PM

Pesquisa Dinâmica e Flexível de Dados em Sistema de Informação *Low-code*

PROJETO DE MESTRADO

Sílvia da Silva Fernandes
MESTRADO EM ENGENHARIA INFORMÁTICA

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

novembro | 2025

Pesquisa Dinâmica e Flexível de Dados em Sistema de Informação *Low-code*

PROJETO DE MESTRADO

Sílvia da Silva Fernandes
MESTRADO EM ENGENHARIA INFORMÁTICA

ORIENTAÇÃO
David Sardinha Andrade de Aveiro

Faculdade de Ciências Exatas e da Engenharia

Mestrado em Engenharia Informática

Pesquisa Dinâmica e Flexível de Dados em Sistema de Informação *Low-code*

Sílvia da Silva Fernandes

Orientado por:

David Sardinha Andrade de Aveiro

Constituição do júri de provas públicas:

Karolina Baras (Professora Auxiliar da Universidade da Madeira), Presidente

Fábio Rúben Silva Mendonça (Professor Auxiliar Convidado da Universidade da Madeira), Vogal

25 de novembro de 2025

Resumo

As plataformas *low-code* têm surgido como resposta à necessidade de sistemas de informação mais flexíveis e adaptáveis, reduzindo a dependência do desenvolvimento de *software* convencional. O DISME (*Dynamic Information System Modeller and Executer*) aparece neste contexto como protótipo capaz de modelar organizações e parametrizar sistemas de informação de forma dinâmica.

Foi desenvolvida uma componente de pesquisa dinâmica integrada no DISME que permite configurar e executar pesquisas de forma visual sem escrever SQL, através de uma interface intuitiva, permitindo que pessoas não técnicas configurem consultas úteis e reutilizáveis. A componente suporta parâmetros dinâmicos e expõe resultados por REST, permitindo que plataformas externas consultem dados guardados na base de dados do projeto.

A arquitetura manteve o modelo EAV (Entidade-Atributo-Valor) utilizando o MySQL como base de dados principal e acrescentou a opção de vistas materializadas em MongoDB para diminuir o tempo de execução em consultas repetidas e com muitas linhas. Para além disso, foi avaliada a leitura por tabelas relacionais convencionais para comparar comportamentos.

Realizaram-se três etapas de avaliação. Nos testes de integração confirmou-se o funcionamento ponta a ponta da pesquisa dinâmica, a criação e reutilização de pesquisas guardadas, a parametrização em tempo de execução e a estabilidade do contrato JSON, incluindo o ciclo das vistas materializadas. Nos testes de *performance* mediram-se três cenários de consulta e verificou-se redução do tempo médio e menor variabilidade quando a leitura ocorreu a partir da vista materializada. Nos testes de usabilidade aplicou-se o SUS a cinco participantes e obteve-se 73,5 pontos em média, com apreciações positivas sobre integração funcional e aprendizagem rápida. Assinalaram-se melhorias de curto prazo na descoberta da tabela de resultados, na criação de filtros e no texto do formulário de publicação por REST. Em conjunto, os resultados evidenciaram uma solução funcional, com ganhos de desempenho e um caminho claro para melhorar a experiência de utilização.

Keywords: plataformas *low-code* · pesquisa dinâmica · NoSQL · DISME · componente · engenharia organizacional

Abstract

Low-code platforms have emerged in response to the need for more flexible and adaptable information systems, reducing reliance on conventional software development. DISME (*Dynamic Information System Modeller and Executer*) appears in this context as a prototype capable of modelling organisations and parameterising information systems dynamically.

A dynamic search component was developed and integrated into DISME that allows users to configure and execute queries visually without writing SQL, through an intuitive interface, enabling non-technical users to design useful and reusable queries. The component supports dynamic parameters and exposes results via REST, enabling external platforms to query data stored in the project's database.

The architecture retained the EAV (Entity-Attribute-Value) model using MySQL as the primary database and added the option of materialised views in MongoDB to reduce execution time for repeated queries with many rows. In addition, reading from conventional relational tables was evaluated to compare behaviour.

Three evaluation stages were carried out. Integration tests confirmed the end-to-end functioning of dynamic queries, the creation and reuse of saved queries, runtime parameterisation, and the stability of the JSON contract, including the cycle of materialised views. In the performance tests, three query scenarios were measured, and a reduction in average time and lower variability were observed when reading from the materialised view. In the usability tests, the SUS was applied to five participants and an average score of 73.5 points was obtained, with positive assessments of functional integration and quick learning. Short-term improvements were noted in the discovery of the results table, the creation of filters, and the text of the REST publication form. Overall, the results showed a functional solution, with performance gains and a clear path to improving the user experience.

Keywords: low-code platforms · dynamic search · NoSQL · DISME · component · enterprise engineering

Agradecimentos

A realização deste trabalho não teria sido possível sem o apoio, incentivo e compreensão de pessoas muito especiais que caminharam ao meu lado.

Um agradecimento muito especial ao meu orientador, David Aveiro, pela orientação segura, pelos conselhos no momento certo e pelas oportunidades de crescimento.

À minha mãe, por ser o meu porto seguro e pela sua fé inabalável em mim.

Ao meu namorado, André, pela paciência, pelo suporte nos dias difíceis e pelas palavras de encorajamento que sempre me confortaram.

A todos que, de alguma forma, contribuíram para a realização desta etapa da minha vida, deixo o meu mais sincero agradecimento. Este trabalho é um reflexo da vossa dedicação, carinho e confiança, e sou verdadeiramente grata por tê-los na minha vida.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

API Application Programming Interface

AQL ArangoDB Query Language

BD Base de Dados

CRUD Create, Read, Update, Delete

CSV Comma-Separated Values

DEMO Design and Engineering Methodology for Organisations

DISME Dynamic Information System Modeller and Executer

DSN Data Source Name

EAV Entity-Attribute-Value

ETL Extract, Transform, Load

HTTP Hypertext Transfer Protocol

ISO8601 ISO 8601 Date and Time Format

JDBC Java Database Connectivity

JSON JavaScript Object Notation

MV Materialized View (Vista Materializada)

NoSQL Not Only SQL

OData Open Data Protocol

OQL Object Query Language

OpenAPI OpenAPI Specification

PDF Portable Document Format

RDB Redis Database snapshot persistence

REST Representational State Transfer

SGBD Sistema de Gestão de Bases de Dados

SOAP Simple Object Access Protocol

SQL Structured Query Language

SaaS Software as a Service

TTL Time To Live

UI User Interface

URL Uniform Resource Locator

UTC Coordinated Universal Time

UTF-8 Unicode Transformation Format 8-bit

XML eXtensible Markup Language

XPath XML Path Language

iPaaS Integration Platform as a Service

1 Introdução

Este capítulo apresenta o contexto do projeto, o problema que motivou este trabalho, os objetivos e contributos propostos e a organização do relatório para orientar a leitura.

1.1 Contexto

Num mundo cada vez mais competitivo, existe uma pressão considerável sobre a maioria das organizações para tornar o seu processo operacional, tático e estratégico cada vez mais eficiente e eficaz. Perante esta situação, muitas organizações decidem implementar um sistema informático, para fazer frente à competição, e obter melhor informação para tomar decisões. No entanto, fatores como o conhecimento insuficiente ou inadequado da realidade organizacional a ser automatizada, e a dificuldade dos sistemas de informação e *software* em geral em acompanhar as constantes mudanças de requisitos e/ou de legislação, levam a que esses projetos falhem em cumprir as expectativas [1]. Neste contexto, a disciplina de engenharia organizacional surge com o objetivo de criar modelos que ajudem a compreender a essência completa de uma organização [2].

Entre as metodologias mais conhecidas destaca-se o DEMO (*Design and Engineering Methodology for Organisations*). A sua proposta principal é separar a essência da organização da sua implementação tecnológica. Na prática, isso significa olhar para a rede de compromissos e transações entre atores, sem se perder em detalhes técnicos. Esta forma de pensar garante que os sistemas de informação refletem o que a organização realmente faz, permitindo que as ferramentas evoluam sem comprometer a lógica organizacional. Esta abordagem é particularmente importante em contextos em que se pretende que os sistemas de informação reflitam de forma fiel a dinâmica organizacional, como é o caso do protótipo DISME (do inglês, *Dynamic Information System Modeller and Executer*) que será descrito mais adiante [2].

Além disso, testemunhamos o aumento das plataformas *low-code*. Essas ferramentas permitem a criação rápida de *software* com um esforço mínimo em comparação com a programação tradicional. Esta tecnologia baseia-se em conceitos existentes, incluindo a engenharia baseada em modelos, geração de código e programação visual. Sendo assim a sua adoção traz vantagens como a redução da necessidade de codificação manual, permitindo um desenvolvimento mais rápido, resultando na diminuição dos custos tanto no desenvolvimento como na manutenção. Para além disto, reduz a complexidade, através da utilização de componentes pré-construídos, fazendo com que pessoas não técnicas consigam criar aplicações. Tudo isto, possibilita a transformação digital e o aumento da agilidade das tecnologias e das empresas [3].

E com base nesses conceitos, surgiu o DISME, um protótipo cujo principal objetivo é modelar e parametrizar informação (*workflows*, responsabilidades, interfaces, etc) de forma dinâmica, através de menus e formulários intuitivos de modo a reduzir a necessidade de programação por parte do utilizador [4]. Além disso, também permite modelar de forma dinâmica uma base de dados. Este protótipo está sendo desenvolvido por uma equipa constituída por bolseiros/mestrandos integrados no *Enterprise Engineering Lab* da ARDITI (Agência Regional para o Desenvolvimento da Investigação, Tecnologia e Inovação) [4]. Foi no âmbito deste protótipo que o presente projeto de mestrado foi realizado, mais especificamente na componente de pesquisa dinâmica, que permite a pesquisa e consulta de informação de forma dinâmica a uma base de dados, a partir de uma interface gráfica amigável, sem o utilizador precisar de programar.

Uma componente de pesquisa ou consulta, numa primeira vista, não parece relevante, contudo a inserção e utilização desta ferramenta permite realizar pesquisas *ad hoc*. Estas pesquisas são projetadas para responder a uma pergunta de negócio específica, utilizando uma ou várias fontes de dados da empresa. Por outras palavras, estas pesquisas são realizadas com o intuito de obter informação quando surge a necessidade de analisar dados que não são abrangidos pelos relatórios estáticos e regulares da empresa. Assim sendo, estas pesquisas consistem em SQL construído dinamicamente, que é normalmente desenhado por ferramentas de consulta presentes no *Dashboard*. Desta forma, através da análise dos resultados destas consultas, os utilizadores podem extrair os *insights* de que precisam para tomar melhores decisões de negócio sem depender exclusivamente de equipas técnicas, além de economizar tempo e recursos financeiros [5]. Esta capacidade democratiza o acesso à informação, uma vez que analistas, gestores ou outros colaboradores conseguem obter respostas rápidas às suas questões. Mais do que uma questão de eficiência, trata-se de promover uma cultura organizacional orientada à decisão baseada em dados [6].

1.2 Problema

Apesar do conjunto de funcionalidades já existentes, a componente de pesquisa dinâmica do DISME continuava a mostrar várias limitações. A realização de pesquisas/*queries*, era possível e simples, mas o alcance da ferramenta era reduzido, uma vez que apenas era possível relacionar tabelas diretamente ligadas à tabela base, o que restringia a criação de pesquisas mais complexas e análises cruzadas de dados que, muitas vezes são necessárias em contexto organizacional.

Também não havia mecanismos de configuração visual mais avançada nem formas de guardar e reutilizar pesquisas feitas pelos utilizadores, o que diminuía a flexibilidade e obrigava a repetir trabalho em situações em que seria útil poder retomar ou editar pesquisas anteriores.

Do ponto de vista da integração, a componente de pesquisa dinâmica não oferecia disponibilização de resultados através de REST API, nem suporte a parâmetros dinâmicos, impedindo a sua reutilização e integração com sistemas externos. Além disso, estava restrita a bases de dados relacionais e não incluía suporte para repositórios NoSQL, o que limitava o seu desempenho e escalabilidade.

Assim, mesmo sendo funcional, a versão anterior da pesquisa dinâmica mostrava-se demasiado simples para ambientes organizacionais mais exigentes, pois faltava-lhe a robustez e a abrangência necessárias para garantir flexibilidade, rapidez e integração. Por esta razão, tornou-se essencial repensar esta componente e avançar para uma nova versão mais completa e alinhada com as necessidades reais das organizações.

1.3 Objetivos

O principal objetivo deste projeto foi então corrigir e melhorar as funcionalidades já implementadas na componente de pesquisa dinâmica, tais como guardar e atualizar as pesquisas efetuadas, assim como mostrá-las ao utilizador na UI. Para tal, foi necessário analisar a melhor forma de guardar as *queries* na base de dados, e posteriormente a criação dessas tabelas e respetivas colunas.

O segundo objetivo consistiu na adaptação da componente para operar com repositórios NoSQL, visando armazenar vistas materializadas, a fim de diminuir o elevado tempo de execução dos resultados extraídos do esquema EAV, que apresenta um elevado número de relações entre as tabelas. Desta forma, foram estudadas as melhores alternativas para satisfazer a necessidade de

guardar os dados do dia a dia de uma forma mais eficiente, pois como os dados são guardados apenas na tabela *"value"*, esta pode transformar-se em um ponto de estrangulamento se muita gente estiver a usar o sistema em paralelo. Foram avaliadas alternativas de armazenamento, nomeadamente a distribuição dos resultados por várias tabelas relacionais e a adoção de repositórios NoSQL, para determinar a solução mais eficiente para o DISME.

O terceiro objetivo foi o desenvolvimento de uma interface para disponibilizar os resultados de *queries* por REST API, possibilitando a comunicação desta componente com sistemas fora do DISME e com outras componentes do projeto.

E o último objetivo consistiu no desenvolvimento de funcionalidades que permitiram que as *queries* recebessem parâmetros de forma dinâmica, através da configuração do *endpoint* REST. Com esta melhoria, o utilizador pode personalizar pesquisas em tempo real, introduzindo valores variáveis consoante o cenário de análise, o que acrescenta flexibilidade e aumenta a relevância prática da ferramenta no apoio à decisão.

1.4 Contributos esperados

Com este trabalho, espera-se oferecer uma solução que aumente a usabilidade do DISME, reduzindo a dependência de conhecimentos técnicos especializados. Pretende-se também disponibilizar às organizações uma ferramenta flexível de pesquisa e análise de dados *ad hoc*, contribuindo para a evolução das plataformas *low-code* baseadas em DEMO e explorando a integração entre modelação organizacional e pesquisa dinâmica. Outro contributo esperado passa por demonstrar o impacto das vistas materializadas na eficiência das consultas, garantindo maior rapidez e consistência na apresentação de resultados. Finalmente, espera-se que este trabalho produza conhecimento aplicável tanto no meio académico como no contexto prático das organizações.

1.5 Estrutura do relatório

O relatório está dividido em oito capítulos, começando pela introdução, onde são apresentados o contexto, o problema e os objetivos, bem como os contributos esperados com este projeto.

A introdução teórica é apresentada no segundo capítulo. Este capítulo delimita o âmbito e o foco da introdução teórica, discute a pesquisa dinâmica e os *visual query builders*, aprofunda desafios e a reutilização/integração de pesquisas, examina modelos de dados (EAV, relacional e NoSQL), aborda o papel das vistas materializadas e das interfaces REST.

Seguidamente, inclui-se o estado da arte com um estudo de três sistemas (Mendix, OutSystems e Skyvia) seguido de uma análise comparativa.

No quarto capítulo está apresentado o enquadramento do sistema, sendo descritos o protótipo DISME, a arquitetura global em que a pesquisa dinâmica se insere, a versão anterior desta componente e a modelação de dados que a sustenta, com destaque para o esquema EAV.

