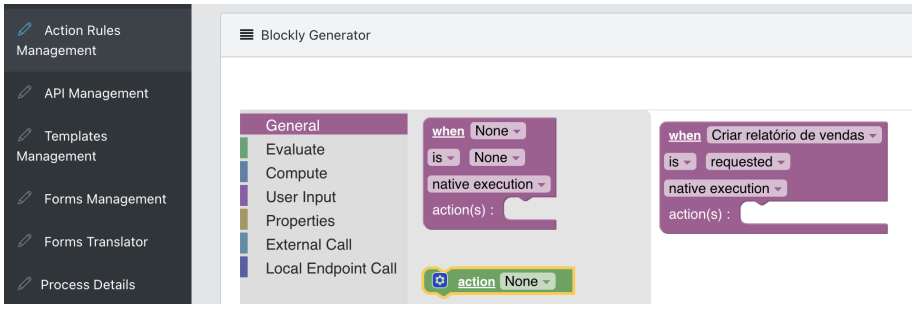

Figura 84. Utilização da peça *"action"* para o exemplo "Criar relatório de vendas".

Para configurar uma ação, é necessário aceder ao separador *"General"* e arrastar a peça *"action"* para a peça *"when is do"* no encaixe *"action(s)"*. No exemplo atual, como podemos observar na Figura 85, definimos o campo *"call name"* como "Chamada *POST* criar relatórios de vendas"para identificar a ação na EU-Rent. Posteriormente, devemos inserido o *external endpoint*, que deve ser indicado, num cenário real, pelo SI externo, através da sua documentação, e selecionar o método da chamada *"POST"*.

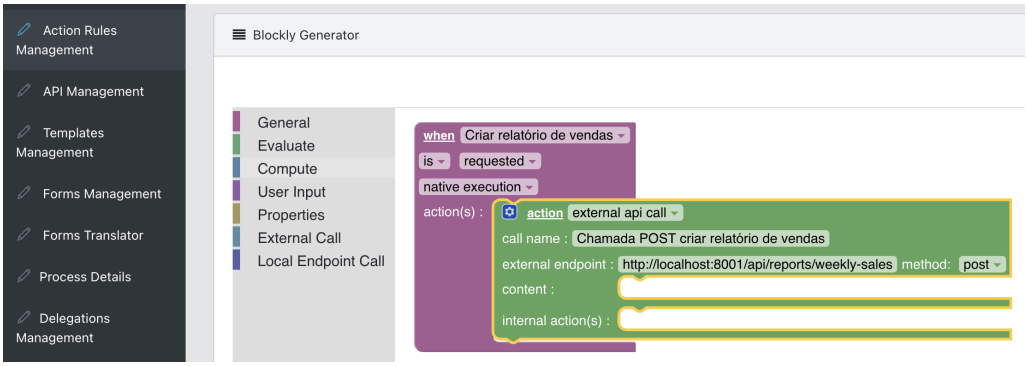

Figura 85. Configuração da peça *action* para o exemplo "Criar relatório de vendas".

Após a configuração base da peça *"action"*, é necessário complementá-la através da adição de duas peças. A peça *"content options"* (Figura 86), disponível no separador *"External Call"*, e a peça *"Show result"* (Figura 87) disponível no mesmo separador.

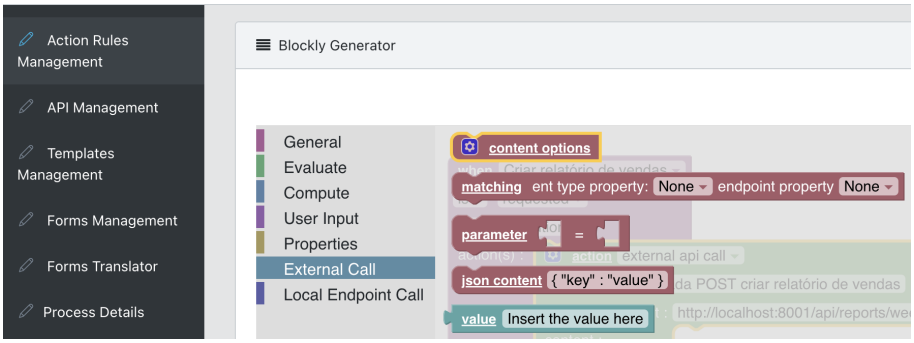

Figura 86. Utilização da peça *content options* no exemplo "Criar relatório de vendas".

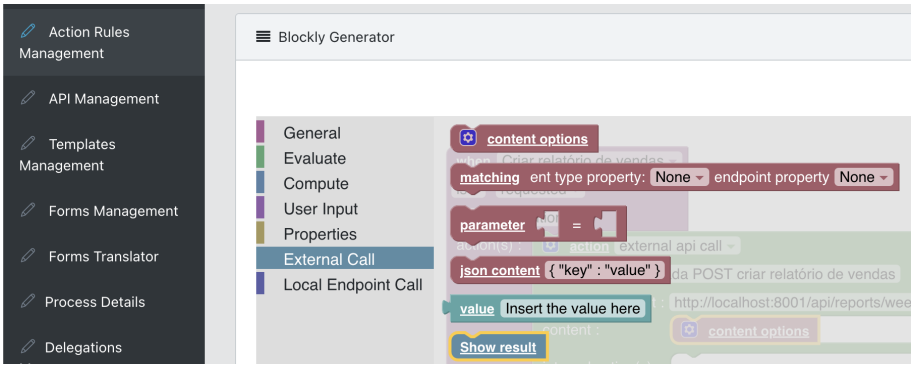

Figura 87. Utilização da peça *Show result* no exemplo "Criar relatório de vendas".

Na Figura 88, podemos observar o resultado do desenho da regra de ação. Após a sua conclusão, a mesma deve ser armazenada utilizando o botão *"Save Action Rule In Database"*. A partir desse momento, termina o desenho da regra de ação e a mesma pode ser executada no sistema.

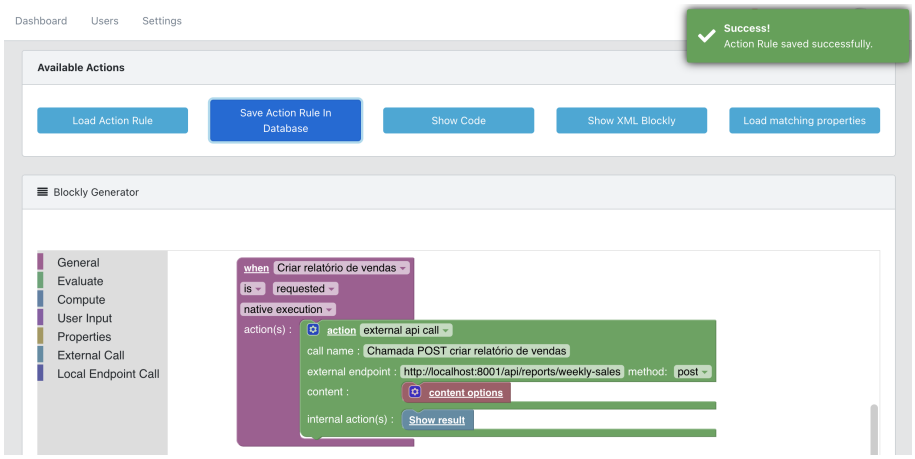

Figura 88. Armazenamento da regra de ação "Criar relatório de vendas".

Para executar a regra de ação, voltamos à *dashboard* e iniciamos o processo, como observado na Figura 89. Para isso, devemos clicar no botão e aguardar que o sistema notifique o utilizador de que o processo foi iniciado e colocado como tarefa pendente (Figura 90).

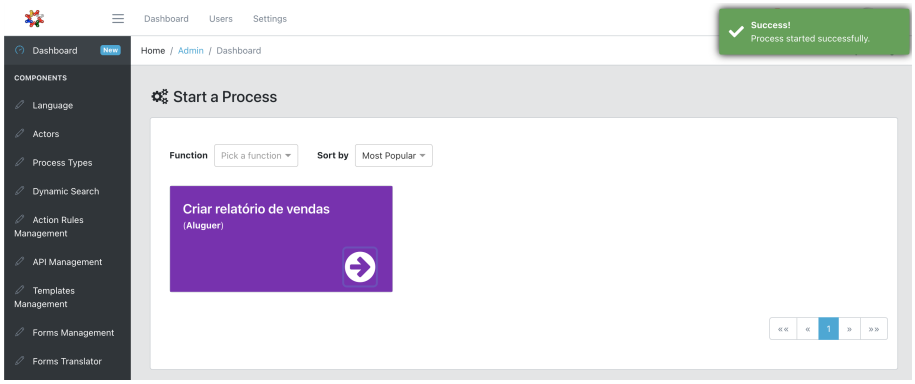

Figura 89. Início do processo de execução de criar relatório de vendas.

Na Figura 91, podemos observar o resultado final da execução do processo. O conteúdo resultante da execução da regra de ação "Criar relatório de vendas"é exibido ao utilizador numa interface dedicada. No exemplo definido, a execução do processo termina, pois não foram definidas outras ações.

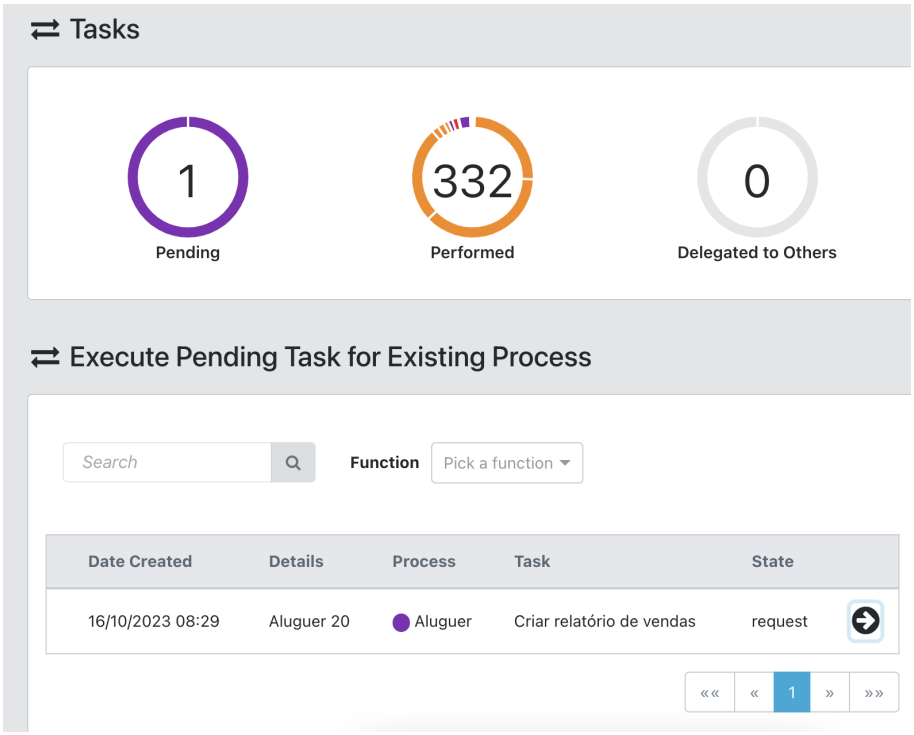

Figura 90. Execução do processo de criar relatório de vendas.

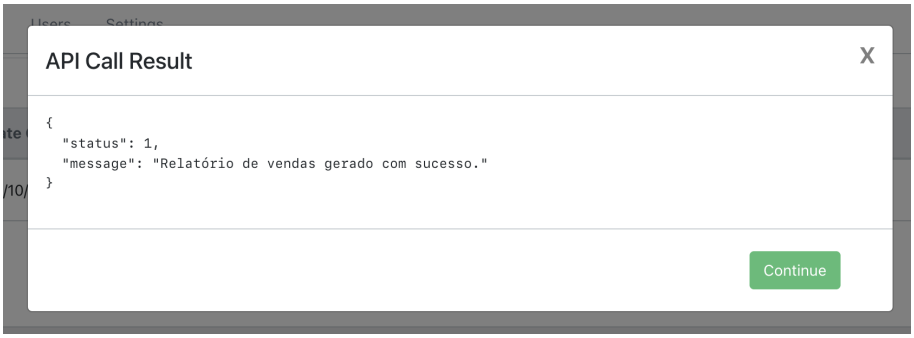

Figura 91. Resultado da execução do exemplo criar relatório de vendas.