No quinto capítulo está apresentada a metodologia e a especificação da solução, sendo que a mesma se divide em requisitos funcionais e não funcionais, a justificação do uso de vistas materializadas e a escolha do MongoDB para ser integrado no projeto e o esquema escolhido para guardar as pesquisas na base de dados.

A implementação está apresentada no sexto capítulo. Os subcapítulos desta secção abordam as tecnologias e ferramentas utilizadas, o desenvolvimento do *backend* em Laravel e do *frontend* em Angular, a publicação por REST e a materialização em MongoDB.

O sétimo capítulo apresenta a avaliação e a discussão do sistema, com testes de integração, medições de latência e testes de usabilidade com SUS, acompanhados da análise dos respetivos resultados.

Por último, tem-se a conclusão deste trabalho, com a síntese dos contributos alcançados e as sugestões de trabalho futuro.

2 Introdução Teórica

A introdução teórica serve dois propósitos neste trabalho. Primeiro, estabelecer a base conceptual que sustenta a proposta de uma pesquisa dinâmica visual que dispensa a escrita de SQL e expõe resultados por REST. E segundo, mapear soluções e escolhas de desenho já testadas por outros autores, para que as decisões tomadas nos capítulos seguintes fiquem justificadas com base em evidência publicada. O capítulo organiza-se em quatro partes que dialogam entre si: começa pela pesquisa dinâmica e pelos *visual query builders*, passa pelos modelos de dados que condicionam a expressividade e o custo das consultas, discute as vistas materializadas enquanto técnica clássica de aceleração de consultas à base de dados e termina com uma descrição de princípios de integração por REST.

Utilizou-se sobretudo o Google Scholar para procurar artigos com combinações de termos como "*visual query builder*", "*dynamic search*", "EAV", "*materialized views*", "NoSQL" e "REST API". Sempre que possível, privilegiaram-se trabalhos publicados a partir de 2021 e aplicaram-se critérios de inclusão focados na relevância direta para a solução do DISME, na existência de avaliação empírica ou de adoção prática e na qualidade editorial. Excluíram-se documentos sem ligação clara ao problema ou de caráter meramente opinativo.

2.1 Âmbito e foco do projeto

Embora o DISME tenha emergido de iniciativas no âmbito da engenharia organizacional, o presente relatório foca-se na componente de pesquisa dinâmica. Sendo assim, a ligação aos conceitos de engenharia organizacional foi abordada apenas na Introdução. As pesquisas visam extrair dados resultantes da atividade organizacional, mas não intervêm na definição de transações, papéis ou regras de ação. Consequentemente, a introdução teórica seguinte privilegia literatura e soluções sobre sistemas *low-code*, *visual query builders* e disponibilização de resultados por REST, bem como nos modelos e técnicas que condicionam o desempenho e a expressividade das consultas (EAV, relacional, NoSQL e vistas materializadas), por serem os elementos que suportam diretamente os objetivos e contribuições deste trabalho.

2.2 Pesquisa dinâmica e *visual query builders*

Neste relatório, utilizam-se os termos "pesquisa dinâmica" e "*visual query builder*" (VQB) para designar conceitos próximos, mas distintos. Esta distinção é importante porque evita ambiguidades e ajuda a separar responsabilidades entre aquilo que o utilizador vê e faz na interface e aquilo que o sistema precisa de garantir no processamento dos dados e na execução das consultas [7]. Para além disto, também usa-se o termo "*query builder*" para designar o componente que constrói a lógica dos filtros (onde se compõem grupos com AND e OR).

O termo "pesquisa dinâmica" refere-se à capacidade de um sistema configurar e executar consultas de forma adaptativa, com parâmetros e com reutilização, sem exigir que se escreva código específico sempre que surge um novo caso, o que inclui definir critérios de filtragem, executar com parâmetros em tempo de execução, guardar e voltar a executar consultas e integrar resultados através de interfaces padronizadas. Na prática, isto permite separar melhor a informação da forma como a mesma é consultada para que mudanças no esquema ou no vocabulário do domínio possam ser efetuadas com pouco esforço e sem depender de programadores [8].

Por sua vez, o termo "*visual query builder*" identifica uma classe de interfaces gráficas que transforma intenções do utilizador em consultas executáveis, substituindo a escrita direta de SQL por uma interação visual em que a pessoa escolhe entidades, combina grupos com conectores lógicos, define propriedades, ordenações e agregações e pode ainda introduzir valores parametrizados para reutilização futura [9]. Isto faz com que a consulta deixe de ser uma instrução momentânea e passe a existir como um artefacto com identidade própria que pode ser guardado, versionado, partilhado e executado em diferentes contextos [7].

2.2.1 Desafios dos VQB

Apesar da promessa de simplicidade, construir um VQB robusto é exigente porque o primeiro grande desafio é a expressividade, sendo então necessário decidir até onde deve ir a liberdade do utilizador, já que selecionar colunas e aplicar filtros simples é fácil de representar, mas quando entram várias junções, subconsultas ou funções analíticas, a interface pode ficar confusa e difícil de interpretar [9].

A literatura recente acrescenta critérios úteis para avaliar a qualidade de um VQB considerado explicável e reprodutível, propondo que a interface funcione com uma gramática visual cujos símbolos e regras correspondem diretamente a operadores relacionais [7], o que permite converter uma consulta visual em SQL de forma fiel e também reconstruir o diagrama a partir do texto quando necessário [9]. É recomendada uma progressão de complexidade que comece por projeções e seleções, introduza junções com metáforas estáveis e disponibilize depois agregações e subconsultas, sempre com pré-visualização do resultado esperado e com a possibilidade de inspecionar e editar o SQL gerado antes da execução [9].

Em relação à usabilidade, há evidência consistente de que o controlo direto com *feedback* imediato melhora a velocidade e a compreensão das tarefas. Cada ajuste nos filtros deve produzir um resultado logo de imediato, para criar uma sensação de causalidade e de controlo que reduz erros e aumenta a satisfação. Quando a execução completa é custosa, mostrar resultados parciais e ir atualizando aos poucos ajuda a manter a fluidez da interação [8].

2.2.2 Reutilização e integração das pesquisas

Um ponto essencial é a reutilização, pois uma pesquisa isolada tem pouco valor se não puder ser reaproveitada e ajustada. As ferramentas mais consolidadas tratam as consultas como artefactos persistentes que podem ser guardados, reexecutados e parametrizados para diferentes contextos, o que favorece a colaboração entre equipas, reduz redundância e permite que o mesmo modelo de pesquisa seja aplicado repetidamente alterando apenas elementos como o intervalo temporal, o departamento ou a região [7].

A integração também é determinante porque as consultas não devem terminar na interface e o verdadeiro valor da pesquisa dinâmica aparece quando os resultados podem ser partilhados com outros serviços, exportados para formatos adequados e usados em relatórios ou painéis externos [8]. Isto requer interfaces estáveis e previsíveis onde cada pesquisa é tratada como um recurso acessível por um identificador estável e executável com operações normalizadas [7]. Esta visão está de acordo com abordagens em que as consultas geradas pelos utilizadores são expostas e versionadas como recursos de API para facilitar automação e interoperabilidade ao longo do ciclo de vida da informação [8].

2.3 Modelos de dados para pesquisa

Plataformas *low-code* e sistemas dinâmicos como o DISME tornam possível criar aplicações de forma rápida sem exigir programação intensiva. Nestes contextos, a forma como os dados são guardados determina quanto os utilizadores conseguem fazer pesquisas de modo eficaz e com rapidez. Neste capítulo abordam-se três modelos de dados principais: o modelo EAV, o modelo relacional tradicional e as bases de dados NoSQL.

Sendo assim, neste capítulo revê-se a literatura mais recente sobre estas abordagens com atenção ao seu uso em plataformas *low-code* e em sistemas semelhantes ao DISME e avaliam-se as vantagens e limitações de cada modelo em termos de desempenho, flexibilidade, escalabilidade e facilidade de integração.

2.3.1 Modelo EAV (Entidade-Atributo-Valor)

O modelo Entidade-Atributo-Valor organiza dados de forma "vertical", armazenando cada valor de atributo como uma linha separada, em vez de colunas fixas [10]. Em vez de uma tabela com muitas colunas (onde a maioria poderia ficar vazia), o EAV usa tipicamente três componentes: uma tabela de entidades (registos principais), uma tabela de atributos (definições de campos possíveis) e uma tabela de valores que liga cada entidade a um atributo e ao seu valor correspondente [11], como é mostrado na Figura 1. Assim, novas propriedades podem ser adicionadas criando novas linhas em vez de alterar a estrutura da base de dados. Esta abordagem é comum em sistemas onde diferentes registos podem ter conjuntos de atributos muito variados ou em constante mudança (por exemplo, registos clínicos ou formulários personalizáveis) [10].

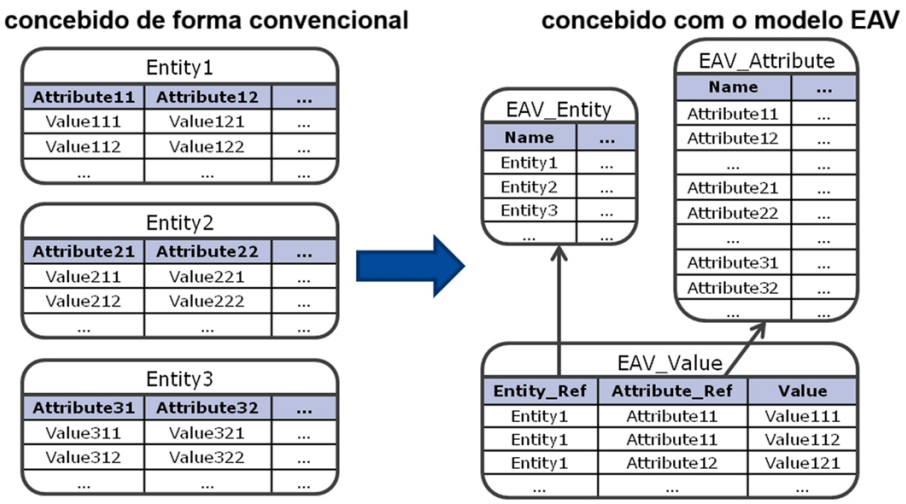


Fig. 1: Comparação entre o desenho convencional e o modelo EAV. À esquerda, cada entidade tem a sua tabela com colunas fixas. À direita, o EAV concentra os dados em "*EAV_Entity*" e "*EAV_Attribute*" e guarda os valores "na vertical" em "*EAV_Value*" [12].

Sendo assim, a adoção do modelo EAV apresenta várias vantagens:

– Em relação à flexibilidade de esquema, o EAV permite introduzir novos atributos sem migrações dispendiosas. Esta opção reduz o risco de interrupções e apoia ciclos de desenvolvimento rápidos, porque o modelo cresce com os dados e não com alterações estruturais [11].

- Sobre dados esparsos, a vantagem é direta, já que apenas se registam os valores que existem. Evitam-se colunas vazias e diminui-se o desperdício de espaço, o que tende a manter a base de dados mais enxuta e organizada [11].

- No que toca à adaptação em interfaces visuais, a presença de uma tabela de atributos facilita a construção de pesquisas dinâmicas. Menus e filtros podem ser preenchidos com base nos atributos em uso, sem reconfigurar a interface sempre que surge um novo campo [11].

- Em plataformas *low-code*, esta organização promove a unificação do modelo interno, permitindo que o utilizador crie quantas "tabelas" e "colunas" quiser na interface, enquanto o sistema mantém o trio entidade, atributo e valor. Um exemplo é o Workato Table Storage[1], apresentado em 2023 como repositório de dados *low-code* baseado em EAV, onde o esquema interno permanece fixo apesar das definições feitas pelo utilizador. Esta uniformidade simplifica a persistência, dispensa conhecimentos de SGBD ou ajustes manuais de índices e sustenta um desempenho estável mesmo com milhões de registos.

- Por fim, quanto à execução imediata de alterações, o EAV encaixa bem em sistemas que seguem o padrão *Adaptive Object Model*, como o DISME. Novos tipos de informação tornam-se operacionais assim que são definidos e as mudanças refletem-se no sistema em execução sem depender de migrações, o que favorece a experimentação e a melhoria contínua [13].

Ainda assim, o EAV traz desvantagens que importa reconhecer antes de o escolher como modelo principal:

- Em consultas o EAV tende a ser mais pesado, porque para recuperar todos os dados de uma entidade é habitual encadear várias junções entre as tabelas de entidades, atributos e valores e, em muitos casos, aplicar agregações para transformar linhas em colunas, o que aumenta a latência quando o volume de dados cresce e torna esta opção menos adequada em cenários onde a rapidez de acesso é crítica [11].

- Na integridade e na validação de tipos, a base de dados oferece menos proteção, pois num esquema relacional cada coluna tem tipo definido e regras claras como `NOT NULL`, `FOREIGN KEY` e `CHECK`, enquanto no EAV os valores são muitas vezes registados de forma genérica ou distribuídos por tipo em tabelas paralelas. Garantir que um campo como "Data de nascimento" é sempre uma data válida passa para a aplicação ou para *triggers* e verificações mais complexas, o que aumenta a probabilidade de inconsistências e desloca a responsabilidade de consistência do SGBD para o código [10].

- No desenvolvimento e na manutenção a complexidade aumenta, porque trabalhar sobre EAV obriga a manipular metadados além dos dados, a mapear atributos, a agregar e a transpor informação e a gerir versões e mudanças de significado ao longo do tempo. A depuração também se torna menos direta, já que a estrutura esperada não está explícita nas tabelas e é preciso recorrer a camadas de mapeamento e transformação na aplicação, o que exige mais esforço e competências da equipa técnica [11].

- No desempenho e na escalabilidade os resultados são irregulares, porque inserir uma entidade com muitos campos implica várias inserções na tabela de valores em vez de uma única linha e, embora índices no par atributo–valor acelerem algumas pesquisas, outras operações como

[1] Workato Table Storage – `https://www.workato.com/product-hub/low-code-data-storage-for-business-applications/`

comparações numéricas e agregações continuam caras. Em muitos cenários soluções com documentos JSON bem indexados ocupam menos espaço e respondem mais depressa, o que mostra que sem otimização e indexação cuidadas o EAV pode sofrer ao trabalhar com grandes volumes de dados [11].

– Na integração com ferramentas externas, o caminho é menos direto, já que muitas soluções de BI e de relatórios esperam esquemas com colunas fixas e, quando se expõe um EAV tal como está, surgem tabelas genéricas de atributo–valor pouco úteis para quem constrói relatórios *ad hoc* em SQL ou precisa de combinar fontes. Torna-se comum criar vistas derivadas ou materializadas e, nalguns casos, sincronizar para um esquema relacional à parte para fins de relatório, o que aumenta a dependência de APIs e de mecanismos específicos da plataforma para obter dados em formatos prontos a usar [14].

2.3.2 Modelo relacional tradicional

O modelo relacional é o modelo clássico de armazenamento de dados, sendo dominante há muitos anos, uma vez que é sólido tanto na teoria como na prática. A informação fica em tabelas com colunas definidas e ligadas entre si por chaves, onde cada tabela representa uma entidade e cada coluna representa um atributo dessa entidade [15].

Nas plataformas *low-code* o mais comum é usar bases de dados relacionais. Ferramentas como Mendix, OutSystems e Bizagi trazem um *visual query builder* que deixa escolher entidades, campos e ligações sem escrever SQL. No ecrã, desenha-se o esquema e a plataforma trata do resto, gera as consultas e guarda os dados no motor relacional.

A seguir resumem-se os pontos que fazem o modelo relacional funcionar bem com *visual query builders* e com plataformas *low-code* que precisam de regras claras, respostas rápidas e integração simples com o resto do sistema:

– Em relação à integridade e consistência dos dados, os SGBD relacionais oferecem garantias ACID (Atomicidade, Consistência, Isolamento e Durabilidade) para cada transação e reduzem o risco de corrupção em operações concorrentes ou em falhas. Graças a esquemas explícitos é possível definir tipos rigorosos e restrições como chaves estrangeiras e unicidade. Se um campo é do tipo `DATE` o SGBD recusa valores inválidos, algo que em EAV ou em algumas abordagens NoSQL tende a ficar a cargo da aplicação [15].

– As bases de dados relacionais estão muito otimizadas para SQL quando o esquema é conhecido. É simples criar índices em colunas muito usadas e as estatísticas do SGBD ajudam o otimizador a escolher planos eficientes, enquanto junções e agregações complexas são suportadas de forma nativa [15].

– As bases de dados relacionais suportam SQL e conectores ODBC/JDBC, o que permite que ferramentas de BI consigam ler dados de plataformas *low-code* para fazer relatórios, migrações e cruzar com outras fontes sem grandes transformações. Nas empresas esta compatibilidade poupa tempo e reduz risco [15].

– A maioria das equipas de TI está habituada a modelos relacionais e a SQL. Mesmo quando a plataforma é *low-code*, é mais fácil olhar para tabelas e colunas conhecidas e aplicar regras que já dominam, como normalizar dados e definir chaves bem claras [15].

– Em termos de adequação a *visual query builders*, um esquema conhecido permite listar entidades, campos e relações de forma intuitiva. O resultado é uma experiência direta e previsível, ainda que sem adicionar colunas em tempo de execução [15].

Apesar de ser o modelo de dados mais popular e de ter inúmeras vantagens, ao sair do cenário de um esquema conhecido e estável, surgem custos e compromissos a serem considerados. Abaixo estão os principais limites que devem ser levados em conta para decidir se o modelo relacional ainda é adequado ou se é mais vantajoso combiná-lo com padrões como EAV ou colunas JSON:

– Em relação à evolução de requisitos, existe rigidez porque o esquema tem de estar definido antes e cada novo campo obriga a migrações com alterações a tabelas e colunas. Em plataformas *low-code* isso implica voltar ao editor, ajustar a entidade e publicar de novo a aplicação, pelo que em cenários com campos definidos pelo utilizador é comum recorrer a padrões EAV ou a colunas `JSON` para contornar a limitação [15].

– No que toca a dados esparsos, muitos atributos opcionais levam à criação de colunas que ficam vazias na maior parte dos registos, aumentando o espaço usado [15].

– Os sistemas relacionais tendem a escalar melhor de forma vertical com máquinas maiores e configurações cuidadas de replicação. Existem soluções com *sharding* e SQL distribuído, mas exigem desenho e operação mais complexos para manter consistência e desempenho, e em muitas aplicações *low-code* um único SGBD por aplicação obriga a afinação profunda ou a *hardware* mais potente para suportar grandes volumes de dados [16].

– Se os dados têm natureza hierárquica ou variam em estrutura (por exemplo, um campo que às vezes é um único valor e outras vezes uma lista de valores ou sub-objetos), o modelo relacional exige modelar isso explicitamente. Assim, para certos tipos de informação complexa, o modelo relacional pode ser menos natural e exigir mais junções e tabelas para representar o mesmo conteúdo [15].

2.3.3 Modelo NoSQL

As bases de dados relacionais, apesar de serem poderosas, flexíveis e as soluções mais conhecidas e utilizadas, têm várias questões ou características que nunca foram ultrapassadas ou fornecidas. Consequentemente, recentemente, uma série de sistemas chamados NoSQL (não apenas SQL) ganhou imediatamente popularidade, com o objetivo de ultrapassar algumas das barreiras impostas pelo modelo relacional [16].

Sendo assim, as bases de dados NoSQL formam uma família de sistemas que colocam de lado o modelo relacional para ganhar flexibilidade de esquema e escalar na horizontal. Há vários tipos e cada um resolve problemas diferentes [17]:

– **Documentos:** Cada registo é um documento em JSON ou formato semelhante, com campos que podem variar entre documentos e incluir coleções e objetos internos, o que dá flexibilidade sem migrações de esquema e aproxima os dados do que a aplicação e as APIs trocam. Um exemplo típico é o MongoDB.

– **Colunas (*wide-column*):** Os dados organizam-se em famílias de colunas e cada linha guarda apenas as colunas de que precisa, ficando o armazenamento naturalmente esparso. O acesso é pensado para leituras por chave e por intervalos ordenados, o que escala muito bem em séries temporais e registos de eventos. Exemplos: Apache Cassandra, HBase, Bigtable.

- **Chave-valor:** Cada entrada associa uma chave única a um valor e as operações centram-se em ler e escrever por essa chave com latências muito baixas. Isto torna este modelo ideal para *cache*, gestão de sessões e contadores, desde que a aplicação conheça sempre a chave e não precise de procurar por atributos internos ou fazer consultas complexas. Exemplo: Redis.
- **Grafos:** As entidades são nós e as relações entre elas são arestas com propriedades. O valor do modelo surge da capacidade de percorrer cadeias de ligações com grande eficiência para responder a perguntas sobre vizinhanças, caminhos e padrões de ligação, o que é particularmente útil em recomendações, redes sociais e deteção de fraude, onde a relação entre elementos é o centro do problema. Exemplo: Neo4j.

No contexto de aplicações dinâmicas e *low-code*, o mais usado é o modelo orientado a documentos em JSON ou formato semelhante porque permite guardar registos com estruturas diferentes e campos aninhados [15]. Na prática, é uma alternativa simples ao EAV para dados semiestruturados. Várias plataformas *low-code* já incorporam NoSQL internamente ou por integração, como por exemplo o sistema Budibase[2], que usa CouchDB como repositório das aplicações e tira partido do armazenamento orientado a documentos.

A seguir apresentam-se as principais vantagens do paradigma NoSQL no contexto em análise:

- A estrutura deste tipo de bases de dados é flexível, já que permitem registos com campos diferentes no mesmo conjunto de dados. É possível acrescentar novos atributos quando necessário sem migrar o esquema e sem reconstruir a entidade a partir de várias tabelas, o que acelera a evolução do modelo [15].
- Muitos NoSQL escalam na horizontal em *clusters* onde os dados se distribuem por vários servidores com *sharding* e a disponibilidade é assegurada por réplicas. Esta arquitetura permite crescer de forma quase linear ao adicionar nós para guardar ou processar mais informação e numa plataforma *low-code* pode significar suportar mais utilizadores e mais dados sem migrações para motores maiores ou reconfigurações complexas [16].
- Quando o modelo permite guardar informação relacionada no mesmo registo, evitam-se *joins* frequentes e as consultas visuais ficam mais simples, porque um único registo já traz os campos necessários, reduzindo a latência [15].
- Em modelos com objetos ou JSON, acrescentam-se campos opcionais diretamente no registo e criam-se índices nos respetivos caminhos. Isto facilita filtros sobre atributos que nem todos os registos possuem e pode acelerar respostas quando comparado com EAV, que dispersa atributos por várias linhas e exige mais junções [15].
- O NoSQL destaca-se em aplicações *web* que guardam grandes volumes de dados heterogéneos como *logs*, telemetria e dados de sensores, e é útil quando a consistência pode ser flexibilizada em favor de desempenho e disponibilidade, de acordo com o teorema CAP [16].

E as desvantagens:

- Muitos sistemas NoSQL sacrificam consistência imediata para ganhar escalabilidade e disponibilidade segundo o CAP (Consistência, Disponibilidade e Tolerância a Partições), escolhendo tolerância a partições e disponibilidade em detrimento de consistência forte. Sem cuidado, a

[2]Budibase – `https://budibase.com/`

aplicação pode ler dados desatualizados ou apenas parcialmente replicados e em contexto *low-code*, onde se espera que a plataforma esconda a complexidade, lidar com consistência eventual pode ser desafiante e gerar comportamentos inesperados para o utilizador final [16].

- Ao contrário do SQL, cada NoSQL tem a sua API ou linguagem como MongoDB com consultas em JSON e Cassandra com CQL, o que exige abstrações num *visual query builder*. Muitas plataformas *low-code* optam por não expor toda essa variedade ao utilizador e limitam as operações pela interface, como permitir filtros e ordenações em MongoDB mas não combinações equivalentes a *joins* complexos entre coleções. Isto significa que certas consultas avançadas podem não ser possíveis sem escrever código ou *scripts*, ou seja, enquanto em SQL o utilizador avançado pode sempre escrever uma *query* manual, em NoSQL essa flexibilidade pode não existir. No *low-code* os designers precisam antecipar operações comuns e implementá-las e há casos como o Workato cujo Table Storage baseado em EAV e NoSQL não permite SQL arbitrário e oferece filtros pré-definidos otimizados, assumindo um compromisso na expressividade para ganhar usabilidade e desempenho [15].
- Existem menos conectores prontos para NoSQL do que para SGBD relacionais e gerar relatórios de BI sobre uma base de dados deste tipo pode exigir ferramentas compatíveis ou a exportação dos dados para formato relacional. Muitos ambientes acabam por montar *pipelines* ETL para mover dados NoSQL para armazenamentos analíticos relacionais e assim usar SQL e ferramentas convencionais, adicionando complexidade. Numa plataforma *low-code* expor dados NoSQL a outros sistemas pode exigir criar APIs específicas ou conectores personalizados e com um SGBD relacional esse processo tende a ser mais transparente, pois mesmo havendo *drivers* para muitos NoSQL, a curva de adoção e o esforço de integração são maiores [15].
- Em muitos sistemas NoSQL não há *joins* como num SGBD relacional ou existem de forma limitada. Quando é necessário cruzar informação de coleções ou partições diferentes existem duas opções principais: duplicar dados para evitar junções e arriscar inconsistências e mais espaço ou fazer várias leituras e juntar resultados na aplicação com mais latência e mais trabalho. Em *low-code* isto traduz-se em fluxos ou regras adicionais para manter coerência e agregar dados manualmente. Algumas ferramentas tentam mitigar com operações de *lookup* ou *join-like* limitadas como no *aggregation pipeline* do MongoDB, mas não é tão simples como usar SQL [15].
- E por fim, apesar de existirem motores NoSQL consolidados, o suporte de longo prazo e a disponibilidade de talento experiente são em geral menores do que no ecossistema relacional. Muitas empresas dispõem de administradores de bases de dados com forte experiência em SQL mas podem não ter competências para otimizar Cassandra ou depurar replicação em MongoDB, e em *low-code* espera-se que a plataforma esconda estes detalhes. Cada NoSQL tem ainda particularidades que a plataforma precisa tratar como limites de tamanho de documento no MongoDB ou escolhas de consistência de leitura no Cassandra, em contraste com PostgreSQL ou SQL Server que são terreno mais conhecido e melhor documentado [16].

2.4 Vistas materializadas

Para além da pesquisa e exploração de dados através de construtores visuais e da sua disponibilização por interfaces REST, existe uma técnica clássica para melhorar o desempenho de pesquisas: as vistas materializadas. Ao contrário das vistas convencionais, que são consultadas sempre que

a pesquisa é executada, uma vista materializada guarda o resultado da pesquisa e permite a sua reutilização em execuções posteriores. Esta técnica reduz a latência em cenários com operações com junções complexas ou agregações sobre grandes volumes de dados [18].

O interesse nas vistas materializadas não está apenas na rapidez, mas também na previsibilidade. Como o resultado está pré-calculado, o tempo de resposta depende menos da complexidade da pesquisa original e mais do tamanho da vista e da eficiência do acesso a esses dados. Em contextos organizacionais, onde relatórios e análises precisam de ser produzidos de forma recorrente, esta previsibilidade torna-se um fator decisivo [19].

Tendo isto presente, a plataforma Databricks mostra que, em *dashboards* analíticos, usar vistas materializadas pode tornar as consultas até 98% mais baratas e 85% mais rápidas do que voltar a construir a *cache* do zero, mantendo a informação atual o suficiente para uso diário [20]. Nas boas práticas da Microsoft para NoSQL, as recomendações apontam para materializar apenas o que é lido com frequência e para manter as vistas atualizadas por agenda ou em resposta a alterações, equilibrando consistência, custo e espaço [21]. A Figura 2 ajuda a visualizar a ideia, os dados operacionais continuam a ser a fonte de verdade e, ao lado, existe uma vista materializada de leitura que já guarda o total por artigo, o esforço faz-se na atualização e a aplicação lê mais depressa e com tempos de resposta mais estáveis.

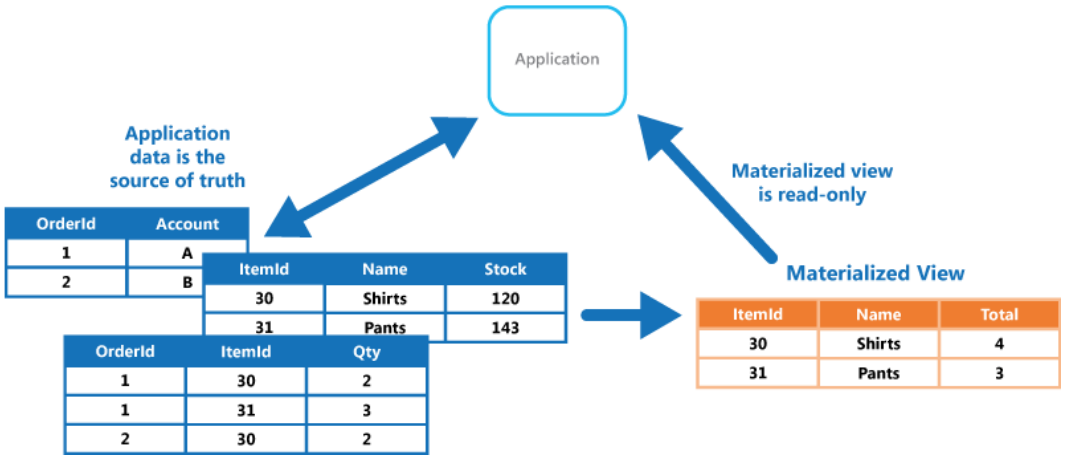


Fig. 2: Exemplo de criação de uma vista materializada [21].

O desafio principal prende-se com a atualização da informação, pois sempre que os dados de origem são atualizados, a vista pode ficar desatualizada. Para lidar com este problema, têm sido estudadas várias políticas de manutenção [21]:

- Imediata: em que a vista é atualizada a cada modificação.
- Diferida: em que a atualização da vista ocorre apenas quando é necessário, evitando gastos desnecessários quando a vista é pouco usada.
- Periódica: em que se segue uma agenda pré-definida, oferecendo assim um equilíbrio entre custo e previsibilidade.

Em bases de dados modernas, estas políticas podem ser combinadas com estratégias incrementais de manutenção, em que apenas as diferenças são propagadas, reduzindo custos de processamento [18, 19].

Para este projeto interessam apenas três formas de materializar resultados que ajudam a reduzir a latência e a estabilizar tempos de resposta:

- A primeira opção é guardar coleções de resultados por consulta, isto é, manter em *cache* duradoura o resultado de pesquisas pesadas para reutilização ao longo da sessão ou em relatórios recorrentes. É normalmente a opção mais simples de implementar e permite criar índices alinhados com o padrão real de leitura, o que acelera repetições e variações próximas. Contudo, exige uma boa gestão para evitar que surjam muitas variações da mesma pesquisa, principalmente em cenários com muitos parâmetros [22].
- A segunda opção é pré-computar agregações por período ou por chave de negócio, como somas e contagens diárias, mensais ou por entidade com um nível de detalhe bem definido. Desta forma, a leitura incide sobre estruturas compactas e o custo de cálculo concentra-se nos momentos de atualização, o que facilita agendar as atualizações e dimensionar os recursos. A desvantagem é a menor flexibilidade fora desse nível de detalhe, por isso perguntas com outro recorte temporal ou outra unidade de análise obrigam a voltar a calcular as métricas [22].
- E por fim, a terceira opção é usar modelos de leitura desnormalizados por entidade. São criadas estruturas de leitura que juntam, numa só representação, os campos e as relações mais usados, evitando reconstruções EAV no momento da consulta, o que reduz junções, simplifica o código da aplicação e traz respostas mais rápidas e estáveis sob carga. O custo é a duplicação de dados e a necessidade de processos de atualização fiáveis e visíveis, para que qualquer mudança na origem seja propagada sem divergências [21].