4.2.2 Atualizar lista de carros no parceiro

Neste cenário de interoperabilidade, abordaremos a ação de atualizar a lista de carros num SI externo, através da realização de uma chamada de API a um SI externo à EU-Rent. O objetivo é atualizar a lista de carros disponíveis da EU-Rent em outra RAC parceira. Suponhamos que a EU-Rent mantém uma parceria com uma RAC e adquire uma nova gama de carros para a sua frota, é crucial garantir que a lista de carros disponíveis no sistema da RAC parceira esteja em conformidade com a lista de carros que a EU-Rent tem disponível para alugar. Esta atualização ocorre na sequência da transação "Adicionar carro", quando a mesma for aceite o sistema notificará a RAC parceira para a atualização da lista de carros.

Para realizar esta tarefa, utilizando a nova componente, devemos aceder à componente *Action Rules Management* e configurar uma regra de ação que permita realizar uma *external api call* para atualizar a lista de carros no parceiro (Figura 92).

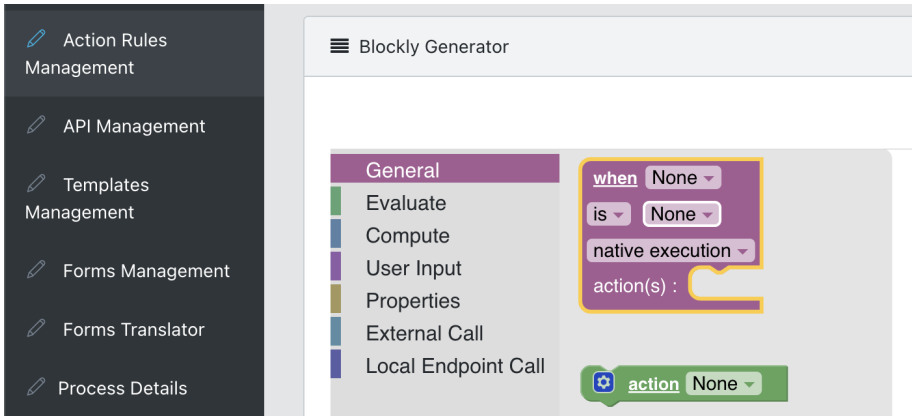

Figura 92. Utilização da componente *"Action Rules Management"* para a criação de uma regra de ação para o exemplo "Atualizar lista de carros no parceiro".

Começamos por utilizar a peça *"when is do"*, disponível no separador *"General"*, e selecionar o processo "Adicionar carro"no estado *"is accepted"* (Figura 93).

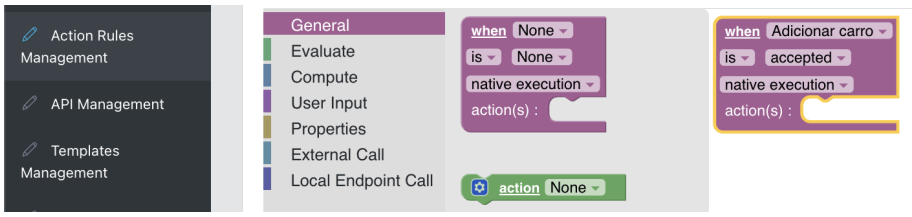

Figura 93. Utilização da peça *"action"* para o exemplo "Atualizar lista de carros no parceiro".

De seguida, configuraremos a peça *action* que deve ser arrastada para a peça *"when is do"*, o campo *"call name"* deve ser definido, por exemplo, como "Notificação de atualização lista de carros". Em seguida, é importante inserir o *endpoint* externo, que será fornecido pelo SI externo, e selecionar o método da chamada, neste caso, "*POST*"(Figura 94).

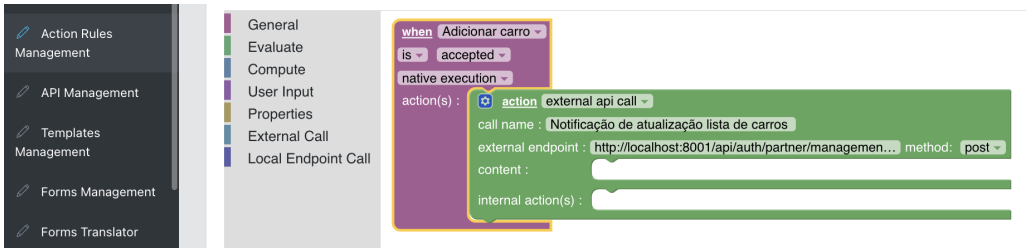

Figura 94. Configuração da peça *action* para o exemplo "Atualizar lista de carros no parceiro".

Para executar esta chamada é necessário enviar, através dos parâmetros do pedido, o *ID* de parceiro. Para esta configuração é necessário a adição da peça *"content options"* (Figura 95). A mesma deve ser alterada de forma a incluir o encaixe dos *parameter(s)* e, em seguida, a informação deve ser inserida utilizando as peças *"parameter"* e *"value"* (Figura 96).

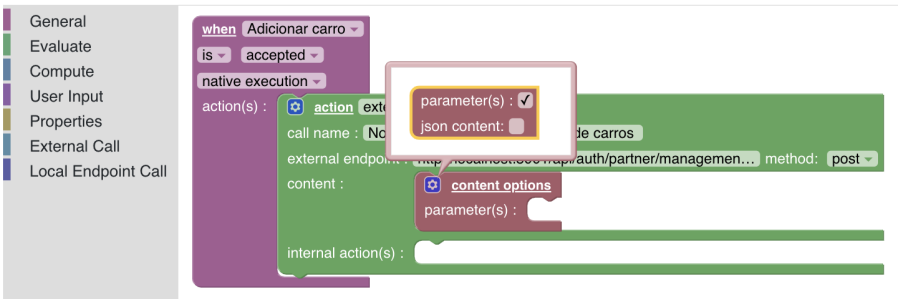

Figura 95. Utilização da peça *"content options"* com *parameter(s)* para o exemplo "Atualizar lista de carros no parceiro".

Para finalizar o desenho da regra de ação colocamos a peça *"Show result"* no encaixe das *"internal action(s)"*. Esta configuração possibilita mostrar ao utilizador o resultado da execução da regra de ação. Uma vez configurada a regra de ação, esta deve ser armazenada utilizando o botão *"Save Action Rule In Database"* (Figura 96).

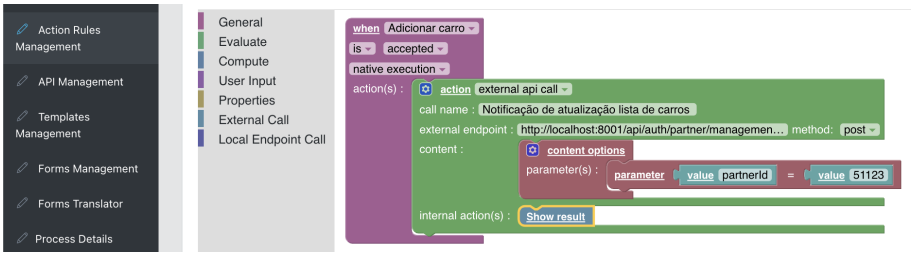

Figura 96. Regra de ação "Atualizar lista de carros no parceiro".

Para executar a regra de ação, o utilizador deve aceder à *dashboard* e iniciar o processo. Isto pode ser feito ao clicar no botão presente na Figura 97 e aguardando que o sistema notifique o utilizador que o processo foi iniciado e colocado nas tarefas pendentes (Figura 98).

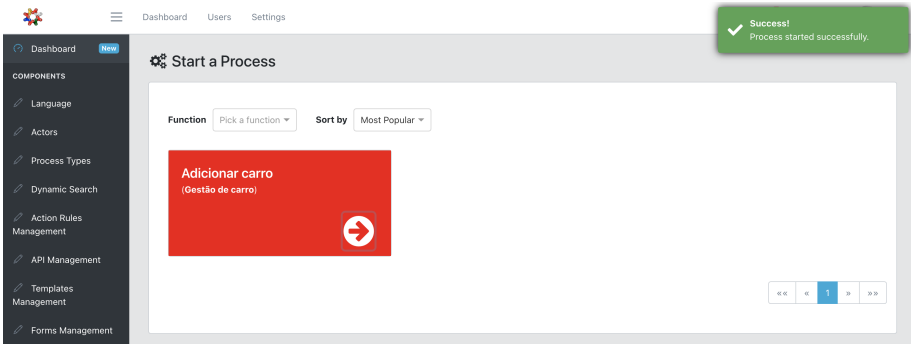

Figura 97. Início do processo de execução de atualizar lista de carros no parceiro.

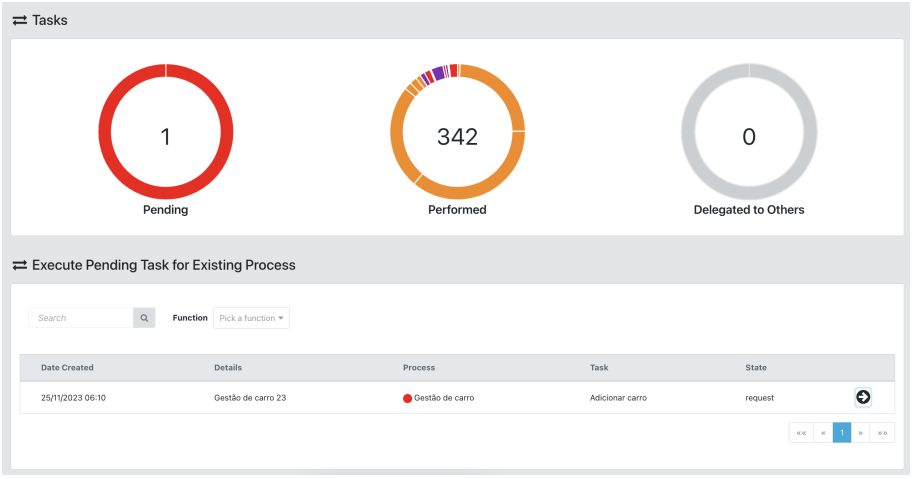

Figura 98. Tarefa criada para atualizar lista de carros no parceiro .

Ao iniciar o processo de execução da regra de ação "Adicionar carro"o sistema abre um formulário para que o utilizador insira os dados do carro que pretende adicionar à RAC. Após os dados serem inseridos o utilizador submete-os clicando no botão para o efeito (Figura 99).

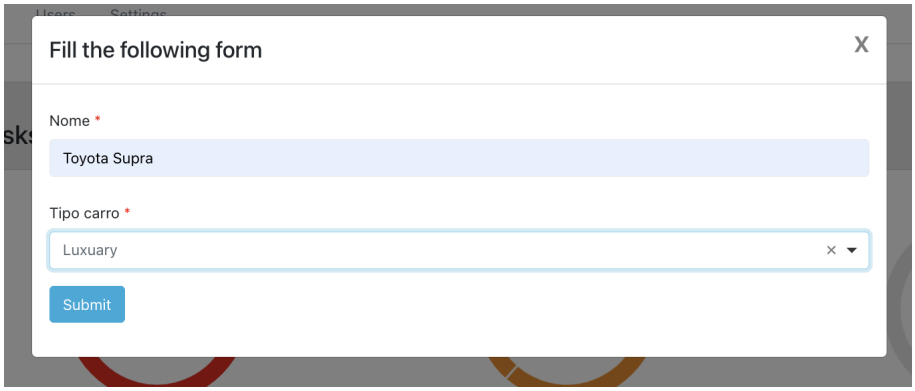

Figura 99. Formulário de inserção de carro no exemplo atualizar lista de carros no parceiro.

O sistema, ao armazenar os dados do novo carro concluí o primeiro estado da transação *requested* e passa para o novo estado *accepted*. O resultado da execução da regra de ação "Adicionar carro"no estado *accepted*, como definido no desenho da mesma, é uma interface dedicada que demonstra o sucesso do envio da notificação ao parceiro (Figura 100).

4.2.3 Cancelar reserva no parceiro

O presente cenário de interoperabilidade aborda a ação de cancelar reserva no parceiro num SI externo. A implementação deste cenário é exequível a partir de uma chamada a uma API externa à EU-Rent.