A literatura é clara sobre o momento certo para usar e para evitar vistas materializadas. Vale a pena usar quando a mesma consulta se repete muitas vezes e segue um padrão estável, quando há junções e agregações pesadas que atrasam a resposta, quando a organização precisa de tempos previsíveis e aceita um pequeno atraso na atualização dos dados [23]. Deve evitar-se quando os dados mudam muito depressa e é necessária consistência imediata, quando o esforço de manter a vista atualizada é maior do que o ganho de desempenho, ou quando o acesso é raro e imprevisível. Nestas situações compensa mais ler diretamente da fonte ou usar uma *cache* simples com expiração curta, porque exige menos esforço e responde de forma adequada ao que é pretendido [21].

2.5 Interfaces REST

Construir uma pesquisa de forma visual é apenas parte do processo. Em muitos casos, os resultados não interessam apenas a quem está diante do ecrã, mas precisam de ser reutilizados noutros contextos como: integrar relatórios periódicos, fornecer dados a aplicações de monitorização ou por quaisquer sistemas externos. É aqui que entram as interfaces REST, hoje a abordagem mais comum para expor informação de forma estável e previsível.

A designação REST, acrónimo de *Representational State Transfer*, foi definida por [24] como um estilo arquitetónico para serviços na *web*. A sua popularidade resulta da simplicidade: cada recurso de informação corresponde a um URI, tal como uma página tem o seu endereço. Ao aceder a esse URI, interage-se com o recurso através de operações uniformes do protocolo HTTP. Em cenários de leitura destaca-se o método GET, que devolve uma representação do recurso e é considerado

seguro por não alterar o estado do servidor. Inclui ainda POST para criar novos recursos, PUT e PATCH para atualizar informação existente, e DELETE para remover um recurso identificado [25]. No âmbito deste projeto, o foco recai no GET, por ser suficiente para assegurar o acesso de forma segura aos resultados das pesquisas.

Para tornar esta ideia concreta, a Figura 3 mostra como um URI típico se decompõe quando são usados parâmetros de consulta [26]:

- **Base do URL:** é a parte estável que identifica o recurso sem parâmetros e junta o esquema (`https`), a autoridade (`domain.com`) e o caminho dentro do serviço (`/product-page`). É nesta parte que costuma estar a versão da API e é isto que deve permanecer igual quando apenas são alterados filtros, ordenações ou paginação.
- **Início dos parâmetros:** começa no carácter `?` e prolonga-se até a um eventual `#`. Aplicam-se as regras de codificação próprias de URI, por isso qualquer carácter fora do conjunto permitido deve ser codificado e a ordem dos pares chave=valor raramente tem significado, embora cada par deva estar completo para ser interpretado corretamente.
- **Chave:** é o nome do parâmetro definido pelo contrato da API, como `color` ou `limit`. É recomendado manter nomes estáveis e previsíveis para que clientes diferentes consigam construir pedidos equivalentes sem depender de detalhes internos do servidor.
- **Valor:** é o conteúdo associado à chave e liga-se com o sinal de igual `=`, como em `color=yellow`. Os valores são *strings* codificadas mas podem representar números (`size=8`), datas em ISO 8601 (`created_at=2025-04-17T16:00:00Z`) ou booleanos (`active=true`), e quando é necessário enviar vários valores para a mesma chave (`color=yellow&color=blue`) ou usar listas acordadas pela API (`color=yellow,blue`).
- **Separador `&`:** une vários pares chave=valor dentro da mesma cadeia de consulta, como em `?color=yellow&size=8`. Se surgir um `&` dentro de um valor ele deve ser codificado para não ser confundido com um separador.

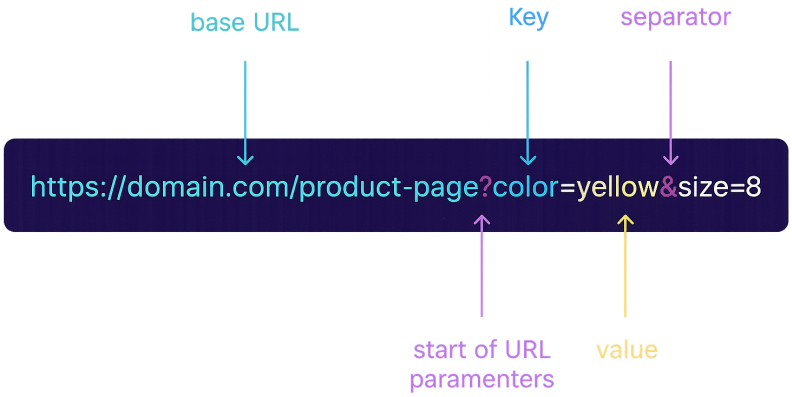


Fig. 3: Partes principais de um URI com parâmetros de consulta [26].

A previsibilidade das interfaces REST estende-se também aos erros e ao comportamento esperado. As respostas seguem códigos padrão: 200 significa sucesso, 404 indica que o recurso não foi encontrado, 500 aponta para uma falha no servidor, entre outros. Em cenários mais exigentes, a

resposta pode vir estruturada em JSON com detalhes sobre o problema, facilitando o diagnóstico sem expor dados sensíveis [27]. Outro aspeto essencial é a possibilidade de evolução controlada através de versões explícitas, que evita ruturas em clientes que dependem da API [28].

A Figura 4 ilustra de forma simplificada o funcionamento de uma API REST, mostrando como um cliente envia um pedido HTTP para um recurso identificado por URI e recebe como resposta uma representação desse recurso em formato JSON.

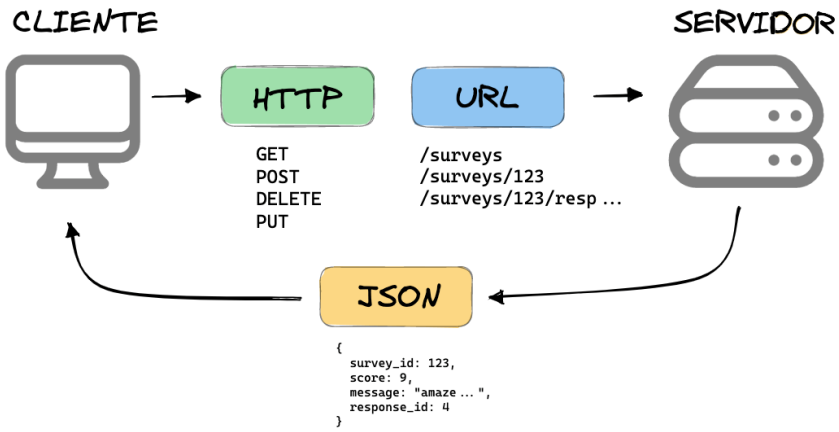


Fig. 4: Funcionamento simplificado de uma API REST [29].

A arquitetura REST também tira partido dos mecanismos de *cache* já existentes na *web*. Cabeçalhos como *Cache-Control* permitem que respostas sejam reutilizadas durante um período definido, e validadores como ETag ajudam a verificar se a informação mudou sem precisar de transferir tudo de novo. Em consultas repetidas, estas estratégias reduzem tempo de espera e custos de rede, sem comprometer a atualidade dos dados [30].

No contexto da pesquisa e exploração de dados, esta abordagem torna-se particularmente relevante. Se os VQB democratizam o acesso à informação dentro da organização, as interfaces REST asseguram que esse mesmo conhecimento pode ser partilhado de forma controlada com outros sistemas. Assim, a informação não fica confinada ao momento em que é consultada, mas passa a estar disponível para ser integrada em processos automáticos, relatórios ou aplicações externas.

3 Estado da Arte

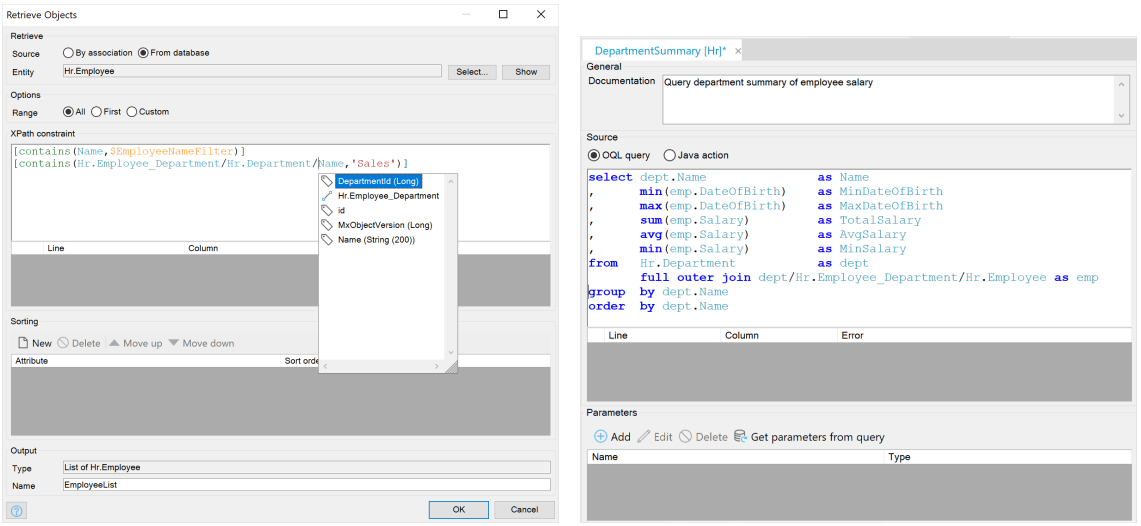
Nesta secção são analisados sistemas relevantes que, tal como o DISME, procuram integrar funcionalidades de pesquisa dinâmica em bases de dados através de abordagens visuais. Foram selecionados três sistemas para análise comparativa: Mendix, OutSystems e Skyvia.

Para a recolha de informação sobre os sistemas, recorreu-se à documentação oficial e artigos técnicos [31–33], permitindo obter uma caracterização completa destas soluções e servir de base à análise comparativa que se segue. Para além da documentação, também explorou-se diretamente as versões disponibilizadas gratuitamente dos três sistemas, sendo possível testar algumas funcionalidades, nomeadamente a criação visual de pesquisas e a sua execução sobre diferentes fontes de dados.

3.1 Mendix

O Mendix é uma plataforma *low-code* bastante conhecida, usada para criar aplicações empresariais de forma rápida. Funciona na *cloud* e procura apoiar todo o ciclo de vida do *software*, desde a ideia inicial até à execução. Para isso, combina ferramentas de integração, definição de fluxos de trabalho e acesso a dados [31].

Na parte de pesquisas, o Mendix oferece duas formas principais de consultar informação. A primeira é o *XPath*, que permite procurar dados no modelo de informação definido na aplicação. Apesar de mais simples de aprender, o XPath é bastante expressivo, pois permite aplicar filtros por atributos, navegar por associações entre entidades, ordenar resultados e até utilizar funções e parâmetros definidos pelo utilizador. A segunda forma é a OQL (*Object Query Language*), uma linguagem semelhante a SQL mas adaptada ao modelo do Mendix. O OQL é indicado para consultas mais complexas, como cruzar várias entidades, aplicar agregações ou calcular estatísticas [34].



a) b)

Fig. 5: Interface do Mendix: a) filtro XPath parametrizado em *"Hr.Employee"* [34]; b) consulta OQL com agregações por departamento [34].

Os exemplos oficiais do *Mendix Evaluation Guide* ajudam a ver a diferença entre as duas abordagens. Na Figura 5a está um caso em XPath sobre a entidade *"Hr.Employee"*, onde a pesquisa é filtrada pelo nome do departamento ("Sales"). A configuração faz-se de forma visual no Studio Pro e o resultado devolve apenas os registos que cumprem as condições [34]. Já a Figura 5b mostra um exemplo em OQL pensado para análise, onde a consulta resume informação por departamento a partir das entidades *"Hr.Department"* e *"Hr.Employee"*. Calcula mínimos e máximos de datas de nascimento, somas e médias de salários e ordena os resultados por nome do departamento. Aqui vê-se a vantagem do *OQL* para relatórios com junções e agregações, visto que, a linguagem é próxima do SQL e oferece mais expressividade para sínteses e métricas do que o *XPath* [34].

As consultas, tanto em XPath como em OQL, podem ser guardadas e reutilizadas em diferentes partes da aplicação [34].

No que toca à integração, o Mendix facilita a partilha de dados com sistemas externos. Permite expor pesquisas como serviços REST (gerando automaticamente documentação no formato OpenAPI) ou publicar serviços OData para filtrar e ordenar resultados. Isto garante que os dados não ficam "presos" dentro da plataforma e podem ser consultados de forma previsível por outras aplicações [35].

Apesar da robustez, há limitações relevantes. A plataforma é apontada como cara e com política de preços pouco transparente, o que restringe o seu acesso a contextos empresariais com maior capacidade de investimento [36]. Além disso, do ponto de vista da pesquisa dinâmica, embora ofereça elevada expressividade, essa vantagem depende do domínio de linguagens técnicas, o que contrasta com abordagens que procuram simplificar ao máximo a experiência do utilizador.

3.2 OutSystems

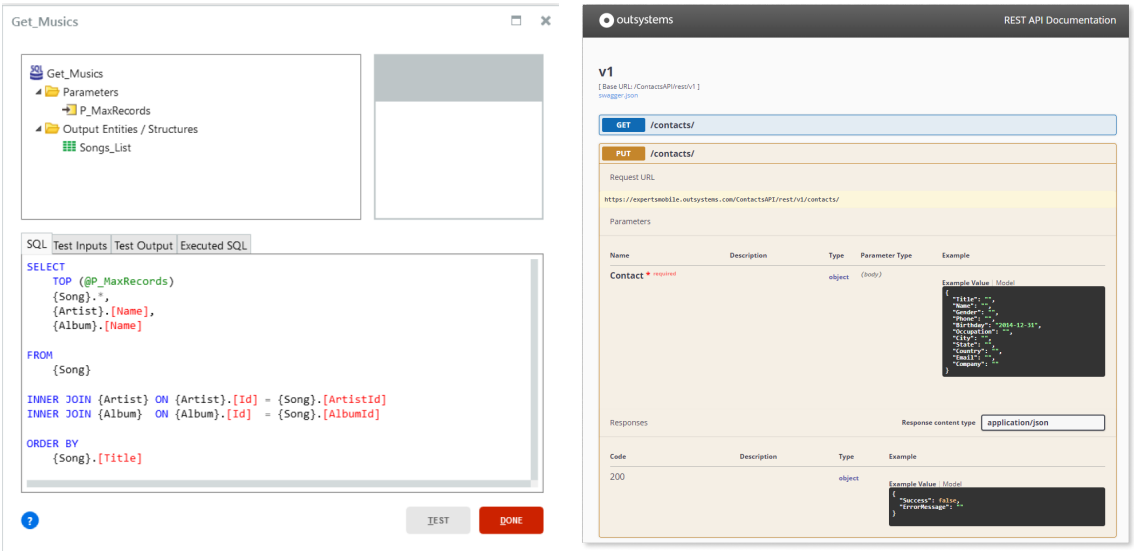
O OutSystems é uma plataforma *low-code* para desenvolver aplicações *web* e móveis em contexto empresarial. Corre na *cloud* e cobre o ciclo completo de desenvolvimento, desde a modelação de dados até à publicação e monitorização. Oferece construção visual por *drag and drop*, testes automatizados, gestão de utilizadores e integra-se com vários SGBD [32].

Na consulta de dados, a parte principal é o editor visual *Aggregate* onde se escolhem entidades, definem-se *joins* pelas relações do modelo, aplicam-se filtros e ordenações e, quando necessário, agregações, sem escrever SQL. Os filtros aceitam parâmetros vindos do ecrã ou de ações lógicas. O sistema gera SQL otimizado para a base de dados configurada e respeita as permissões definidas nas entidades e nos papéis de utilizador [37].

Quando o caso de uso pede mais do que o editor visual consegue exprimir (por exemplo subconsultas, junções complexas ou transformações específicas), a plataforma recorre a *Advanced SQL*, onde é possível escrever a instrução diretamente, com acesso a parâmetros e entidades do modelo [38].

No exemplo da Figura 6a vê-se o editor visual com uma pesquisa com dois *joins*, dois filtros parametrizados e a propriedade *Max Records* definida [39].

As consultas criadas com *Aggregates* ou *Advanced SQL* podem ser encapsuladas em ações do servidor e chamadas em vários ecrãs ou fluxos, com os mesmos parâmetros [37].



a) b)

Fig. 6: Interface do OutSystems: a) *Aggregate* com *joins* e filtros parametrizados [39]; b) publicação de *endpoint* REST com documentação gerada [40].

Para integração, a plataforma expõe dados por REST de forma nativa, onde define-se a ação que produz a resposta, configuram-se *inputs* e *outputs* e o OutSystems trata da publicação do *endpoint*, da documentação OpenAPI (Figura 6b) e da segurança ao nível do módulo e dos papéis [41]. O OutSystems integra com SGBD relacionais como SQL Server, Oracle, MySQL e PostgreSQL, incluindo suporte para MongoDB, embora com configuração extra e menor flexibilidade face às bases de dados relacionais [32].

3.3 Skyvia

O Skyvia é um serviço *cloud* orientado à integração e ao acesso a dados. Para este projeto apenas interessam as componentes *Query* e *Connect* [33].

No *Query*, o *visual query builder* organiza a configuração em painéis de *Result Fields*, *Filters*, *Sort Fields* e *Details* [42], como é possível ver na Figura 7, permitindo aos utilizadores criarem consultas visuais a diferentes bases de dados e serviços, sem escrever código SQL diretamente. Com esta ferramenta é possível unir dados provenientes de múltiplas fontes, aplicar filtros e agregações de forma intuitiva e exportar resultados para formatos como CSV, Excel ou Google Sheets. Adicionalmente, o Skyvia disponibiliza *add-ins* para integração direta com ferramentas como o Microsoft Excel e o Google Sheets, permitindo executar consultas e atualizar relatórios de forma automática [42].

A consulta de dados, por parte de aplicações externas faz-se através do *Connect*, que cria *endpoints* OData (v3 e v4) a partir de bases de dados e serviços *cloud*. Os clientes interagem por HTTP usando as convenções OData ($select, $filter, $orderby, $top/$skip) com respostas em JSON (v4) e suporte a autenticação básica quando configurada [43–45].

Para mostrar o percurso desde a construção até à publicação, apresentam-se dois ecrãs do Skyvia. A Figura 7 mostra o *Query Builder* no modo visual com três áreas em destaque: *Result*

Fields, *Filters* e *Sort Fields*, onde estão selecionadas colunas para a contagem de pedidos, o mês da data de encomenda e os nomes dos funcionários. Os filtros combinam condições *All* e *Any*, incluindo um intervalo de datas e a escolha de apelidos específicos. A ordenação está definida por data do pedido e apelido do funcionário. À direita surge o painel de detalhes do campo *"FK_Orders_Employees.LastName"*, onde é possível filtrar por um valor como "Leverling" [42].

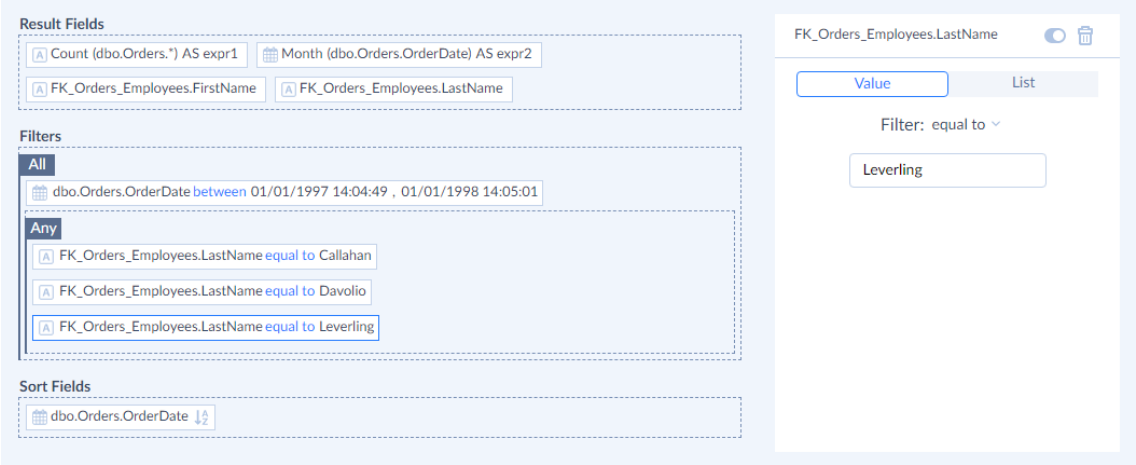


Fig. 7: Skyvia: *Query Builder* com *Result Fields*, *Filters* e *Sort Fields* preenchidos [42].

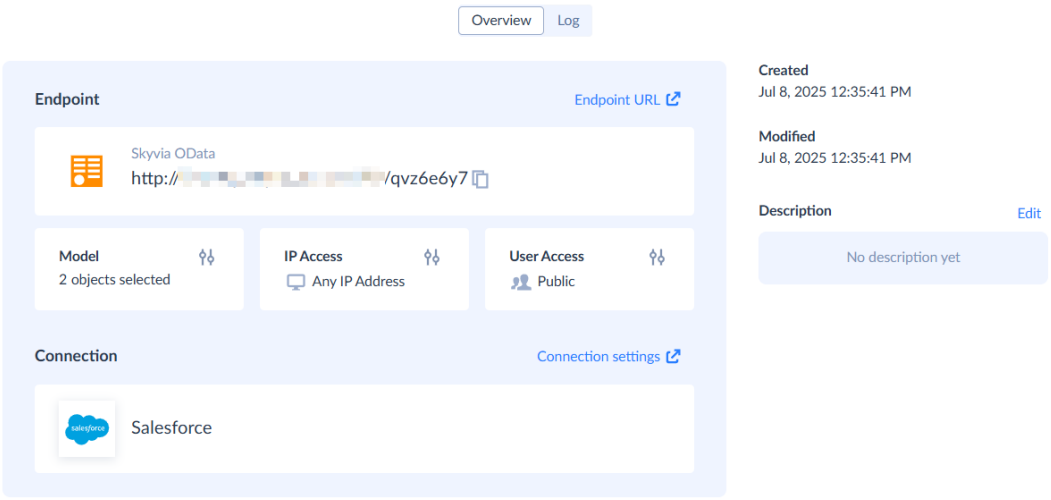


Fig. 8: Skyvia: *endpoint* OData já criado [46].

A Figura 8 apresenta o ecrã *Overview* de um *endpoint* OData já criado, com o URL público do serviço, o modelo com dois objetos expostos, as políticas de acesso por IP e por utilizador, as datas de criação e modificação e a ligação à fonte *"Salesforce"*. Este ecrã resume o contrato de consumo por HTTP e facilita o teste direto do *endpoint* [46].

3.4 Análise comparativa dos sistemas estudados

Com base na revisão realizada anteriormente de duas plataformas *low-code* (Mendix e OutSystems) e de um serviço de integração de dados em *cloud* com *visual query builder* (Skyvia), é possível identificar semelhanças e diferenças relevantes. A Tabela 1 apresenta um resumo das funcionalidades em comum e dos aspetos distintivos entre os sistemas analisados, focando nas dimensões definidas anteriormente:

Tabela 1: Resumo das funcionalidades dos sistemas existentes abordados.

Característica	**Mendix**	**OutSystems**	**Skyvia**
Tipo de solução	*Low-code* orientado a modelos	*Low-code* para aplicações empresariais	*iPaaS* (integração/ETL) com *visual query builder*
Construção de consultas	OQL, XPath e conectores (JDBC/SQL externo)	*Aggregates* visuais; *Advanced SQL Queries* para casos complexos	Construtor visual sem SQL; *add-ins* para Excel/Sheets
Fontes de dados	BDs SQL e alguns NoSQL via conetores	SQL Server, Oracle, MySQL, PostgreSQL; suporte a MongoDB	Várias fontes *cloud* (SaaS) e BDs SQL
Parametrização de consultas	Sim (parâmetros em OQL/XPath/SQL; microflows)	Sim (inputs nos *Aggregates* e parâmetros em SQL)	Limitado (filtros no UI; sem SQL)
Exportação de resultados	CSV/Excel	CSV/Excel	CSV/Excel/Pdf
Acesso por aplicações externas	REST (OpenAPI) e OData (HTTP com `$select`, `$filter`, `$orderby`, `$top`/`$skip`)	REST nativo com documentação gerada; também SOAP (legado)	*Endpoints* OData (Skyvia Connect) para HTTP/JSON
Guardar e reutilizar consultas	Sim (artefactos de dados no projeto)	Sim (Data Actions/Resources)	Sim (na *cloud*; sujeito a quotas)
Perfil de utilizador	Equipa técnica e *citizen developers* com literacia técnica	Equipas de desenvolvimento; requer conhecimentos em cenários avançados	Analistas e utilizadores de negócio; acessível a não técnicos

Observando a Tabela 1, verifica-se que as plataformas analisadas disponibilizam construção visual de consultas, ligação a múltiplas fontes de dados e mecanismos de exportação/consumo de resultados. Nos sistemas Mendix e OutSystems esta capacidade surge integrada no desenvolvimento de aplicações empresariais, enquanto no Skyvia está orientada à integração e exploração de dados em *cloud*. Em comum, destaca-se a possibilidade de parametrização básica, a reutilização/partilha de artefactos e a existência de opções (ainda que com limitações) de agendamento de execuções [31–33].

As diferenças tornam-se evidentes no posicionamento e profundidade das funcionalidades. O Mendix combina flexibilidade de modelação com várias opções de consulta (OQL/XPath e conec-

tores SQL), o que aumenta a expressividade mas também a exigência técnica e o custo de adoção. O OutSystems oferece um editor visual otimizado (*Aggregates*) para a maioria dos cenários e recorre a SQL quando a complexidade excede o editor, elevando a curva de aprendizagem em casos avançados. O Skyvia privilegia a simplicidade e a rapidez de ligação a serviços *cloud* e bases de dados [31–33].

Em poucas palavras, as plataformas analisadas ilustram diferentes formas de responder às necessidades de exploração de dados em ambientes *low-code*. Cada uma delas oferece contributos relevantes, mas também deixa em aberto lacunas significativas, sobretudo no equilíbrio entre acessibilidade para utilizadores não técnicos, flexibilidade para lidar com cenários complexos e integração direta com a realidade organizacional. Estas limitações constituem espaço para a evolução de novas abordagens que procurem conciliar simplicidade de uso com capacidade de resposta a contextos exigentes, abrindo caminho a soluções que articulem melhor a exploração de dados com a modelação organizacional.

As observações reunidas não são apenas uma síntese da literatura. A partir destas observações, resulta um conjunto prático de ideias e boas práticas que orientará o desenho ao longo do trabalho, servindo de base às decisões sobre configuração de pesquisas, parametrização, reutilização e exposição por serviços REST, com atenção à simplicidade do desenho e ao desempenho do sistema.

O capítulo seguinte detalha o enquadramento conceptual do DISME, a arquitetura que o suporta e o modelo EAV em que assenta.

4 Enquadramento do Sistema

Esta secção descreve a plataforma DISME no contexto deste projeto, apresenta a arquitetura global em que a componente de pesquisa se integra, revisita a versão anterior da pesquisa dinâmica e introduz a modelação de dados que sustenta o sistema. A ideia é que o leitor consiga, a partir daqui, compreender como as várias peças se articulam e por que razão as opções tomadas respondem aos objetivos definidos.

4.1 O protótipo DISME: visão geral e objetivos

O *Dynamic Information System Modeller and Executer* (DISME) nasce com a ambição de aproximar a maneira como a organização descreve o seu trabalho do sistema que o concretiza, com o mínimo de barreiras técnicas e sem exigir código sempre que o negócio muda. É uma plataforma *low-code* desenvolvida no *Enterprise Engineering Lab* da ARDITI e assenta na metodologia DEMO, o que permite captar a essência de processos e das interações entre atores e transformar essa representação em comportamento executável na aplicação. Ao mesmo tempo, preenche um espaço ainda pouco explorado ao funcionar, em simultâneo, como modelador organizacional, sistema de informação e gestor de *workflows*, configurável a partir de diagramas, formulários e tabelas, com uma interface que reduz de forma significativa a dependência de programação direta [47].

O principal objetivo da plataforma é tornar a criação e a adaptação de sistemas de informação mais ágeis, flexíveis e acessíveis. Essa ambição cumpre-se ao executar diretamente modelos num ambiente *low-code*, o que baixa a barreira técnica à inovação e mantém a correspondência entre o que se modela e o que efetivamente se observa em execução. Para que isto funcione no dia a dia, a plataforma reduz o esforço de passagem de requisitos a execução, mantém a lógica de negócio transparente e auditável, facilita integrações quando é preciso falar com sistemas externos e garante que o que se descreve com peças visuais ganha corpo na aplicação sem criar dependências escondidas. É precisamente esta coerência entre análise e execução que distingue o DISME de abordagens onde a lógica se dispersa por *scripts* difíceis de seguir. Por isso o protótipo reúne um conjunto de funcionalidades que acompanham todo o ciclo entre modelar, configurar e usar, preservando um fio contínuo e previsível do princípio ao fim [47].

4.1.1 Componentes principais do DISME

A aplicação do DISME tem três funcionalidades principais: *Diagram Editor*, *System Modeller* e *System Execution*.

O *Diagram Editor* permite ao utilizador visualizar os diagramas implementados no DISME, assim como editar esses mesmos diagramas. Para além disso tem como propósito facilitar a criação e design de novos diagramas de uma forma visual e, caso seja necessário e desejado, automatizar algumas das suas etapas de implementação. Esta componente ainda se encontra em desenvolvimento [47].

De forma resumida, no *System Modeller*, um ou mais utilizadores assumem a função de administrador e são capazes de moldar cada processo de uma organização criando e editando transações, suas relações, bem como associar formulários de entrada a essas transações ou em etapas de transações específicas. Esses formulários são gerados dinamicamente pelo executor do sistema quando os utilizadores estão cumprindo suas tarefas organizacionais. Os utilizadores que modelam

o sistema, não precisam de nenhuma habilidade de programação específica, apenas alguns conhecimentos básicos de modelagem de engenharia organizacional que se aproximam da "linguagem/ representação" utilizada dentro das organizações [47].

O *System Execution* é basicamente responsável por executar em modo de produção o sistema de informações modelado. Aqui os utilizadores assim que efetuam *login* no DISME, são direcionados ao *Dashboard*, onde é exibida uma lista das tarefas que eles têm permissão para executar. Um utilizador pode executar um ato de solicitação para iniciar algum processo específico ou reagir a um determinado estado de processo para o qual ele ou ela recebeu autoridade e responsabilidade para fazê-lo. Se alguma propriedade/entidade estiver associada a esse ato, o utilizador terá que preencher um formulário, gerado automaticamente com base nos parâmetros especificados [47].

4.2 Arquitetura global

O DISME é uma aplicação *web* com arquitetura cliente-servidor, como mostrado na Figura 10. No lado do cliente está uma aplicação de página única que apresenta o *dashboard*, o editor de regras e os ecrãs de consulta. No lado do servidor está a REST API, responsável por receber pedidos, aplicar a lógica de negócio e responder em JSON [27]. A base de dados guarda as descrições necessárias ao funcionamento do sistema e os registos do dia a dia. Esta comunicação é estável e previsível, por isso facilita testes, manutenção e evolução sem interromper o trabalho dos utilizadores.