Considerando que a EU-Rent e outra RAC mantêm uma parceria estratégica que permite aos clientes da EU-Rent efetuarem reservas em veículos disponíveis na RAC parceira. Imaginemos que um cliente, após ter realizado uma reserva de um veículo da RAC parceira, decide alterar as datas do aluguer. Nesse sentido, existe todo um processo de alteração de reserva que passa pelo

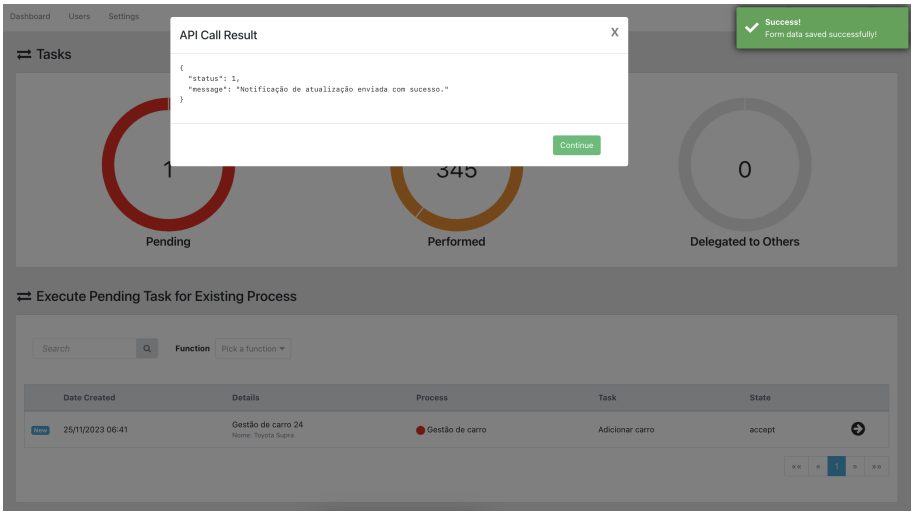

Figura 100. Resultado da execução do exemplo atualizar lista de carros no parceiro.

cancelamento da reserva antiga e pela criação de uma nova reserva com as novas datas. Neste cenário, é essencial que uma das etapas do termo do aluguer seja a comunicação com o SI parceiro para o cancelamento da reserva. Para atingir esse objetivo, é necessário criar uma regra de ação que realize uma chamada externa a uma API da RAC no momento.

Com esse propósito, para iniciar a construção da regra de ação, devemos aceder à componente *Action Rules Management*. Como mencionado anteriormente, esta componente permite desenhar e configurar uma regra de ação para realizar uma *external api call*.

Na Figura 101, observamos a primeira peça que deve ser colocada na área de trabalho, *"when is do"*, disponível no separador *"General"*. O processo a ser selecionado é o "Cancelar reserva no parceiro", esta ação acontece no seguimento de um pedido de cancelamento de reserva por isso deve ser selecionado o estado *"revoke request requested"*.

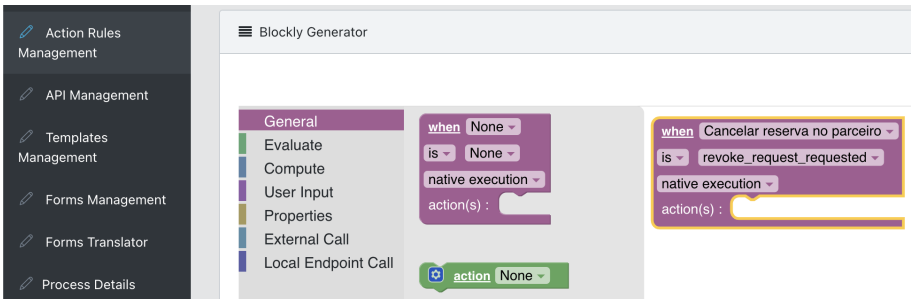

Figura 101. Utilização da componente *"Action Rules Management"* para a criação de uma regra de ação para o exemplo "Cancelar reserva no parceiro".

A segunda peça a ser utilizada é a peça *"action"* presente no separador *"General"* (Figura 102). Esta peça conecta-se à peça *"when is do"* e, na sua configuração de *"external api call"*, permite executar uma chamada externa à EU-Rent. Devemos definir o campo *"call name"* como "Cancelar reserva no parceiro"para identificar o nome da chamada. De seguida, inserimos o *endpoint* externo, fornecido pelo SI externo, e selecionamos o método da chamada como *"POST"*.

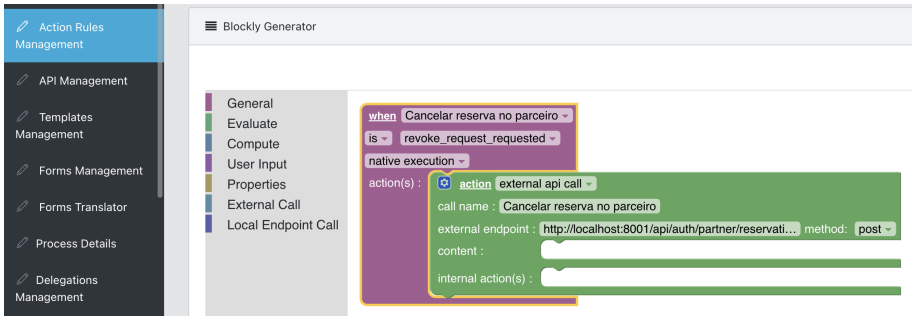

Figura 102. Utilização da peça *"action"* para o exemplo "Cancelar reserva no parceiro".

O *endpoint* utilizado neste cenário requer que seja enviado, juntamente com o pedido, o número da reserva como parâmetro. Para ajustar a peça *"content options"* com o propósito de adicionar parâmetros é necessário ativar a opção *"parameter(s)"* e, posteriormente, inserir *{"order":"RC-71253"}* na peça como observado na Figura 103.

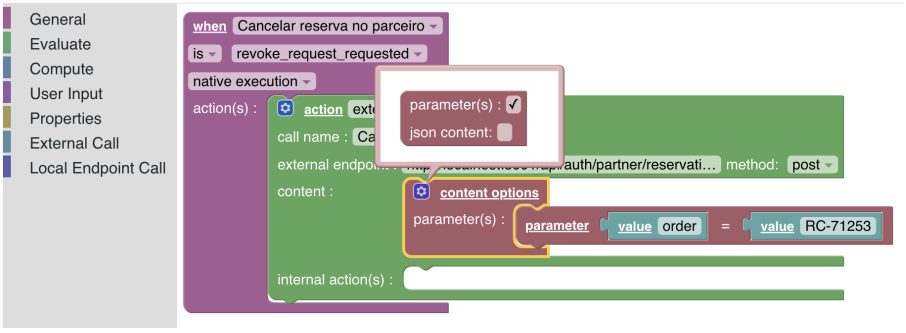

Figura 103. Peça *"content options"* ajustada com *"parameter(s)"* para o exemplo "Cancelar reserva no parceiro".

Para finalizar e armazenar o desenho da regra de ação, é necessário recorrer ao botão *"Save Action Rule In Database"*, que, em caso de sucesso, notifica o utilizador através de uma notificação do sistema (Figura 104).

Ao concluir o processo de desenho da regra de ação para o cenário atual, procedemos à execução do mesmo. Para isso, deve-se aceder à *dashboard* e iniciar o processo ao clicar no botão observado na Figura 105. De seguida, o sistema notifica o utilizador com o início do processo e uma tarefa é criada na zona de tarefas pendentes (Figura 106).

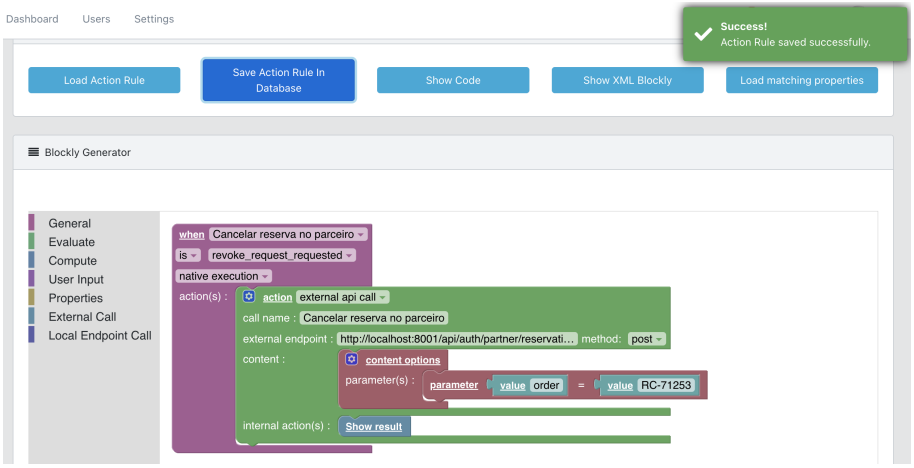

Figura 104. Armazenamento da regra de ação "Cancelar reserva no parceiro".

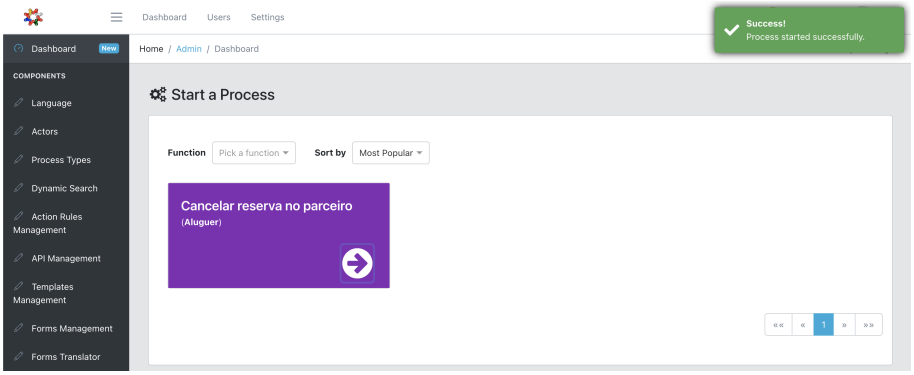

Figura 105. Armazenamento da regra de ação "Cancelar reserva no parceiro".

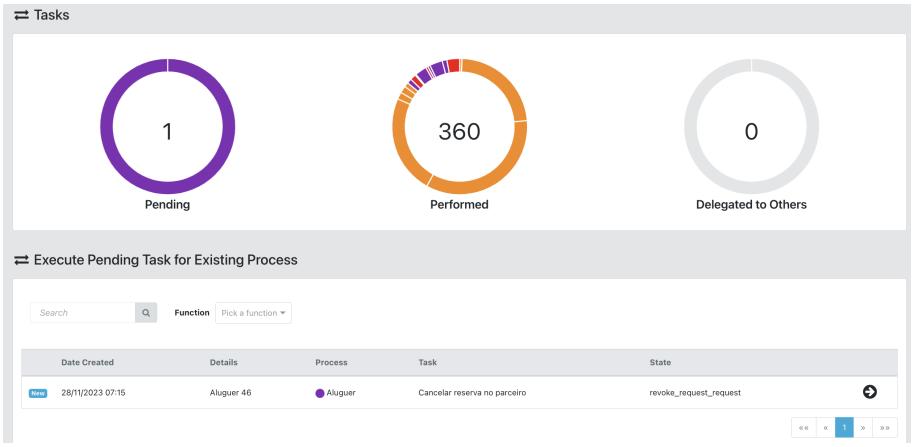

Figura 106. Início do processo de execução de cancelar reserva no parceiro.

Depois do botão de iniciar a tarefa, presente na Figura 106, ser pressionado, o sistema inicia o tratamento do fluxo de informações e processos de acordo com a especificação da regra de ação. Na Figura 107, observamos o resultado devolvido pela RAC parceira após a realização da chamada.

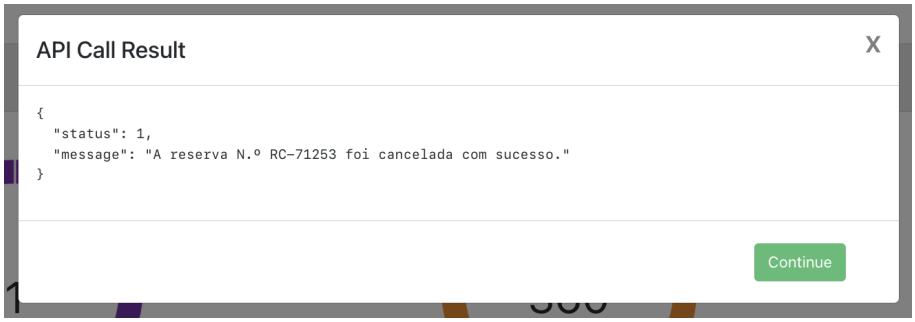

Figura 107. Resultado da execução do processo de cancelar reserva no parceiro.