Sobre esta topologia assenta a ideia que dá flexibilidade ao DISME, o *Adaptive Object-Model* (AOM), fazendo com que em vez de colocar tudo em classes de código, o sistema trabalhe com descrições que são lidas em tempo de execução. Ou seja, há um nível que executa e há um nível de descrições onde se define que entidades existem, que campos têm e como se relacionam. Quando a organização precisa de mudar algo, atualiza a descrição e a aplicação acompanha de imediato, sem reprogramação. Assim mantém-se o alinhamento entre o que se modela e o que se observa na interface [48, 49].

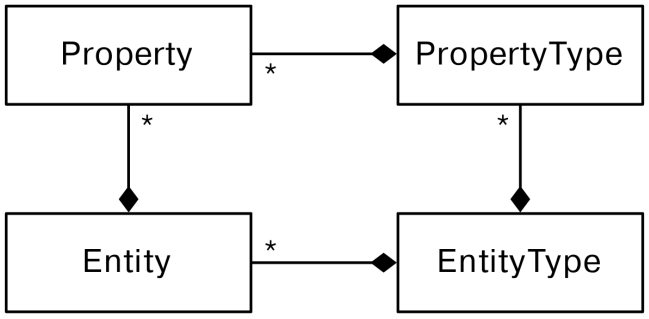


Fig. 9: O padrão Type Square [50].

Como é possível ver na Figura 9, o padrão *Type Square* (implementado com o modelo EAV) torna esta mecânica concreta ao trabalhar com quatro peças articuladas [50]:

- *"Entity Type"* descreve o tipo de entidade e o seu papel no domínio.
- *"Entity"* representa cada ocorrência desse tipo com identificador próprio.
- *"Property Type"* define um campo possível com nome e tipo de dados.

– E *"Property"* é o valor desse campo para uma "*Entity*" concreta.

Criar um novo tipo significa declarar um novo *"Entity Type"*, acrescentar informação significa declarar um novo *"Property Type"* [50]. Por exemplo, hoje existe o tipo "Audiência" com campos como "Assunto" e "Estado". Se amanhã for necessário registar o "Canal" do pedido, declara-se a *"Property"* "Canal" para o *"Entity Type"* "Audiência". A partir desse momento o campo aparece na interface, surge nas respostas da REST API e pode ser usado em filtros, sem tocar no código.

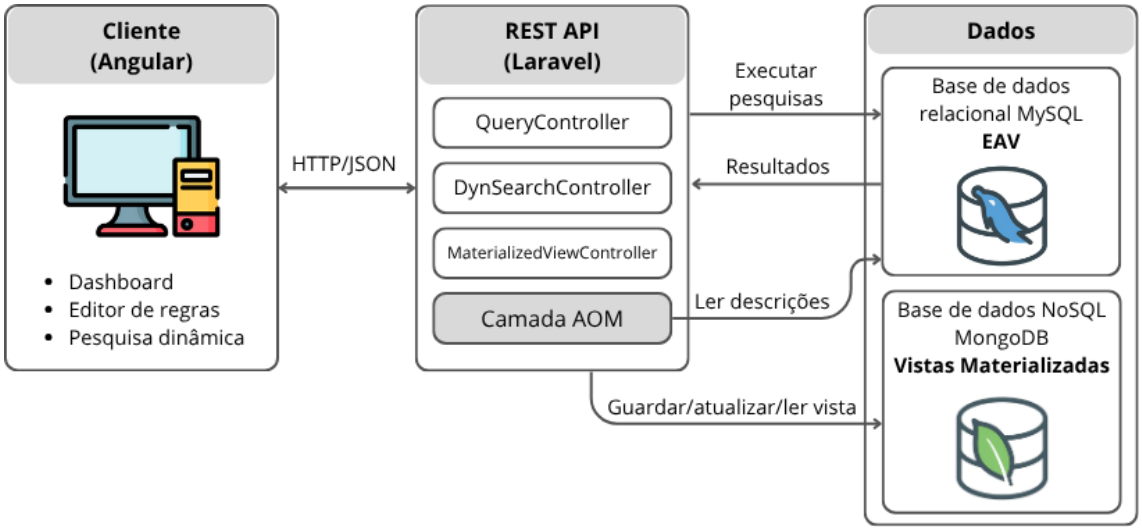


Fig. 10: Arquitetura geral do DISME, com foco na componente de pesquisa dinâmica.

4.3 Modelação da base de dados

Depois de apresentar a plataforma e a sua arquitetura nas subsecções 4.1 e 4.2, explica-se agora como os dados são guardados (com principal destaque para as tabelas mais importantes ao desenvolvimento deste projeto), uma vez que não foram efetuadas alterações nestas tabelas, retomando as ideias sobre *Adaptive Object-Model* e modelo EAV e mostrando, de forma direta, como o *Type Square* está concretizado nas tabelas do DISME. O esquema usa português e inglês desde a base, por isso existe a tabela "*language*" que serve de referência comum para os nomes que a interface apresenta ao utilizador, o que permite ter o mesmo valor com o nome correto em cada língua sem duplicar estruturas.

Tal como foi descrito, o *Adaptive Object-Model* é concretizado através do modelo EAV, que traduz o *Type Square* em tabelas e chaves claras. A Figura 11 mostra estas ligações.

A família das entidades começa na tabela "*ent_type*", onde cada linha define um tipo de entidade disponível no sistema e onde ficam campos técnicos que apoiam a execução como *state* e *transaction_type_id*. Os nomes que o utilizador vê para cada tipo ficam na tabela "*ent_type_name*" e ligam à tabela "*language*", o que permite apresentar o mesmo tipo com nomes diferentes conforme a língua selecionada. As instâncias reais são guardadas na tabela "*entity*", que referencia a tabela "*ent_type*" e garante a rastreabilidade necessária às operações de criação, atualização e histórico.

Fig. 11: Estrutura EAV das principais tabelas da pesquisa.

As propriedades que dão expressividade ao modelo estão na tabela "*property*". Cada registo declara que uma propriedade existe para um determinado "*ent_type*" e define o seu tipo. O campo *value_type* indica o tipo de valor esperado e abrange *text*, *bool*, *int*, *double*, *enum*, *date* e *time*. E ainda estão previstos casos especiais: *prop_ref* quando a propriedade aponta para outra entidade e *file* quando o valor representa um ficheiro. A propriedade pode também indicar a *scope* para dizer se é específica de uma entidade, de um processo ou global e pode ativar opções operacionais como *requires_translation* quando a apresentação depende da língua, *editable* quando é configurável na interface, *soft_delete* quando admite remoção lógica e *multiple_values* quando permite mais do que um valor por entidade. Se a propriedade for uma referência para outra propriedade, os campos *fk_property_id* e *fk_entity_type_id* definem o destino da ligação e evitam mudanças ao esquema físico. O nome que o utilizador vê para cada propriedade fica na tabela "*property_name*", também ligada à tabela "*language*".

Quando a propriedade tem um conjunto fechado de opções o modelo usa enumeração. As opções ficam na tabela "*prop_allowed_value*" e os respetivos nomes em "*prop_allowed_value_name*". Assim, o conjunto de valores pode crescer por configuração e a interface mostra sempre o nome correto em cada língua sem tocar na estrutura base. Quando a propriedade tem unidade associada, aplica-se o mesmo princípio através das tabelas de tipos de unidade e dos respetivos nomes, mantendo coerência entre a base de dados e a interface.

Os valores são guardados de forma vertical na tabela "*value*", onde cada linha liga uma "*entity*" a uma "*property*" e persiste o conteúdo no formato indicado por *value_type*. Quando a apresentação depende de tradução, o texto correspondente é guardado na tabela "*value_text*" e fica associado à tabela "*language*". Esta organização evita tabelas muito largas com muitas colunas vazias e reduz a presença de valores nulos, ao mesmo tempo que deixa claro o caminho para indexar e para compor as consultas usadas pela aplicação.

4.4 Versão anterior da componente *Dynamic Search*

Depois de apresentar a visão geral do DISME, a arquitetura e a base de dados nas subsecções anteriores, é importante revisitar a versão anterior da componente de pesquisa dinâmica, para entender o ponto de partida deste projeto, assim como os motivos pelos quais foram necessárias as melhorias descritas nos capítulos seguintes.

A versão anterior seguia um fluxo sequencial dividido em quatro etapas simples e totalmente visuais (ilustradas nas Figuras 12 a 14):

1. No primeiro passo era apresentado ao utilizador uma *checkbox* contendo todos os tipos de entidades armazenados na base de dados. Aqui o utilizador apenas podia selecionar tipos de entidade que estavam relacionados entre si através das suas propriedades, ou seja, só era permitido selecionar mais do que um tipo de entidade, se o primeiro tipo de entidade selecionado (tabela base) possuísse propriedades do tipo *prop_ref*. Deste modo, apenas os tipos de entidades que tivessem propriedades que o primeiro tipo de entidade selecionado apontava (através da coluna *fk_property_id*) eram permitidos selecionar para evitar relacionar tabelas que não tinham nada em comum. No final, aparecia um botão para avançar para o próximo passo:

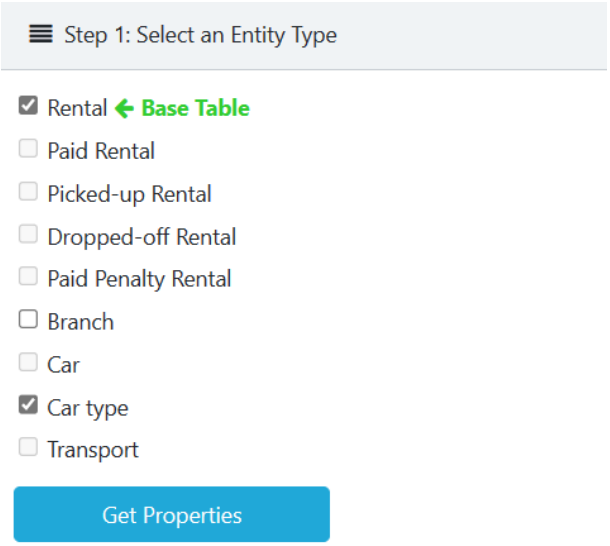


Fig. 12: Passo 1: Escolha dos tipos de entidade na versão anterior da componente.

2. No segundo passo, era apresentado ao utilizador, para cada tipo de entidade selecionado duas *selectboxes* à frente, contendo todas as propriedades pertencentes a esse tipo de entidade. A primeira *selectbox* servia para selecionar e guardar todas as propriedades cujo valor o utilizador desejava ver na tabela final de resultados, já a segunda *selectbox* servia para selecionar e guardar todas as propriedades que o utilizador desejava usar como filtro nos resultados finais, não sendo obrigatório preencher a mesma. No final, aparecia um botão para avançar para o próximo passo:

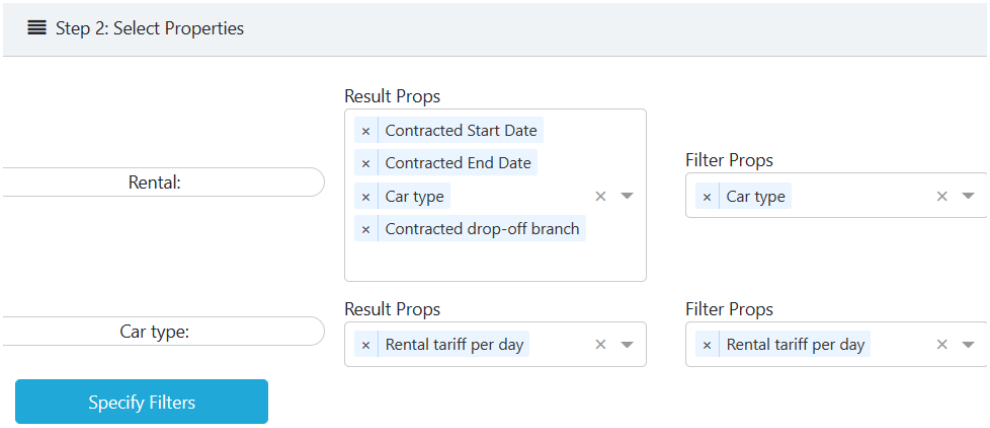


Fig. 13: Passo 2: Escolha das propriedades na versão anterior.

3. No terceiro passo, caso existissem propriedades de filtragem, era apresentado o *queryBuilder* para filtrar os resultados da pesquisa à base de dados. O utilizador poderia inserir as *rules* e *rulesets* que pretendesse. Cada *rule* continha um campo que continha uma *selectbox* com todas as propriedades de filtragem, um campo com o operador, e um campo vazio para o inserir o valor que pretendia usar como filtro. Estes dois últimos campos variavam de acordo com o tipo de valor da propriedade. Por exemplo, no caso de o tipo de valor da propriedade ser um

date, apenas era permitido ao utilizador inserir datas no terceiro campo, o mesmo acontecia com números, texto, entre outros. No campo do operador, por exemplo eram apresentadas as opções ['=', '! ='] no caso de o valor do tipo de propriedade ser um *prop_ref* ou *enum*, as opções ['=', '! =', '<', '<=', '>', '>='] quando o valor da propriedade era do tipo *double* ou int, etc. Aqui é importante ressaltar que esta parte foi a que menos sofreu alterações de toda a componente. No final, aparecia um botão para avançar para a apresentação dos resultados da *query*:

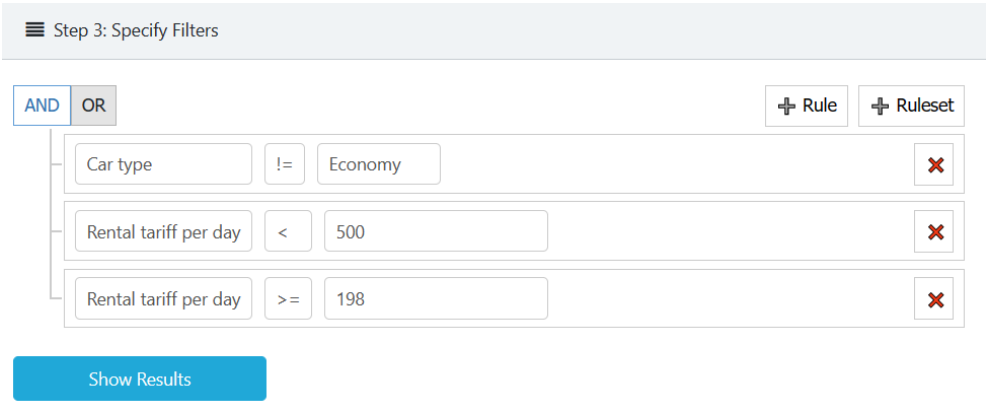


Fig. 14: Passo 3: Construção de filtros no *query builder* da versão anterior.

4. No final, era mostrado em uma tabela os resultados da *query*, com um cabeçalho de acordo com as propriedades selecionadas para incluir no resultado, com a possibilidade do utilizador exportar os resultados no formato .XLSX.

4.4.1 Limitações da versão anterior

Embora oferecesse uma construção visual e consistente das pesquisas, com passos simples e uma curva de aprendizagem suave porque cada ecrã pedia apenas uma decisão de cada vez, esta versão tinha limitações importantes que dificultavam a adoção em cenários mais exigentes, nomeadamente:

- Tempo de construção muito longo, até para casos simples. Para obter os resultados era necessário passar pelas quatro etapas obrigatoriamente, onde cada avanço significava mais um clique, muitas vezes desnecessário, nomeadamente nos casos em que a pesquisa não tinha filtros. Para além disto, nas *select boxes*, para escolher cada propriedade eram necessários dois cliques sucessivos, um para selecionar a *selectbox* e outro para selecionar a propriedade desejada, o que tornava morosa a configuração de consultas com muitas colunas.
- Não existiam funções de agregação nem *group by*, o que impedia análises mais complexas aos dados.
- Não havia opção para guardar e atualizar *queries*, o que desincentivava a construção de uma biblioteca de pesquisas reutilizáveis.
- A ausência de parâmetros dinâmicos impedia reutilizar a mesma consulta com valores variáveis, obrigando a reconstruir filtros sempre que se pretendia alterar algum valor.

- A execução era exclusiva sobre MySQL (EAV), com tempos de resposta elevados, principalmente em cenários com maior volume de dados e junções de tabelas.
- Não existia uma REST API dedicada para disponibilização dos resultados a sistemas externos.

Estas limitações constituíram o ponto de partida para a evolução da componente. As melhorias introduzidas visaram encurtar o percurso do utilizador, reduzir cliques desnecessários, reforçar a expressividade com agregações e parâmetros, permitir guardar e atualizar consultas e permitir integração por REST, aumentando a interoperabilidade do DISME. A avaliação da versão anterior permitiu identificar o que manter e o que mudar e serviu de base à definição dos requisitos apresentada no Capítulo 5.

5 Metodologia e Especificação da Solução

Este capítulo apresenta a metodologia e a especificação da solução. Depois da introdução teórica, estado da arte e do enquadramento do sistema, consolida a forma como o trabalho foi conduzido e as decisões tomadas, preparando a leitura dos capítulos de Implementação e Avaliação.

A metodologia adotada foi incremental e orientada a evidência. O progresso fez-se por ciclos curtos com objetivos claros e entregas funcionais que podiam ser verificadas em ambiente de teste. O levantamento e a análise de requisitos resultaram sobretudo da avaliação da versão anterior do DISME, da comparação com sistemas existentes, abordados no Capítulo 3, e de discussões técnicas com a equipa do projeto. As decisões foram validadas internamente por revisão técnica e por protótipos executáveis.

O capítulo organiza-se de forma simples. Primeiro, apresentam-se os requisitos funcionais e não funcionais, em seguida, justifica-se a decisão de usar vistas materializadas no projeto e a escolha do MongoDB como repositório para guardá-las e, no final, mostra-se o esquema adotado para armazenar as pesquisas na base de dados.

5.1 Requisitos do sistema

Este subcapítulo consolida os Requisitos Funcionais (RF) e os Requisitos Não Funcionais (RNF) do DISME para a componente de pesquisa dinâmica. Os requisitos são identificados como RFx (funcionais) e RNFx (não funcionais), com subitens RFx.y quando necessário. Sendo assim, considera-se, para efeitos de especificação, o utilizador que configura as pesquisas na interface como ator principal, sem definir perfis adicionais.

5.1.1 Requisitos funcionais (RF)

Os requisitos funcionais dizem o que o sistema deve fazer para que os utilizadores cumpram as suas tarefas. Têm de ser verificáveis, por isso escrevem-se de forma concreta e observável, com condições de entrada claras, comportamento esperado e resultado mensurável [51].

Os requisitos funcionais apresentados a seguir estão organizados por partes: configuração de pesquisas, parametrização, publicação por REST API e vistas materializadas.

5.1.1.1 Requisitos funcionais da componente de pesquisa dinâmica

RF1 - O sistema deverá permitir listar as pesquisas existentes, apresentando nome, descrição, data de criação/atualização e ações de guardar e editar.
RF1.1 - O sistema deverá permitir pesquisar por texto e ordenar por colunas na tabela de pesquisas.
RF1.2 - O sistema deverá permitir criar uma nova pesquisa a partir da configuração corrente.
RF1.2.1 - O sistema deverá identificar como campos obrigatórios, no ato de criação, pelo menos: nome da pesquisa, tipo de entidade base, e pelo menos uma propriedade a incluir no resultado final.
RF1.3 - O sistema deverá permitir atualizar uma pesquisa existente.
RF1.3.1 - O sistema deverá alterar referências internas a pesquisas quando ocorrer a atualização da mesma.
RF1.4 - O sistema deverá permitir duplicar uma pesquisa (guardar como) para reutilização.
RF1.5 - O sistema deverá permitir eliminar uma pesquisa, com confirmação prévia.
RF1.6 - O sistema deverá reconstituir toda a configuração de uma pesquisa previamente guardada.

RF1.7 - O sistema deverá permitir iniciar a configuração de uma nova pesquisa do zero (construtor) ou pela tabela de pesquisas guardadas.

RF2 - O sistema deverá permitir selecionar o tipo de entidade base da pesquisa.
RF2.1 - O sistema deverá apresentar as propriedades dos tipos de entidades, indicando o tipo de valor (texto, número, *bool*, *enum*, referência, data/hora).
RF2.2 - O sistema deverá permitir navegar relações válidas entre tipos de entidades a partir da tabela base para inclusão de propriedades relacionadas.

RF3 - O sistema deverá permitir definir as propriedades (colunas) do resultado.
RF3.1 - O sistema deverá permitir marcar propriedades como incluídas no resultado.
RF3.2 - O sistema deverá permitir marcar propriedades como filtros para aplicar no resultado.
RF3.3 - O sistema deverá permitir definir rótulos (*alias*) por coluna e a ordenação ascendente/descendente.
RF3.4 - O sistema deverá disponibilizar funções de agregação permitidas por tipo de propriedade e respetivo GROUP BY. Por exemplo, COUNT para todos os tipos, SUM e AVG apenas para numéricos.
RF3.5 - O sistema deverá validar coerência entre agregações e agrupamentos e, em caso de incoerência, devolver mensagem de erro.

RF4 - O sistema deverá disponibilizar um *query builder* para escrita de filtros.
RF4.1 - O sistema deverá impedir a aplicação de operadores incompatíveis com o tipo de dado.
RF4.2 - O sistema deverá suportar filtros de intervalo para datas e números, com limites configuráveis.

RF5 - O sistema deverá permitir executar uma pesquisa e apresentar os resultados.
RF5.1 - O sistema deverá devolver um objeto de resultados com `header` (*array* com o cabeçalho da tabela), `resultRows` (*array* com as linhas da tabela).
RF5.2 - O sistema deverá apresentar a duração da execução dos resultados ao utilizador.
RF5.3 - O sistema deverá permitir cancelar execuções longas.
RF5.4 - O sistema deverá aplicar limites configuráveis de linhas e devolver erro descritivo quando excedidos.

5.1.1.2 Requisitos funcionais de parâmetros dinâmicos

RF6 - O sistema deverá permitir marcar filtros como parâmetros.
RF6.1 - O sistema deverá permitir atribuir um nome a cada parâmetro.
RF6.1.1 - O sistema deverá validar que o nome do parâmetro não é vazio, não contém caracteres especiais e é único na base de dados.
RF6.2 - O sistema deverá permitir definir valores por defeito e obrigatoriedade por parâmetro.
RF6.3 - O sistema deverá apresentar a lista de parâmetros na ordem mostrada no *query builder*.
RF6.4 – O sistema deverá validar o valor de cada parâmetro segundo o seu tipo.

5.1.1.3 Requisitos funcionais de publicação e integração por REST

RF7 - O sistema deverá disponibilizar *endpoints* REST para os resultados das pesquisas possam ser acedidos por aplicações externas.
RF7.1 - O sistema deverá apresentar ao utilizador o URL do *endpoint* no padrão /api/auth/dynamic/query/{nome}?p1=&p2=...
RF7.2 - O sistema deverá permitir testar o *endpoint* a partir da interface, devolvendo o JSON com

os resultados ou a mensagem de erro descritiva.
RF7.3 - O sistema deverá permitir nomear o *endpoint* a partir da interface.
RF7.4 - O sistema deverá permitir escolher que tipo de autorização é necessária para consultar os resultados através do *endpoint*.

5.1.1.4 Requisitos funcionais das vistas materializadas

RF8 - O sistema deverá permitir guardar os resultados de uma pesquisa como vista materializada.
RF8.1 - O sistema deverá reutilizar uma vista materializada quando o tipo de entidade base, propriedades, agregações e filtros são idênticos ao da pesquisa a executar.
RF8.2 - O sistema deverá suportar políticas de expiração por tempo (TTL).
RF8.3 – O sistema deverá permitir o armazenamento e a atualização manual da vista materializada por ação do utilizador.
RF8.4 – O sistema deverá guardar automaticamente uma vista materializada sempre que o resultado da pesquisa ultrapasse um limiar de linhas configurável, definido num ficheiro de configuração.

5.1.2 Requisitos não funcionais (RNF)

Os requisitos não funcionais são, por natureza, mais difíceis de medir e descrever. Aplicam-se ao sistema como um todo e não a um perfil específico de utilizador e organizam-se por atributos de qualidade como *usability*, *security*, *safety*, *efficiency* e *portability*, entre outros [51].

RNF1 - O sistema deverá apresentar tempos de resposta adequados, beneficiando de *cache* (vistas materializadas) em execuções repetidas da mesma forma de pesquisa.
RNF1.1 - O sistema deverá suportar paginação eficiente no servidor.
RNF1.2 - O sistema deverá permitir configurar limites de execução e de exportação.

RNF2 - O sistema deverá garantir segurança de acesso aos serviços.
RNF2.1 - O sistema deverá exigir autenticação por *Bearer Token* nos *endpoints* internos do sistema.
RNF2.2 - O sistema deverá exigir autenticação por *Bearer Token* para acesso ao componente de pesquisa dinâmica.
RNF2.3 - O sistema deverá registar acessos e operações relevantes para verificação.

RNF3 - O sistema deverá assegurar fiabilidade e consistência dos dados.
RNF3.1 - O sistema deverá executar atualizações com migração de referências quando aplicável.
RNF3.2 - O sistema deverá evitar referências órfãs após atualizações de pesquisas.
RNF3.3 - O sistema deverá fornecer mensagens de erro uniformes e não bloquear a interface.

RNF4 - O sistema deverá ser fácil de manter e de evoluir.
RNF4.1 - O sistema deverá separar responsabilidades por camadas (UI, serviços REST, base de dados).
RNF4.2 - O sistema deverá manter uma gramática estável para regras/operadores e parâmetros, para possível extensão sem erros.
RNF4.3 - O sistema deverá permitir testes com serviços isoláveis.

RNF5 - O sistema deverá garantir interoperabilidade.
RNF5.1 - Datas em ISO 8601 em UTC na API e codificação UTF-8.
RNF5.2 - O sistema deverá respeitar a semântica HTTP.

RNF6 - O sistema deverá suportar internacionalização e acessibilidade.
RNF6.1 - O sistema deverá suportar a interface em pelo menos duas línguas (português e inglês).

RNF7 - O sistema deverá proporcionar observabilidade.
RNF7.1 - O sistema deverá registar *logs* de operações CRUD, chamadas REST e erros com contexto mínimo necessário.
RNF7.2 - O sistema deverá expor métricas técnicas (tempos médios de execução, taxa de erros, etc).

5.2 Materialização e *cache* de resultados

Esta subsecção não repete a introdução teórica do Capítulo 2.4. O objetivo é apenas justificar a decisão de materializar e de usar *cache* no contexto concreto do DISME. Posto isto, a decisão de incorporar vistas materializadas na componente de pesquisa dinâmica decorre de três fatores de desenho que estão presentes no DISME e que condicionam a experiência do utilizador e os requisitos de desempenho. Primeiro, o modelo EAV sobre relacional é flexível, mas torna as consultas pesadas à medida que aumentam propriedades e filtros, o que afeta a fluidez. Depois, o trabalho diário repete muitas pesquisas para ecrãs e *dashboards*, por isso compensa reutilizar resultados já calculados. Por fim, a integração por REST precisa de tempos de resposta estáveis para facilitar o consumo por serviços externos e manter a previsibilidade no lado do cliente. Desta forma, a materialização desloca o custo para momentos controlados e transforma leituras repetidas em acessos diretos a resultados já calculados, como já ficou sublinhado na revisão de literatura.

Na prática, neste projeto existem dois modos complementares de materialização que serão descritos com mais detalhes no Capítulo da Implementação.

Para suportar os ecrãs e a API, o projeto usa ainda *cache* persistente com chave de consulta. Na primeira execução, calcula no MySQL, serializa o resultado e guarda um documento JSON com cabeçalho e linhas. Nas execuções seguintes, lê diretamente da *cache*. Este desenho reduz a carga na base de dados de origem, acelera a abertura de ecrãs e *dashboards* e mantém tempos de resposta mais estáveis para consumo por REST. O benefício é maior quando há pesquisas recorrentes ou variações muito próximas da mesma consulta.

Os riscos são conhecidos e mitigados. Pode surgir desatualização face às tabelas de origem, por isso define-se uma política de atualização manual ou agendada e mostra-se a hora do último *refresh*. O espaço em disco cresce com o número de consultas guardadas, o que se controla com chaves consistentes, limpeza regular de objetos pouco usados e com o limiar das 100 linhas, o que evita materializar resultados triviais.

No âmbito deste projeto, centrado em leitura intensiva e previsibilidade de latência, a combinação de vistas materializadas com *cache* persistente oferece o melhor equilíbrio entre esforço de implementação, estabilidade e operação simples.

5.3 Escolha do MongoDB para guardar as vistas materializadas

A escolha da base de dados a utilizar é uma das decisões mais importantes a tomar. Se a escolha for errada, a migração para outra base de dados torna-se um procedimento muito dispendioso e arriscado [52].

Foram consideradas várias bases de dados, tendo em conta vários fatores: popularidade, facilidade de integração com o Laravel, custo, escalabilidade, performance e padrões de pesquisa. As principais bases de dados consideradas foram então: ArangoDB, Neo4j, Redis e MongoDB.

ArangoDB é uma base de dados *open-source* e multi modelo, desenvolvida em C++. Suporta modelos de dados de grafo, de documentos e de *key-value* que permitem aos utilizadores combinar livremente todos os modelos de dados numa única consulta [53].

Neo4j é uma base de dados de grafo que é deliberada para armazenar e processar grafos para gerir dados interligados. Este tipo de bases de dados ajudam a encontrar relações entre os dados e a extrair o seu verdadeiro valor. Todavia, estas estruturas tendem a não ser boas para armazenar e gerir enormes listas de coisas [54].

Redis é uma base de dados *open-source* conhecida por ser uma base de dados rápida, de armazenamento de dados de *key-value in-memory*, *cache*, *message broker*, e fila de espera. Contudo, apesar de ser a melhor opção para armazenar grandes volumes de dados, requer muita RAM porque é *in-memory*, para além de ter uma linguagem de consulta mais complicada de aprender [55]. Apesar de não ser uma base de dados de grafo, possui um modulo denominado *RedisGraph* que permite armazenar dados com esta estrutura [56].

O MongoDB é escrito em C++ e oferece alta disponibilidade, fácil escalabilidade e melhor desempenho. Ele armazena dados na forma de documentos JSON. Esta base de dados oferece suporte à consulta dinâmica de documentos usando uma linguagem de consulta baseada em documentos que é quase tão poderosa quanto o SQL. As principais características do MongoDB são *AutoSharding*, a possibilidade de replicação ascendente e indexação [57].

Com base na informação acima, a base de dados ArangoDB seria uma excelente opção, contudo após uma pesquisa mais aprofundada descobriu-se que não é compatível com a versão do Laravel utilizada no protótipo [53].

O Neo4j só seria integrado no DISME apenas para guardar dados estruturados, uma vez que não é uma boa base de dados para armazenar grandes listas.

O Redis é uma excelente base de dados, porém o facto de consumir muita RAM, e o facto de não suportar uma linguagem de consulta fizeram com que a integração desta não fosse muito viável [55].

Sendo assim, no final, a base de dados MongoDB revelou ser a melhor opção, pois é a que apresenta mais vantagens em detrimento das outras bases de dados, e possui melhor documentação para integração com o Laravel. A maior desvantagem é que a capacidade de trabalhar com estruturas de dados em grafo nunca será tão boa quanto trabalhar com um modelo de base de dados de grafo [57]. O Laravel não suporta o MongoDB diretamente, então um pacote de terceiros deverá ser instalado, e um dos pacotes mais populares sugeridos para uso é a biblioteca "Laravel-mongoDB" [58]. Esta biblioteca usa as classes originais do Laravel, o que significa que são usados exatamente os mesmos métodos [58].