4.2.4 Consultar tipos de carro do parceiro

Para testar e validar as funções implementadas na nova componente do DISME, usaremos o cenário "Consultar tipos de carro do parceiro".

Considerando a mesma parceria estratégica da validação 4.2.3, imaginemos que a EU-Rent pretende diversificar os tipos de carro que disponibiliza aos clientes e necessita de recorrer a uma RAC parceira, para isso pode consultar através de uma chamada externa os tipos de carro da RAC parceira ou até mesmo um tipo de carro em específico.

O DISME, com a sua nova componente, permite que essa consulta seja realizada por meio do desenho de uma regra de ação. Para isso, seguiremos para a componente *"Action Rules Management"*(Figura 108) e iniciamos o desenho da nova regra de ação com a utilização da peça *"when is do"*, presente no separador *"General"*.

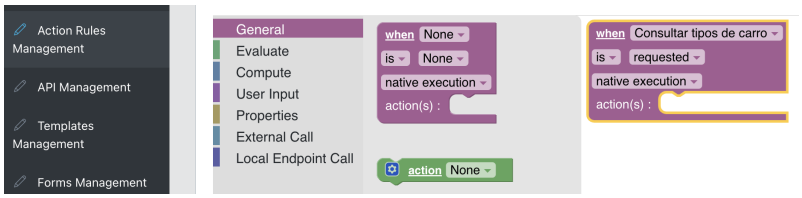

Figura 108. Utilização da componente *"Action Rules Management"* para a criação de uma regra de ação para o exemplo "Consultar tipos de carro do parceiro".

Na Figura 108, ao selecionar o processo atual, "Consultar tipos de carro do parceiro", no estado *"is requested"* o sistema tem a indicação de quando é que o processo deverá iniciar e podemos continuar com o desenho da regra de ação definido a sua única ação, *"external api call"* (Figura 109). O campo *"call name"* foi definido como "Consultar tipos de carro de parceiro"e o método selecionado para a chamada é o *GET*.

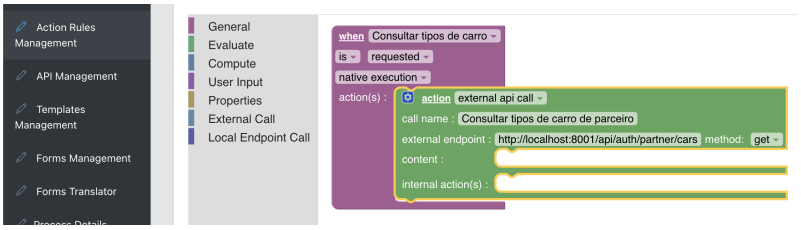

Figura 109. Utilização da peça *"when is do"* e *"action"* para a criação de uma regra de ação para o exemplo "Consultar tipos de carro do parceiro".

Após a configuração base da ação iremos complementá-la com a configuração do seu *"content"* e das suas *"internal action(s)"*. O *endpoint* utilizado requer a indicação do tipo de carro que pretendemos consultar. Neste caso, o tipo de carro pretendido é económico logo enviamos o parâmetro *type* com o valor *economy* ao realizar a chamada. Para esse fim, ao aceder ao separador *"External Call"* utilizamos as peças *"parameter"* e *"value"* para a indicação do tipo de carro (Figura 110).

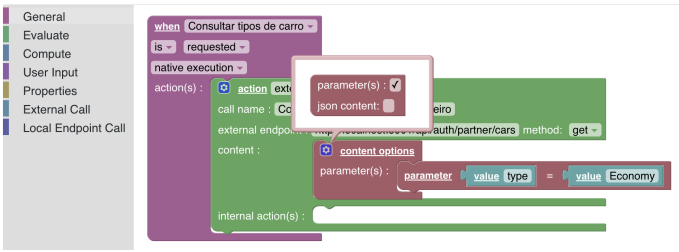

Figura 110. Configuração do tipo de carro utilizando as peças *"content options"*, *"parameter"e "value"* no exemplo "Consultar tipos de carro do parceiro".

O resultado final desejado neste cenário é mostrar ao utilizador os tipos de carro económicos da RAC parceira. Nesse sentido, devemos utilizar a peça *"show result"* e finalizar a conceção da regra de ação utilizando o botão *"Save Action Rule In Database"* para armazená-la como observado na Figura 111.

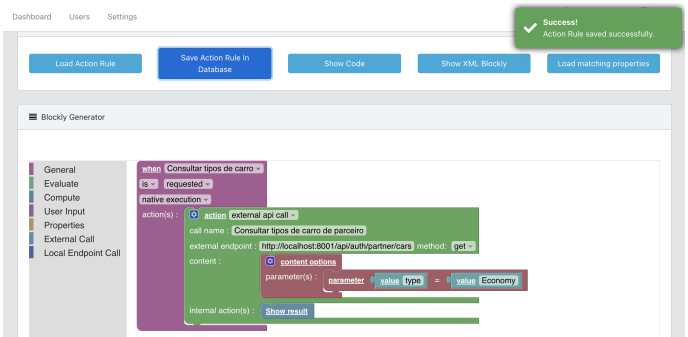

Figura 111. Utilização da peça *"Show result"* no exemplo "Consultar tipos de carro do parceiro".

Voltando à página inicial da EU-Rent, a *dashboard*, podemos executar a regra de ação inerente ao processo "Consultar tipos de carro do parceiro"pressionando o botão de iniciar o processo (Figura 112). O processo, ao ser iniciado pelo sistema, notifica o utilizador com uma notificação de sucesso e coloca-o na lista de tarefas pendentes do utilizador (Figura 113).

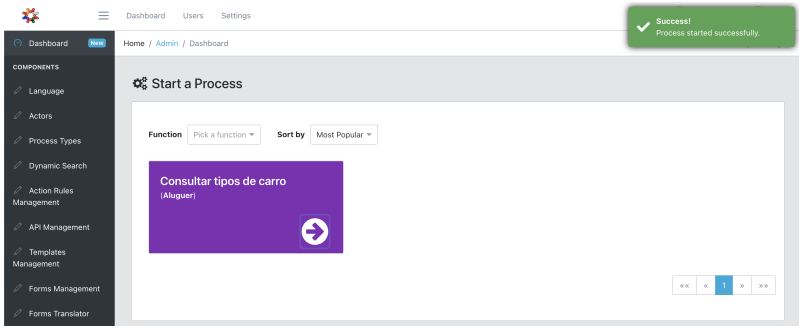

Figura 112. Início do processo de consultar reservas do parceiro.

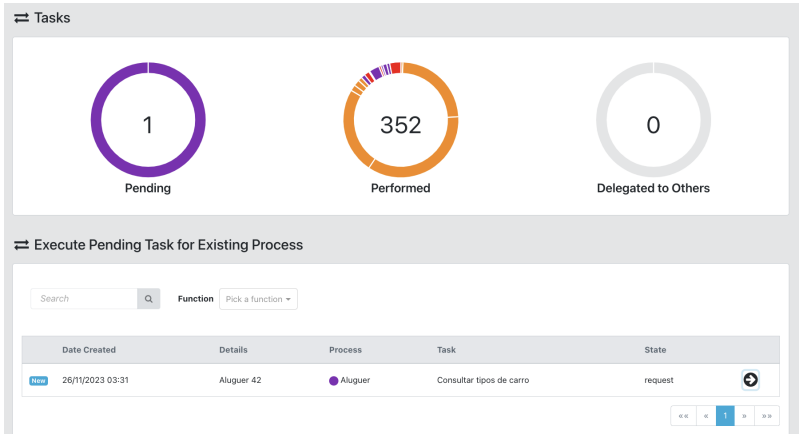

Figura 113. Tarefa de consultar tipos de carro do parceiro inserida na lista de pendentes.

O botão de continuar o processo, presente na lista de tarefas pendentes, ao ser utilizado executa a regra de ação. Na Figura 114, observamos o resultado de realizar uma *external api call* no contexto de uma regra de ação para consultar tipos de carro do parceiro.

Figura 114. Resultado da execução do exemplo consultar tipos de carro do parceiro.

4.2.5 Importar lista de carros de parceiro

Neste cenário de interoperabilidade, abordaremos a ação de importar a lista de carros de um parceiro recorrendo a uma *external api call*. O Objetivo é garantir a criação de vários valores internos à EU-Rent em simultâneo.

Imaginemos que a EU-Rent inicia uma parceria com outra RAC e que necessita de importar para o seu sistema a lista de carros da nova parceira. Para garantir que a lista de carros seja totalmente importada é possível, através da ação interna *"create entity(ies)"*, inserir vários carros no sistema utilizando uma REST API.

Para iniciar o desenvolvimento deste processo devemos aceder ao separador *"Action Rules Management"* e desenhar a regra de ação responsável pelo tratamento do novo processo. Utilizando a peça *"when is do"* seleciona-se o processo que pretendemos desenvolver, neste caso "Importar lista de carros de parceiro", e selecionar o estado *"is requested"(Figura 115)*.

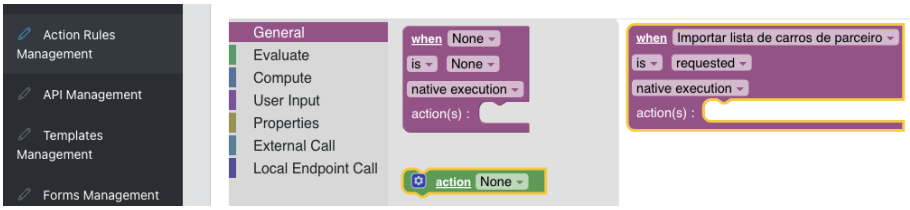

Figura 115. Desenvolvimento do processo "Importar lista de carros de parceiro"utilizando a componente *"Action Rules Management"*.

Em seguida, utilizaremos a peça *"action"* com a ação *external api call*. Identificaremos a chamada como "Chamada *GET* importar lista de carros de parceiro". O *endpoint* e método utilizado são fornecidos pelo novo parceiro (Figura 116).

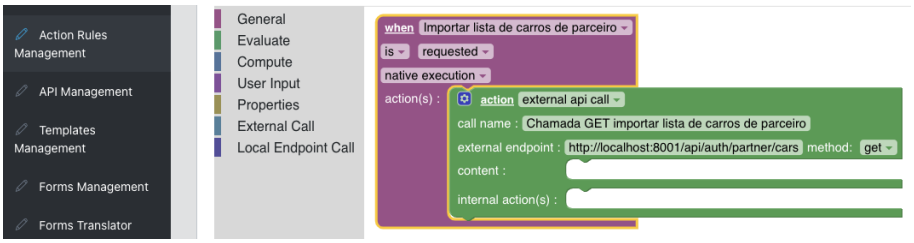

Figura 116. Configuração da peça *"action"* para o exemplo "Importar lista de carros de parceiro".

Na Figura 117 visualizamos o separador *"External Call"* com todas as peças auxiliares para as *action* do tipo *"external api call"*. Para continuar o desenho da nova regra, utilizamos a peça *"content options"* sem configurações adicionais e a peça *"create entity(ies)"* que realizará o armazenamento dos valores importados do SI externo.

Ao utilizar a peça *"create entity(ies)"*, estamos a indicar ao sistema que devem ser criadas novas entidades. Para especificar que entidades é que devem ser criadas com o resultado da realização da chamada, na peça devemos selecionar no campo *"entity type"* o tipo de entidade interna ao sistema, no caso atual é carro (Figura 118).

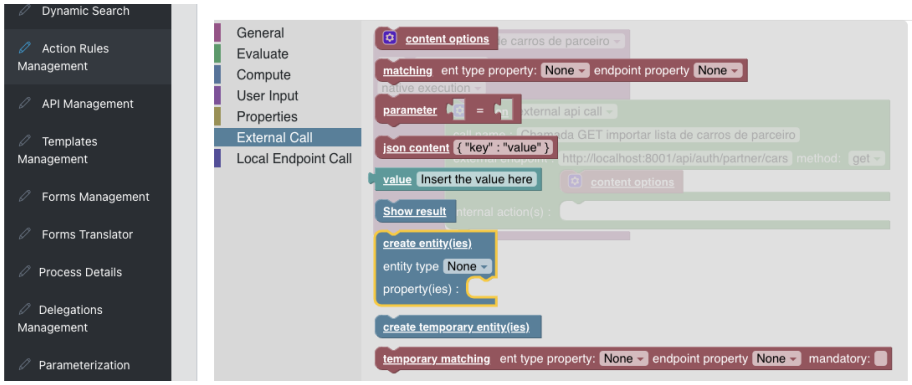

Figura 117. Separador *"External Call"* com peças auxiliares para o exemplo "Importar lista de carros de parceiro".

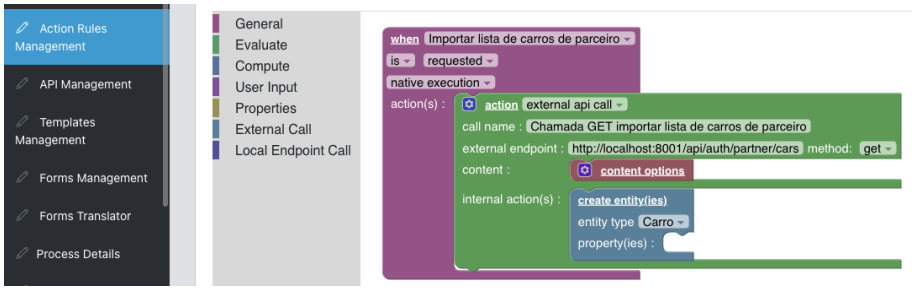

Figura 118. Utilização da peça *create entity(ies)* para o exemplo "Importar lista de carros de parceiro".

Uma entidade interna ao sistema tem propriedades. O tipo de entidade carro tem duas propriedades associadas, estas são: nome e tipo de carro. Assim sendo, utilizando a peça *"matching"* é crucial realizar a correspondência das propriedades internas com as propriedades externas. A Figura 119 demonstra a realização de uma chamada auxiliar para identificar as propriedades externas ao sistema, esta chamada permite identificar quantas entidades externas foram identificadas e quais as suas propriedades.

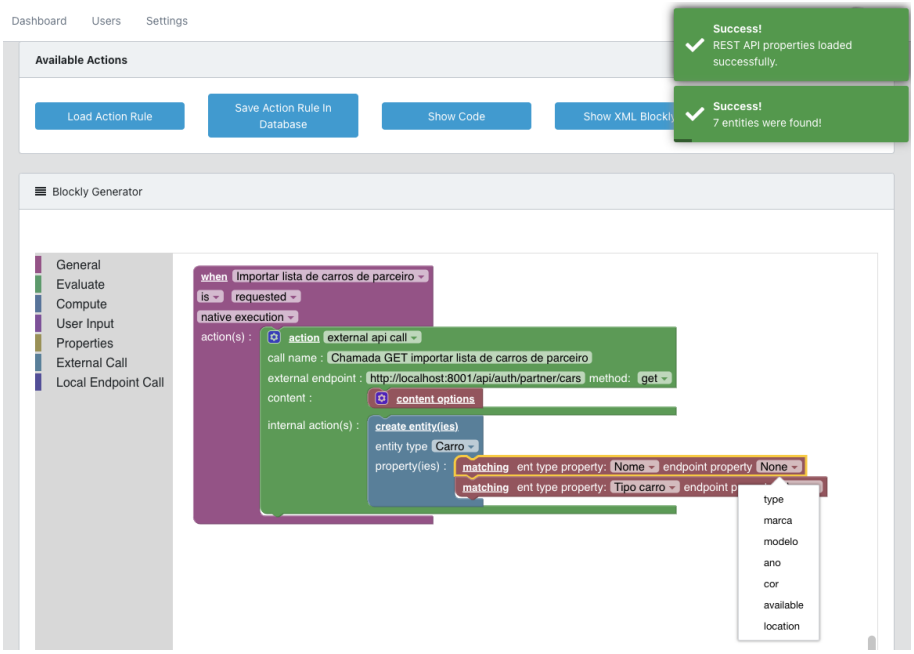

Figura 119. Correspondência das propriedades internas com as propriedades externas na regra de ação "Importar lista de carros de parceiro".

Após a correspondência das propriedades e a finalização da *action "external api call"* inserimos uma nova ação disponível no sistema. Esta ação permite configurar uma notificação ou caixa de texto com informações ou mensagens para o utilizador final. No contexto desta regra de ação, após a importação da lista de carros, irá ser exibida uma caixa de texto ao utilizador final com a mensagem "A lista de carros foi importada com sucesso."como observado na Figura 120.

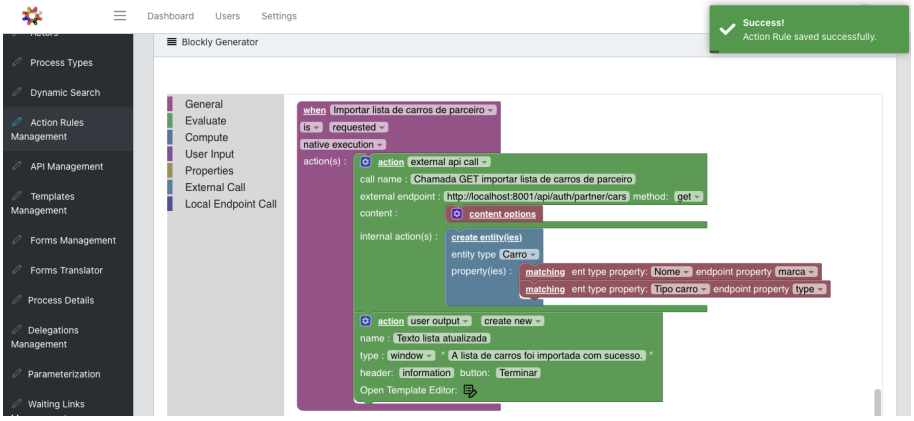

Figura 120. Utilização da peça *"action"* com a opção *"user output"* no exemplo "Importar lista de carros de parceiro".

Após a regra de ação ser guardada voltamos à *dashboard* e iniciamos o processo "importar lista de carros de parceiro"ao clicar no botão para o efeito (Figura 121). Aguardamos uma notificação do sistema de que o processo foi iniciado e adicionada à lista de tarefas pendentes (Figura 122).

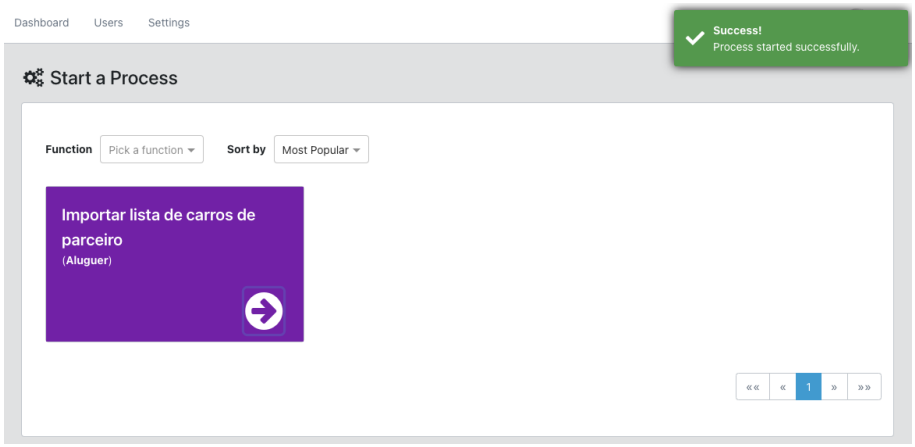

Figura 121. Início do processo de execução de importar lista de carros de parceiro.

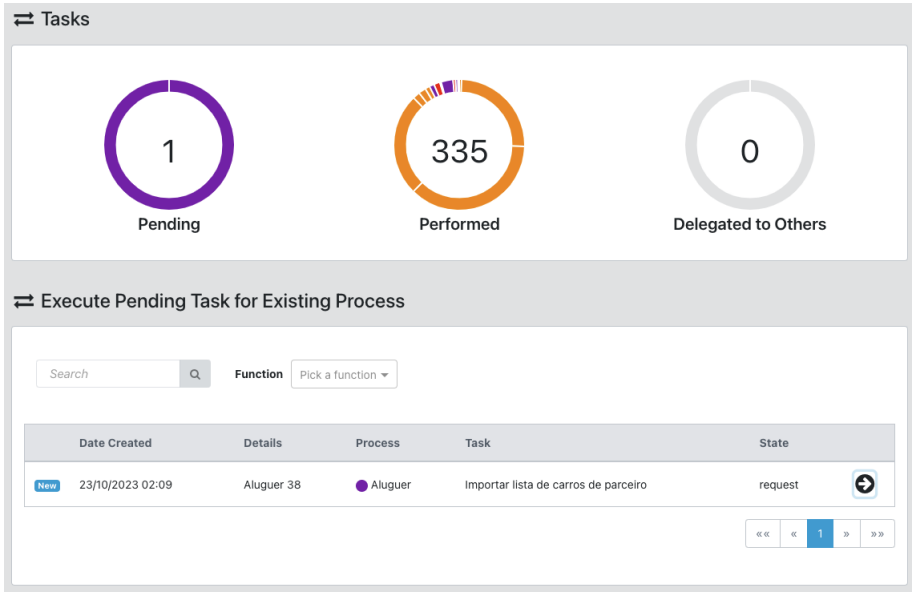

Figura 122. Execução do processo de importar lista de carros de parceiro.

Depois de pressionar o botão presente na lista de tarefas pendentes para continuar a tarefa atual, o sistema inicia o tratamento do fluxo de informações e processos de acordo com a especificação da regra de ação. Na Figura 123 observamos que o sistema, ao executar o processo, encontrou 7 entidades do tipo carro e que as mesmas foram inseridas no sistema da EU-Rent com a correspondência pretendida.

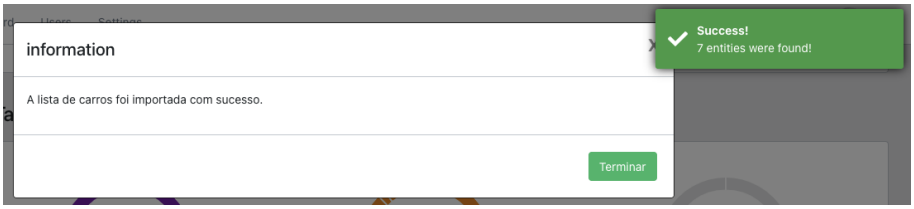

Figura 123. Resultado da execução do processo importar lista de carros de parceiro.

4.2.6 Contratação de aluguer

O cenário que será desenvolvido de seguida é a contratação de um aluguer na EU-Rent. Neste cenário, iremos desenhar a regra de ação que representa o processo de um cliente da EU-Rent iniciar uma contratação de aluguer utilizando os carros disponíveis de um parceiro da mesma.

Utilizando a nova ação interna *"create temporary entity(ies)"* é possível obter a lista atualizada de carros de um parceiro da EU-Rent e exibir ao utilizar para que o mesmo possa selecionar um carro no momento em que realiza um pedido de contratação de aluguer. A lista de carros exibida ao utilizador é guardada no sistema temporariamente.

Para iniciar o processo de desenho desta regra de ação devemos aceder à componente *"Action Rules Management"*, como apresentado na Figura 124, e utilizar a peça *"when is do"* arrastando-a para a área de trabalho.

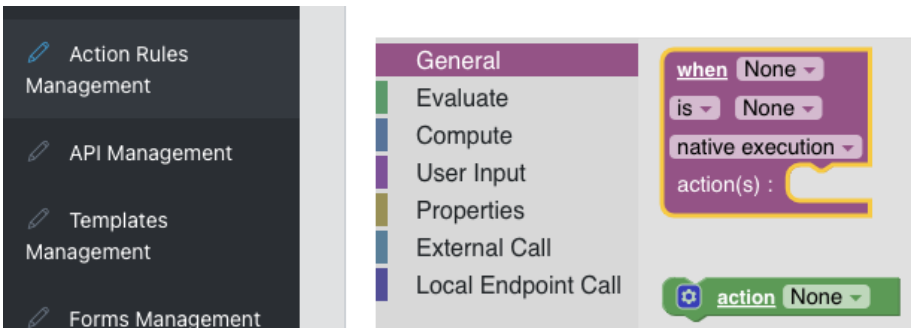

Figura 124. Desenvolvimento do processo "Contratação de aluguer"utilizando a componente *"Action Rules Management"*.

Na Figura 125 observamos a peça *"when is do"* configurada de modo a iniciar o processo "Contratação de aluguer"quando o mesmo for solicitado.

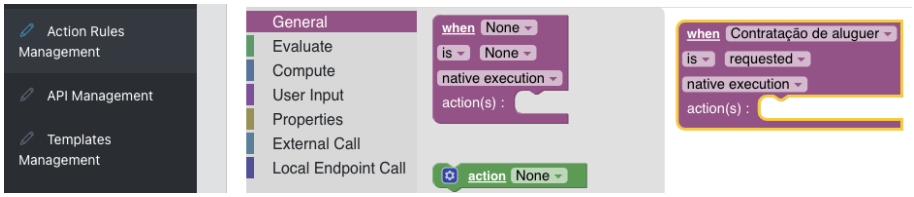

Figura 125. Desenvolvimento do processo "Contratação de aluguer"utilizando a componente "Action Rules Management".

De seguida conectamos a peça *"action"*, inserimos o nome da ação como "Chamada GET lista de carros", colocamos o *endpoint* fornecido pela RAC parceira e escolhemos o método de leitura, *GET*. Colocamos a peça *"content options"* no conteúdo e conectamos a peça *"create temporary entity(ies)"* nas ações internas, ambas as peças disponíveis no separador *"External Call"* (Figura 126). Neste momento a regra de ação está desenhada para, ao ser iniciada, realizar uma *external api call* ao *endpoint* inserido e guardar todos os valores obtidos em variáveis temporárias para serem utilizados mais tarde.

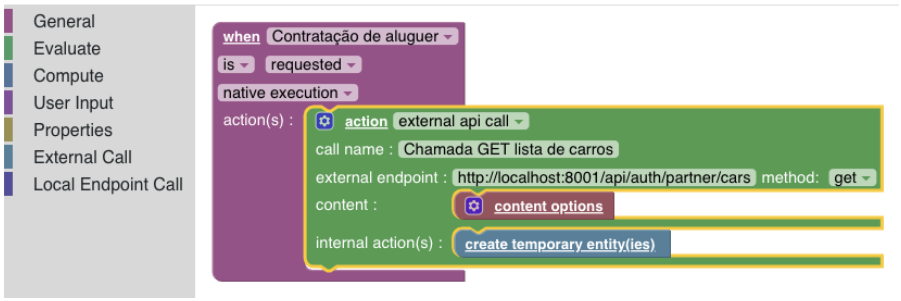

Figura 126. Utilização da peça *"create temporary entity(ies)"* para o exemplo "Contratação de aluguer".

Na Figura 127 inserimos uma *action* de *user output* ao utilizador. Esta ação disponibiliza ao utilizador uma notificação em formato caixa de texto. A configuração realizada irá exibir uma mensagem de boas vindas ao cliente no momento em que o mesmo iniciar o processo de contratação.

Na Figura 128 inserimos uma nova *action*, desta vez, para sinalizar o sistema de que existe a necessidade de criar uma entidade através do preenchimento de campos pelo cliente da EU-Rent.

Nesta *action* selecionamos a opção *"user input of a single entity type"* e chamamos de "Formulário de contratação". No campo *"scope"* selecionamos a entidade que pretendemos registar valores, neste exemplo, aluguer e de seguida conectamos várias peças do tipo *"property"* disponíveis no separador *"Properties"*.

Nas peças do tipo *property* selecionamos as propriedades do tipo aluguer que pretendemos guardar valores, podemos selecionar várias e ativar a opção de obrigatoriedade. No caso de validação atual iremos solicitar o nome do condutor, a data de nascimento do condutor, verificar se a carta de condução é valida, a data de início e fim da contratação e se o cliente pretende uma viatura com GPS.

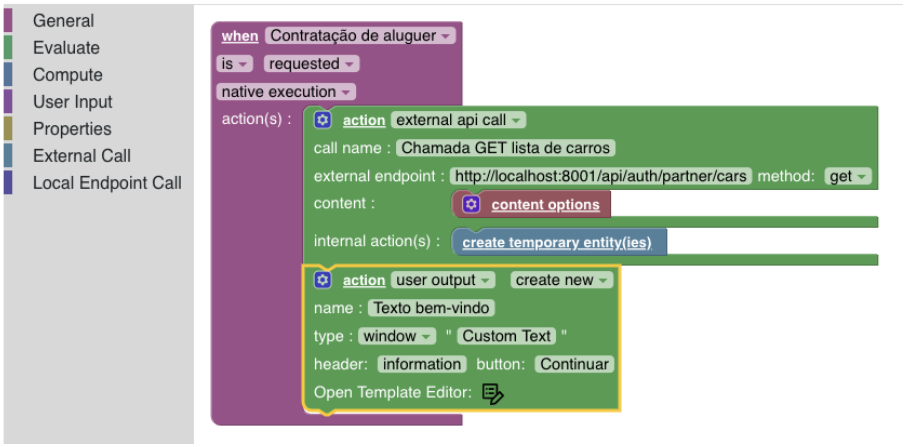

Figura 127. Utilização da peça *"action"* com a opção *"user output"* no exemplo "Contratação de aluguer".

Figura 128. Utilização da peça *"action"* com a opção *"user input of a single entity type"* no exemplo "Contratação de aluguer".

Após a peça *"temporary matching"*, exibida na Figura 129, ser colocada, permitindo a realização da correspondência entre as propriedades internas e externas, é necessário utilizar o botão *"Load matching properties"* para a realização de uma chamada auxiliar que identifica as propriedades externas ao sistema e importa-as na lista da *endpoint property*.

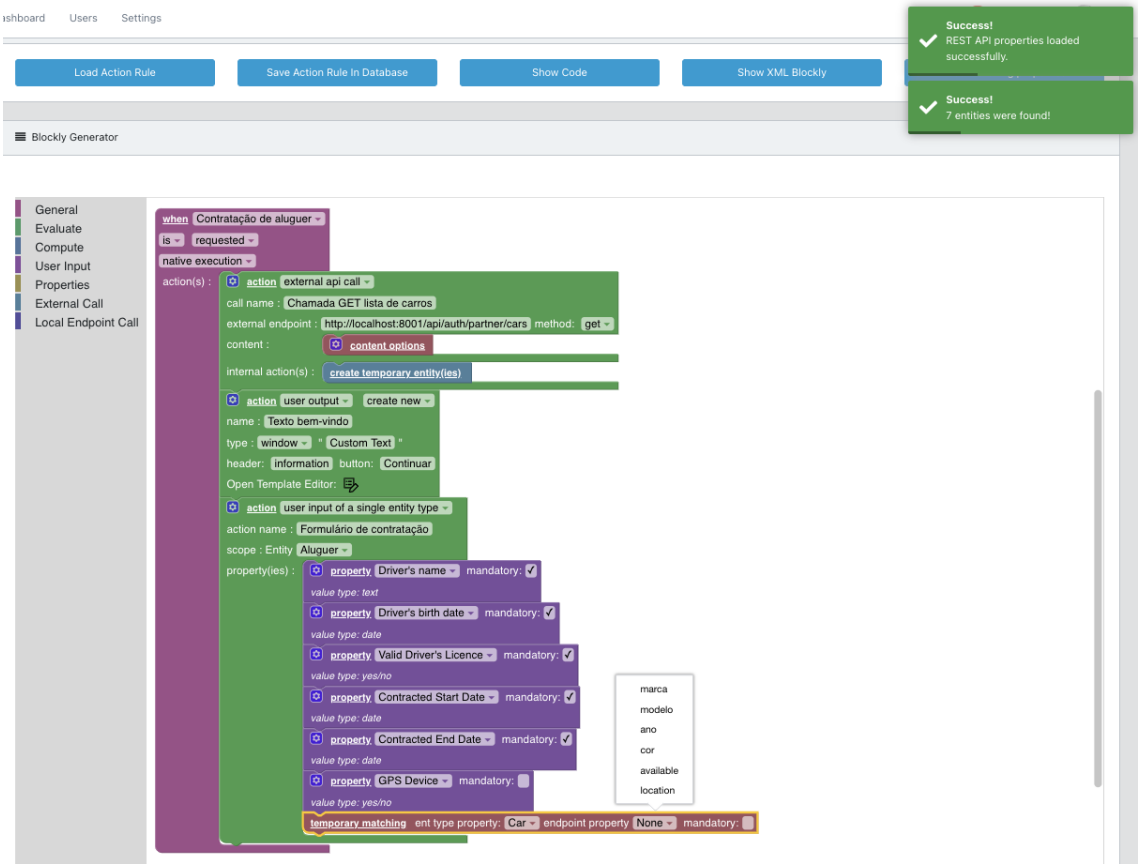

Figura 129. Correspondência das propriedades internas com as propriedades temporárias externas na regra de ação "Contratação de aluguer".

Como a chamada à REST API da RAC parceira e a criação das entidades temporárias será executada na primeira *action*, devemos realizar nesta etapa a correspondência da propriedade interna *car* com uma propriedade temporária externa, no nosso cenário, pretendemos que seja a marca.

A próxima etapa é adicionar uma nova *action*, neste caso, de *user output* para apresentar ao utilizador uma caixa de texto com uma mensagem de agradecimento. Posto isto, o desenho da regra de ação encontra-se concluído e podemos armazená-lo utilizando o botão *"Save Action Rule In Database"* (Figura 130).

As *actions* do tipo *"user input of a single entity type"* requerem a criação de um formulário que é construido na componente *Forms Management*. Acedendo à componente, clicamos no botão para criar um formulário e denominamos de "Formulário de contratação", selecionamos a *action* da regra de ação que acabámos de criar (Figura 131) e iniciamos o desenho do formulário.

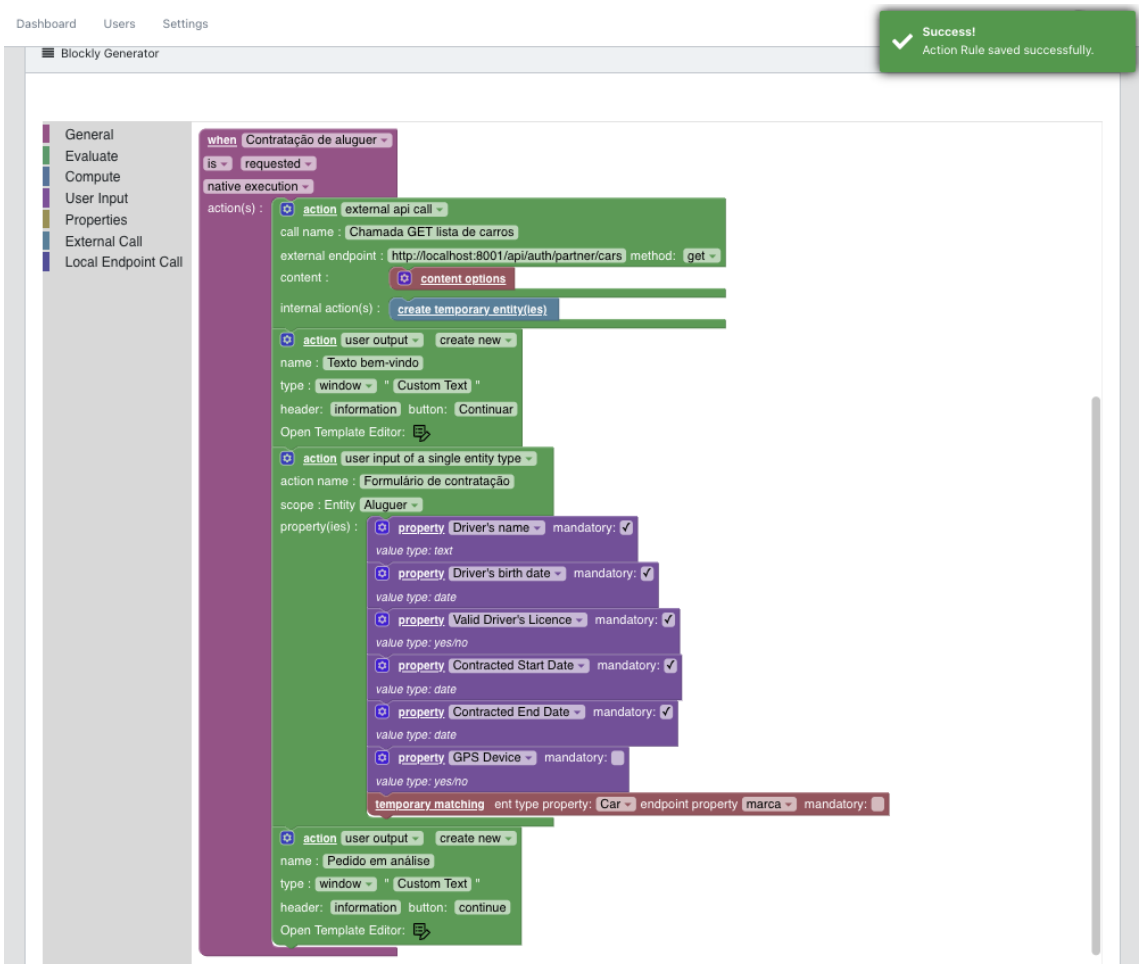

Figura 130. Armazenamento da regra de ação "Contratação de aluguer".

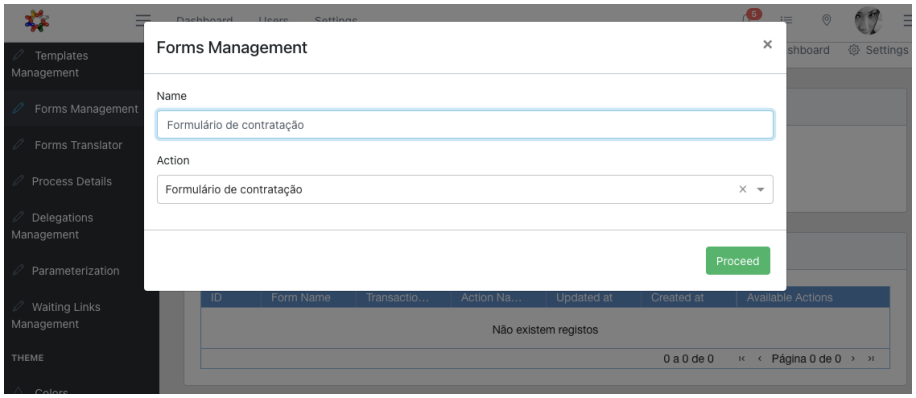

Figura 131. Componente *Forms Management* de gestão de formulários.

Ao carregar o formulário o sistema disponibiliza em formato *drag-and-drop* todas as propriedades que foram selecionadas no contexto da construção da regra de ação em formato de retângulo. Na Figura 132 observamos o aspeto final do formulário depois da sua configuração.

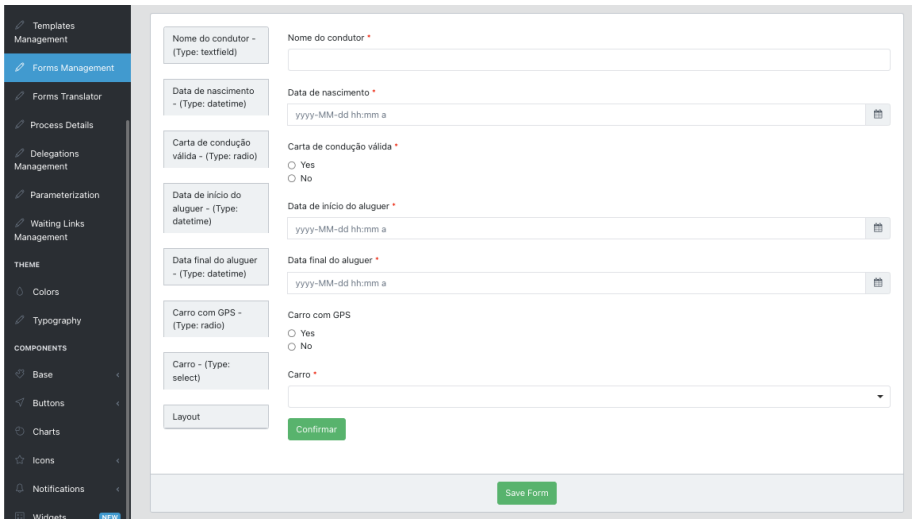

Figura 132. Interface para a criação do formulário para a contratação de aluguer.

Ao armazenar o formulário construido concluímos a última etapa necessária para a construção do processo contratação de aluguer. Desta forma iniciamos o processo ao clicar no botão "Contratação de aluguer"presente na Figura 133.

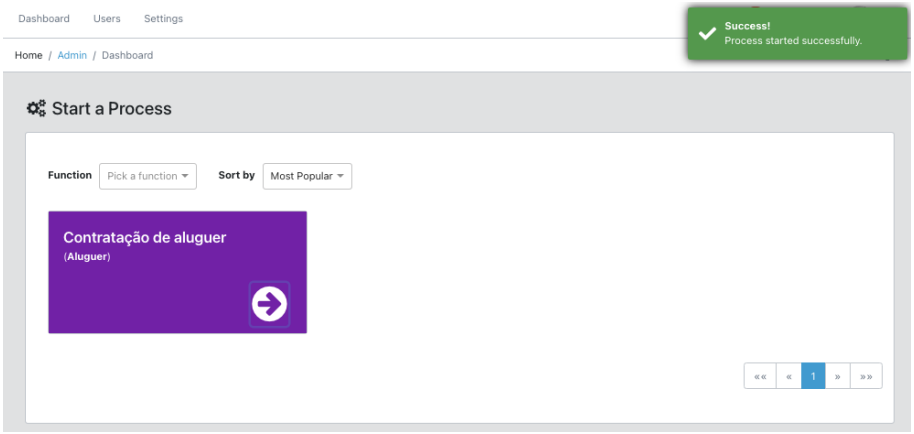

Figura 133. Início do processo de execução de contratação de aluguer.

O sistema, ao iniciar o processo informa o utilizador notificando o mesmo de que o processo foi iniciado e insere o processo na lista de tarefas pendentes do utilizador (Figura 134). Para iniciar a contratação de aluguer clicamos na seta presente na tarefa e o sistema inicia o processo.

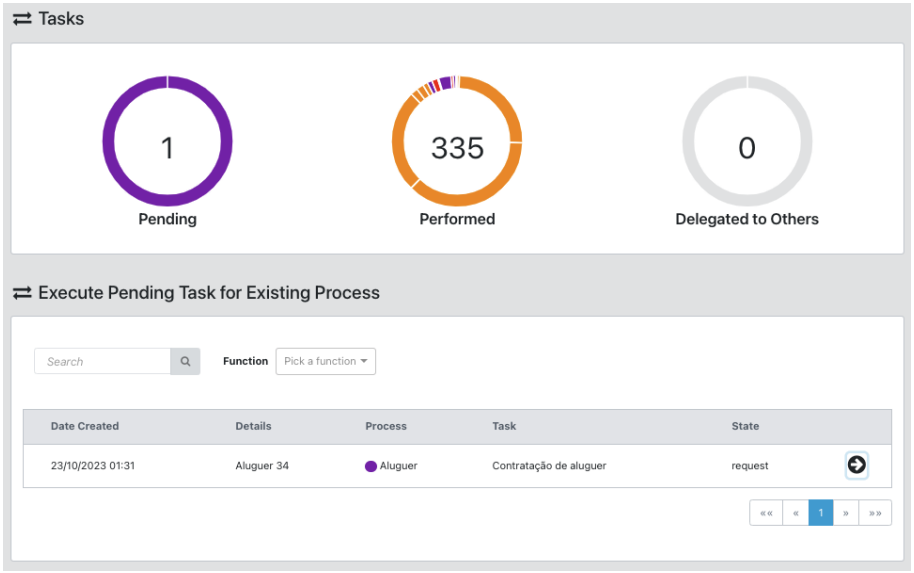

Figura 134. Execução do processo de contratação de aluguer.

Ao iniciar o processo o sistema executa todo o fluxo de informações e processos de acordo com a especificação da regra de ação começando pela primeira *action* a *external api call*. Nesta fase de execução o sistema realiza a chamada à REST API da RAC parceira e obtém a lista de carros atualizada criando as entidades temporárias com a informação devolvida da chamada e notificando o utilizador.

Na segunda fase do processo o sistema exibe a caixa de texto com a mensagem de boas vindas à EU-Rent configurada na *action user output* (Figura 135).

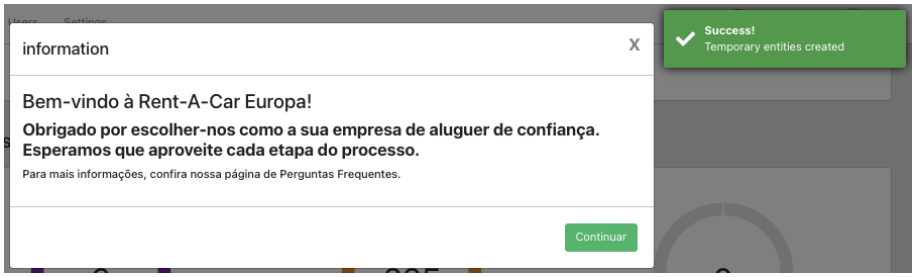

Figura 135. Resultado da execução das *actions external api call* e *user output*.

Após a execução das primeiras ações definidas, o sistema exibe ao cliente o formulário com os campos solicitados e com a lista de carros proveniente da RAC parceira. A Figura 136 mostra o formulário preenchido com as informações inseridas por um cliente.

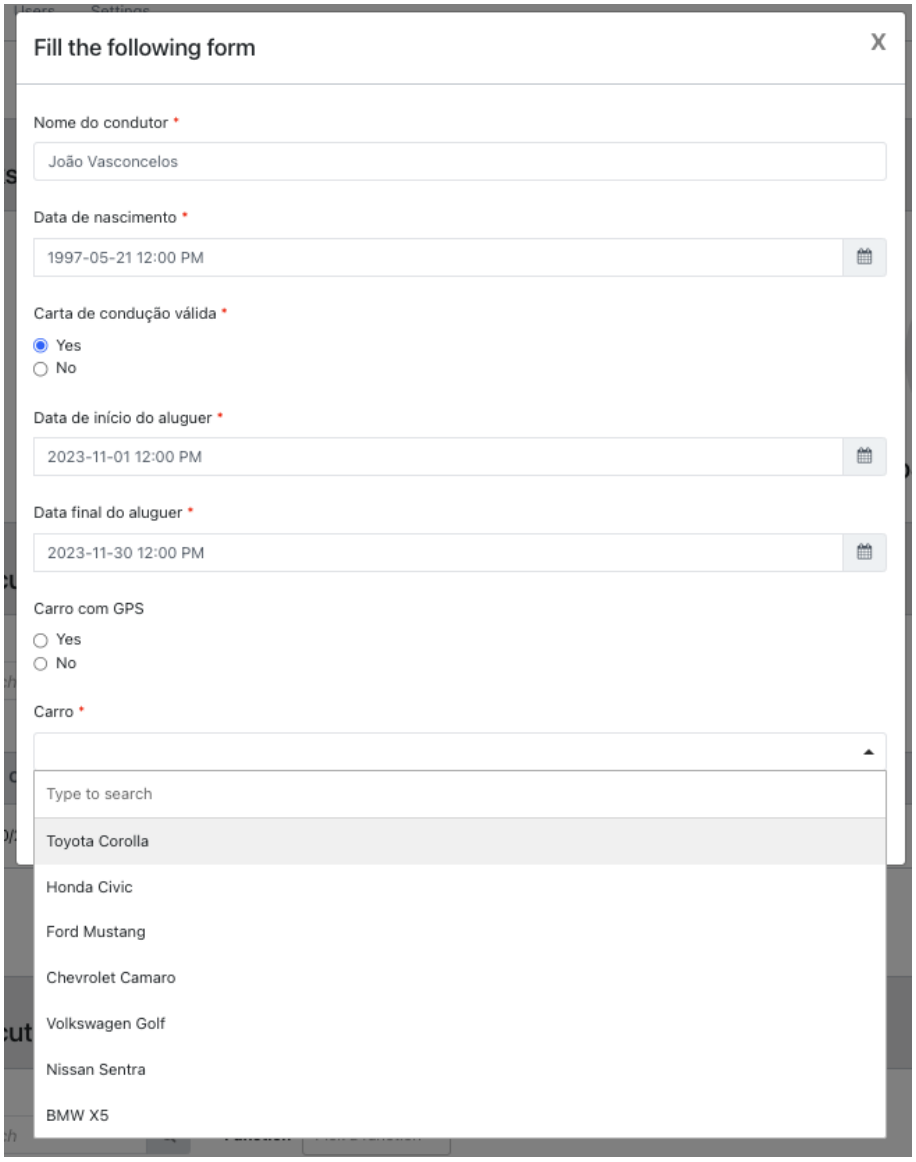

Figura 136. Formulário de contratação de aluguer preenchido por cliente.

Na Figura 137 observamos a execução da última *action user output* onde o sistema finaliza a execução do pedido de contratação de aluguer com uma mensagem informativa.

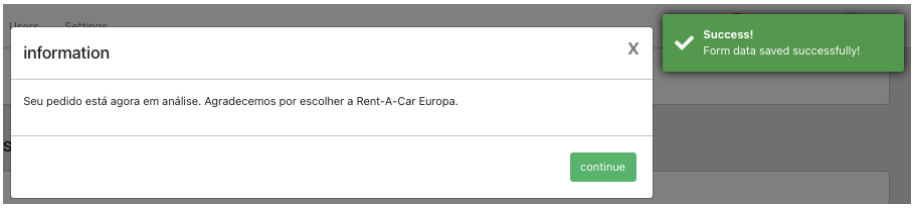

Figura 137. Mensagem final exibida em caixa de texto na contratação de aluguer.

5 Conclusão

O desenvolvimento e a implementação bem sucedida da componente *API Management* no DISME representa um marco significativo na procura de soluções eficazes no contexto de interoperabilidade de SI. Ao longo desta dissertação, exploramos as complexidades desse domínio, detalhando a concetualização, execução e validação da nova componente.

Abordamos a necessidade de promover a interoperabilidade entre sistemas de informação, sendo esta, fundamental em diversos domínios desde o setor da saúde até o empresarial para garantir a troca eficiente de informações entre sistemas heterogéneos.

Concluímos que a interoperabilidade é um pilar em crescimento nas tecnologias de informação atuais. Com os avanços tecnológicos as organizações percebem que não podem ter os seus sistemas isolados, e a necessidade de comunicação entre os sistemas de informação das organizações torna-se uma prioridade.

Ao procurar soluções, entramos na fase de implementação onde aprimoramos a *low-code platform* DISME. O resultado da implementação foi uma plataforma que permitiu a criação de regras de ação capazes de desencadear a comunicação e a execução de tarefas em resposta a processos específicos. Estas regras de ação reduzem intervenções manuais que provocam o risco de erros.

Ficou claro que proporcionar ferramentas intuitivas para criar e gerir as regras de ação é essencial. A interface do DISME oferece a utilizadores de todos os níveis de experiência a capacidade de construir, modificar e gerir essas regras, tornando a interoperabilidade acessível a uma vasta gama de utilizadores, independentemente da sua especialização em integração de sistemas.

O capítulo de validação tornou-se crucial nesta dissertação, pois submetemos o DISME a testes rigorosos abrangendo cenários reais, como consulta a sistemas externos, importação de dados e execução de processos complexos. Os resultados obtidos no capítulo são fundamentais para as nossas conclusões.

A validação revelou que o DISME é uma ferramenta robusta e eficaz para promover a interoperabilidade. Nas situações de teste, o sistema mostrou-se apto a executar as ações apresentadas. Esta confirmação valida a abordagem adotada e revela a sua utilidade em situações reais.

No decorrer da dissertação, ficou evidente que a flexibilidade do DISME é importante. O sistema mostrou-se capaz de se adaptar a diversos tipos de integrações, permitindo a personalização de acordo com as necessidades específicas de cada validação. Esta flexibilidade é essencial, considerando os diversos sistemas e protocolos com os quais nos deparamos.

Este projeto não só representa um marco no caminho da interoperabilidade, mas também uma validação da viabilidade de soluções práticas e flexíveis. Enquanto a sociedade e as organizações continuarem a depender da eficiente troca de informações, a interoperabilidade mantém-se no centro das inovações. O DISME surge como uma ferramenta valiosa para alcançar este objetivo, pois oferece uma abordagem acessível e flexível.

Contudo, é fundamental reconhecer que a interoperabilidade é um campo em constante mudança, à medida que novas tecnologias e padrões surgem, soluções como o DISME devem adaptar-se e evoluir os seus sistemas de informação de forma a atender às mudanças necessárias. Salientamos a importância da colaboração e parcerias entre organizações, a interoperabilidade envolve a inte-

gração de sistemas de informação de diferentes entidades e a cooperação é essencial para o êxito destas iniciativas.

Em conclusão, este projeto contribui significativamente para promover a interoperabilidade no DISME e demonstrou a viabilidade de soluções práticas e flexíveis. Além disso, é importante ressaltar que os resultados e descobertas deste projeto estendem-se para além desta dissertação, sendo incorporados em dois artigos científicos. Estas publicações representam uma valiosa extensão deste estudo, expandindo a sua contribuição na área. Com inovação contínua e colaboração, podemos esperar um futuro onde a interoperabilidade torna-se fundamental na eficácia de troca de informações em sistemas de informação.

5.1 Limitações e trabalho futuro

Durante o desenvolvimento e implementação da componente *API Management* no DISME, surgiram várias limitações que representaram desafios significativos que impactaram, numa fase inicial, o desenvolvimento do projeto. As duas principais limitações encontradas foram:

1. ***Low-code platform* de interoperabilidade *open-source*:** Uma das barreiras enfrentadas no projeto foi a dificuldade em encontrar *low-code platforms open-source* que oferecessem soluções de interoperabilidade com REST APIs. A interoperabilidade é uma parte essencial na comunicação entre sistemas de informação, encontrar ferramentas *low-code platform* que fossem totalmente *open-source* foi uma tarefa desafiadora.
2. **REST APIs**: Outra limitação enfrentada foi a escassez de REST APIs públicas que permitissem a execução de ações como *GET, UPDATE, PUT* e *DELETE*. Muitos sistemas de informação e serviços não disponibilizam essas APIs publicamente, o que dificultou a capacidade do DISME de integrar-se com outros sistemas.

À medida que o projeto DISME evolui e procura continuamente melhorar a sua interoperabilidade e eficiência na troca de informações entre sistemas de informação, surgem duas áreas que podem ser exploradas futuramente essas áreas são:

1. **Implementação de segurança e autenticação:** Um dos trabalhos futuros é a implementação de medidas de segurança e autenticação ao realizar *external api calls*. A segurança dos dados transmitidos é uma preocupação fundamental em qualquer ambiente, isto é especialmente relevante quando se trata de sistemas de informação que partilham dados sensíveis. A implementação de autenticação e medidas de segurança na execução das *external api calls* é essencial para proteger os dados e garantir a integridade das operações.
2. **Documentação das REST APIs:** Outra área de desenvolvimento futuro envolve a criação de uma componente dedicada à verificação de documentação das REST APIs utilizadas na realização de *external api calls*, possivelmente utilizando ferramentas como *Swagger IO* [11]. Ter uma ferramenta que possa analisar, verificar e validar automaticamente a documentação das APIs é essencial para interoperabilidade eficaz. Isto ajudaria os utilizadores a entenderem as funcionalidades, estruturas e requisitos das APIs de forma clara e concisa.

[11] `https://swagger.io/`

Referências

[1] S. Katuu, "Enterprise resource planning: Past, present, and future," *New Review of Information Networking*, vol. 25, 2020.

[2] D. Aveiro and V. Caires, "DEMO model based rapid REST API management in a low code platform," in *Proceedings of the 22nd CIAO! Doctoral Consortium, and Enterprise Engineering Working Conference Forum 2022 co-located with 12th Enterprise Engineering Working Conference (EEWC 2022), November 2-3, 2022, Leusden, the Netherlands*, ser. CEUR Workshop Proceedings, S. Guerreiro, C. Griffo, and M. Jacob, Eds., vol. 3388. CEUR-WS.org, 2022. [Online]. Available: https://ceur-ws.org/Vol-3388/paper2.pdf

[3] V. Caires, J. Vasconcelos, D. Pinto, V. Freitas, and D. Aveiro, "Rapid REST API management in a DEMO based low code platform," in *Advances in Enterprise Engineering XVI - 13th Enterprise Engineering Working Conference, EEWC 2023, Vienna, Austria, November 28-29, 2023, Revised Selected Papers*, ser. CEUR Workshop Proceedings. CEUR-WS.org, 2023.

[4] HIMSS, "Interoperability in healthcare," *HIMSS*, 2023, acedido em: 20 de julho de 2023. [Online]. Available: https://www.himss.org/resources/interoperability-healthcare

[5] Empeek, "Why is interoperability important in healthcare in 2023?" *Empeek*, 2023. [Online]. Available: https://empeek.com/why-is-interoperability-important-in-healthcare/

[6] HIMSS, "Interoperability progress and the challenges and opportunities ahead," *HIMSS*, 2021. [Online]. Available: https://www.himss.org/resources/interoperability-progress-and-challenges-and-opportunities-ahead

[7] Forbes, "The importance of fully interoperable healthcare systems," *Forbes Tech Council*, 2021. [Online]. Available: https://www.forbes.com/sites/forbestechcouncil/2021/05/28/the-importance-of-fully-interoperable-healthcare-systems/

[8] Postman. (2020) What is a rest api? examples, uses challenges. Originalmente publicado a 9 de julho 2020, Acedido em: 22 de julho de 2023. [Online]. Available: https://blog.postman.com/rest-api-examples/

[9] IBM. (2023) What is a rest api? Acedido em: 22 de julho de 2023. [Online]. Available: https://www.ibm.com/topics/rest-apis

[10] Stackify. (2023) What are crud operations? examples, tutorials more. Acedido em: 22 de julho de 2023. [Online]. Available: https://stackify.com/what-are-crud-operations/

[11] Hevo. (2023) Rest api best practices and standards in 2023. Acedido em: 22 de julho de 2023. [Online]. Available: https://hevodata.com/learn/rest-api-best-practices/

[12] "O que é XML? - Explicação sobre Extensible Markup Language (XML - Linguagem extensível de marcação) - AWS," https://aws.amazon.com/what-is/xml/, acedido em: 23 de julho de 2023.

[13] "XML Introduction - XML: Extensible Markup Language," https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction, acedido em: 23 de julho de 2023.

[14] Ramotion, "Exploring rest api: Definition, benefits, and key features," Ramotion, 2013. [Online]. Available: https://www.ramotion.com/blog/rest-api/

[15] w3schools, "What is json," https://www.w3schools.com/whatis/whatis_json.asp, 2023, acedido em: 24 de julho de 2023.

[16] Yovak, "Advantages and disadvantages of json," Online, N/A, acedido em: 24 de julho de 2023. [Online]. Available: https://yovak.com/advantages-and-disadvantages-of-json/

[17] What is json api? json vs graphql vs rest api comparison - rapidapi. Acedido em: 24 de julho de 2023. [Online]. Available: https://rapidapi.com/blog/api-glossary/what-is-json-api-json-vs-graphql-vs-rest-api-comparison/

[18] What is json. Acedido em: 24 de julho de 2023. [Online]. Available: https://restfulapi.net/introduction-to-json/

[19] "Silk Performer - Web Help: Introduction to SOAP (Simple Object Access Protocol)," https://www.microfocus.com/documentation/silk-performer/205/en/silkperformer-205-webhelp-en/GUID-FEFE9379-8382-48C7-984D-55D98D6BFD37.html, acedido em: 24 de julho de 2023.

[20] Mendix. (2023) Low-code guide. [Online]. Available: https://www.mendix.com/low-code-guide/

[21] D. Aveiro, V. Freitas, E. Cunha, F. Quintal, and Y. Almeida, "Traditional vs. low-code development: comparing needed effort and system complexity in the nexusbrant experiment," in *25th IEEE Conference on Business Informatics, CBI 2023 - Volume 1, Prague, Czech Republic, June 21-23, 2023*. IEEE, 2023, pp. 1–10. [Online]. Available: https://doi.org/10.1109/CBI58679.2023.10187470

[22] M. A. d. Cruz, H. T. de Paula, B. P. Caputo, S. B. Mafra, P. Lorenz, and J. J. Rodrigues, "Olp–a restful open low-code platform," *Future Internet*, vol. 13, no. 1, January 2021.

[23] Budibase. (2023) Overview. Budibase is an open-source low-code platform that helps you build internal tools in minutes... [Online]. Available: https://docs.budibase.com/

[24] V. Freitas, "Extension of action rule grammar and implementation of processing engine of a demo based low-code platform," Master's thesis, University of Madeira, 2023.