5.4 Modelo de dados das pesquisas guardadas

Para guardar as *queries* foram adicionadas à base de dados as tabelas que armazenam a configuração da pesquisa para futura reutilização. A definição de alto nível é guardada na tabela *"query"*, onde cada consulta tem um identificador e um *base_ent_type_id* que ancora o grafo de navegação.

O nome da pesquisa é guardado em *"query_name"*. A seleção de colunas para o resultado, com opções de ordenação, agrupamento e agregação, é registada em *"query_has_result"*, que liga cada *"property"* à *"query"* correspondente.

O *query builder* organiza-se como uma árvore n-ária através de várias tabelas. Os nós são guardados em *"query_term"*, e podem ser do tipo *rule* ou *ruleset*. Os grupos lógicos AND e OR vivem em *"ruleset"*. A associação entre grupos e nós é materializada em *"ruleset_has_query_term"*. Os filtros que representam condições concretas são guardados em *"query_filter"* e indicam a propriedade, o operador e os valores envolvidos. Quando pretende-se reutilizar a mesma consulta com valores diferentes sem a reconstruir, entram os parâmetros descritos em *"query_parameter"* e associados aos nós certos através de *"term_has_query_parameter"*. A ligação da árvore à tabela *query* aparece em *"query_has_term"*.

Para publicação, a tabela *"rest_api"* mapeia uma pesquisa guardada para um *endpoint* HTTP e define quais os parâmetros esperados no pedido.

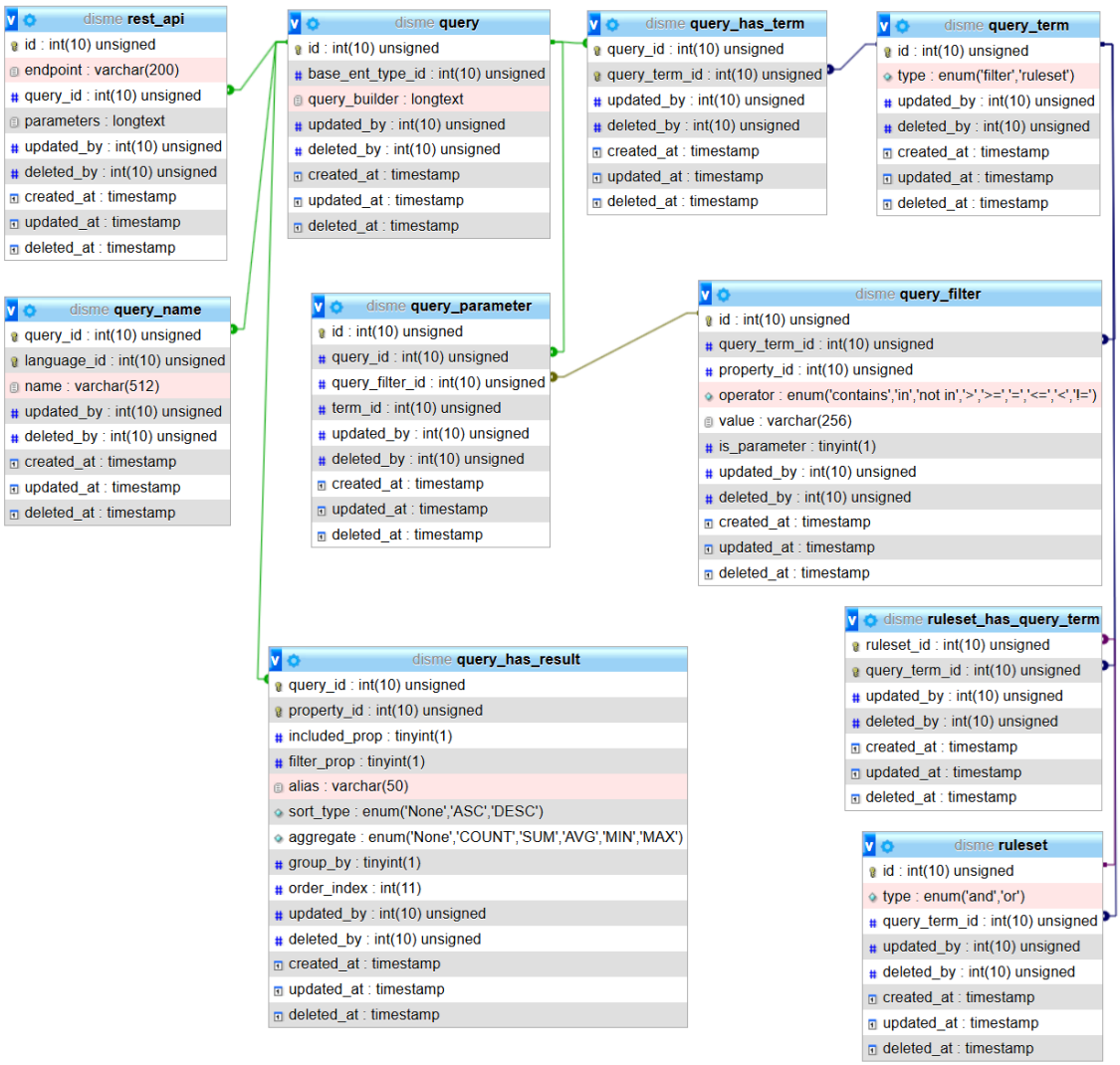


Fig. 15: Modelo de dados das pesquisas guardadas e dos *endpoints* REST.

6 Implementação

6.1 Introdução

A implementação corresponde à fase em que a solução proposta foi transformada em funcionalidades concretas, integradas na plataforma DISME. Sendo assim, este capítulo descreve as tecnologias adotadas, a estrutura do código do projeto e o modo como cada objetivo definido foi implementado no sistema. A fundamentação teórica e a motivação encontram-se nos capítulos anteriores, enquanto a análise quantitativa de desempenho e usabilidade é apresentada no capítulo de Avaliação.

A solução evoluiu incrementalmente sobre a versão anterior do DISME, apresentada no Capítulo 4.4, corrigindo as limitações identificadas e adicionando novas funcionalidades ao longo da execução. Em particular, foram implementadas e refinadas funcionalidades que permitem configurar e executar pesquisas complexas de forma visual, integrar mecanismos de *cache* utilizando a base de dados NoSQL MongoDB e disponibilizar resultados através de serviços REST. Paralelamente, desenvolveu-se ainda a funcionalidade que permite às *queries* receber parâmetros de forma dinâmica.

6.2 Tecnologias e ferramentas utilizadas

O desenvolvimento da solução assentou em um conjunto de tecnologias já utilizadas na versão anterior do DISME, complementadas por novas ferramentas que permitiram dar resposta aos objetivos definidos.

No *frontend*, o DISME recorre à *framework* Angular[3]. Esta escolha justifica-se pela facilidade em construir componentes reutilizáveis e pela integração nativa com TypeScript, o que permite estruturar a lógica da aplicação de forma clara e organizada. Sendo uma *framework* pensada para aplicações *single page*, o Angular revelou-se particularmente útil na implementação da componente de pesquisa dinâmica, pois a interação com o utilizador é constante, o que exige que as atualizações sejam rápidas na interface.

No *backend*, o protótipo usa o Laravel[4], uma *framework* PHP que simplifica a criação de APIs e a ligação a bases de dados através do *ORM Eloquent*. Para além de fornecer mecanismos de autenticação, segurança e gestão de rotas, o Laravel foi fundamental para gerir a execução das *queries* dinâmicas e coordenar a comunicação entre o *frontend*, a base de dados relacional e o repositório NoSQL.

A arquitetura deste projeto combina duas bases de dados: MySQL[5] e MongoDB[6]. O MySQL mantém-se como a base de dados principal, guardando toda a informação estrutural da plataforma. Já o MongoDB foi introduzido como apoio às pesquisas, permitindo armazenar vistas materializadas e apresentar os resultados de forma quase imediata. Para facilitar a visualização e exploração dos documentos guardados em MongoDB, utilizou-se o MongoDB Compass[7].

[3] Angular – `https://angular.dev`
[4] Laravel – `https://laravel.com/`
[5] MySQL – `https://www.mysql.com/`
[6] MongoDB – `https://www.mongodb.com/`
[7] MongoDB Compass – `https://www.mongodb.com/products/compass`

Para além destas tecnologias, várias ferramentas de apoio acompanharam o desenvolvimento. O XAMPP[8] serviu como ambiente local para executar e testar a aplicação com Apache[9], PHP e MySQL. O Postman[10] foi usado para testar e validar os *endpoints* da REST API, garantindo que os resultados eram devolvidos de forma correta. O GitHub[11] foi a plataforma de controlo de versões, assegurando histórico e colaboração no código.

A nível de IDEs, alternou-se entre VS Code[12] e PhpStorm[13], ambos escolhidos pela integração com PHP, TypeScript e SQL, facilitando a escrita e depuração do código em diferentes fases do projeto.

Para garantir reprodutibilidade, a Tabela 2 mostra as versões usadas das tecnologias e dependências necessárias para executar o projeto (as dependências encontram-se em `composer.lock` e `package-lock.json`), e respetivos comandos para verificar as versões.

Tabela 2: Tecnologias e versões utilizadas na implementação, com comandos de verificação.

	Componente	**Versão**	**Verificação**
Frontend	Node.js (+ npm)	v14.16.0 / v6.14.11	`node -v` / `npm -v`
Frontend	Angular	v8.2.14	`npx ng version`
Frontend	TypeScript	v3.5.3	`npx tsc -v`
Backend	PHP	v7.4.11	`php -v`
Backend	Composer	v2.4.4	`composer --version`
Backend	Laravel	v5.8.38	`php artisan --version`
Backend	Extensão PHP ext-mongodb	v1.13.0	`php -r "echo phpversion('mongodb');"`
Dados	Servidor SQL (MariaDB)	v10.4.14	`mysql --version`
Dados	Cliente SQL (mysql)	v15.1	`mysql -V`
Dados	Biblioteca PHP mongodb/mongodb	v1.12.0	`composer show mongodb/mongodb`
Dados	Pacote Laravel jenssegers/mongodb	v3.5.3	`composer show jenssegers/mongodb`

6.3 Estrutura do código

No *frontend* (Angular), os ficheiros principais encontram-se em `./src/app` e estão separados entre componentes e serviços. Enquanto que os componentes tratam do ecrã e das interações, os serviços tratam dos dados e das chamadas HTTP. A consequência desta separação é um código mais simples de manter, evoluir e testar:

[8] XAMPP – `https://www.apachefriends.org/`
[9] Apache – `https://httpd.apache.org/`
[10] Postman – `https://www.postman.com/`
[11] GitHub – `https://github.com/`
[12] Visual Studio Code – `https://code.visualstudio.com/`
[13] PhpStorm – `https://www.jetbrains.com/phpstorm/`

- `./package.json`: descreve o projeto e declara as dependências necessárias para instalar e executar o protótipo.
- `./angular.json`: ficheiro que o Angular CLI lê para saber quais projetos existem no *workspace* e como executar cada alvo (*build*, *serve*, *test*, e2e), além de *assets*, estilos/*scripts* globais, *paths* e *schematics*.
- `./tsconfig.json`: configura o TypeScript e as opções de compilação do projeto.
- `./src/index.html`: é o principal documento HTML do Angular, pois contém o *<app-root>* onde os *bundles* são injetados e a aplicação é renderizada.
- `./src/main.ts`: primeiro ficheiro a ser executado após a inicialização da aplicação.
- `./src/environments/`: pasta com os ficheiros `environment.ts`, `environment.prod.ts`, entre outros, onde definem-se variáveis e URLs por ambiente.
- `./src/app/app.module.ts` — este é o módulo raiz da aplicação. Aqui os componentes são declarados, os módulos importados e os *providers* globais configurados.
- `./src/app/app-routing.module.ts` — define as rotas da aplicação, incluindo as da componente de pesquisa, fazendo com que a navegação seja previsível.
- `./src/app/dynSearch/` — núcleo da componente de pesquisa dinâmica:
  - `dynSearch.component.html` — interface traduzível (i18n) onde o utilizador constrói a pesquisa de forma visual.
  - `dynSearch.component.ts` — liga o ficheiro anterior aos serviços REST, coordena o construtor visual, a grelha AG Grid e ainda é responsável por guardar, carregar, publicar e exportar as pesquisas realizadas.
  - `save-query.component.ts`, `update-query.component.ts`, `delete-query.component.ts` — ficheiros responsáveis pelos *modals* de confirmação quando o utilizador deseja guardar, editar ou apagar uma pesquisa, respetivamente.
- `./src/app/modal-dynamic-rest-api/modal-dynamic-rest-api.component.ts` — permite criar o *endpoint* REST e editar parâmetros e valores, tornando o contrato de integração explícito para clientes externos.
- `./src/app/shared/interfaces/` — tipos TypeScript como `Query` que definem a gramática canónica trocada com a API, garantindo tipagem forte e evitando desvios entre UI e servidor.
- `./src/app/shared/rest-api/dyn-search-api.service.ts` — serviço Angular que faz a ponte entre a componente de pesquisa dinâmica e o controlador geral da pesquisa dinâmica no *backend*. Encapsula as chamadas HTTP e oferece tratamento uniforme de erros.
- `./src/app/shared/rest-api/query-api.service.ts` — serviço Angular que faz a ponte entre a componente de pesquisa dinâmica e o controlador específico para guardar, ler, atualizar e remover pesquisas no *backend*. Encapsula as chamadas HTTP e oferece tratamento uniforme de erros.

No *backend* (Laravel), a API é protegida por *middlewares* de autenticação, CORS e limites de pedidos. Os controladores funcionam como "porteiros", pois recebem o pedido, validam, chamam os serviços certos e constroem a resposta. Os modelos são o mapa das tabelas MySQL e das

coleções MongoDB e garantem leituras e escritas consistentes. Com esta divisão, é possível mudar a implementação interna sem alterar os URLs:

- `./composer.json` regista dependências PHP, *scripts* e metadados do projeto, permitindo instalação reproduzível com Composer.
- `./.env` define variáveis de ambiente, incluindo DSNs de MySQL e MongoDB, chaves de aplicação e parâmetros de *cache*.
- `./routes/api.php` — define as rotas da API, mapeando os endpoints REST, como os da pesquisa dinâmica para os controladores `QueryController` e `DynSearchController`.
- `./app/Http/Controllers/DynSearchController.php` — execução de pesquisas e publicação por REST, valida o JSON canónico, constrói a pesquisa, gera SQL e aplica filtros.
- `./app/Http/Controllers/QueryController.php` — responsável por gerir, guardar, atualizar e remover o artefacto *query*.
- `./app/Http/Controllers/MaterializedViewController.php` — responsável pela leitura e criação das vistas materializadas na base de dados MongoDB.
- `./app/Query.php` — modelo *Eloquent* da tabela `query` com a forma da pesquisa.
- `./app/Value.php` e `./app/EntType.php` — exemplos de modelos do núcleo EAV.
- `./app/MaterializedView.php` — modelo com `connection='mongodb'` que guarda a estrutura, os parâmetros e os dados dos resultados materializados.
- `./database/migrations/`: contém os *scripts* que criam e modificam tabelas, que permitem instalar a base de dados do zero.
- `./database/seeders/`: contém *scripts* que preenchem a base de dados com dados úteis principalmente para testes.
- `./config/database.php` e `./config/cache.php`: estes dois ficheiros definem as ligações às bases de dados MySQL e MongoDB.
- `./storage/logs/`: guarda ficheiros de registo com um identificador por pedido para ajudar a seguir o percurso UI→API→dados e a recolher métricas.

Esta organização separa com nitidez a UI, a API e os dados. Os serviços ficam pequenos, com responsabilidades claras e fáceis de testar. A evolução faz-se sem afetar os clientes porque as rotas e os contratos se mantêm estáveis, e porque o formato canónico da pesquisa está separado da execução e da publicação.

6.4 Como cada objetivo foi implementado

Partindo da organização do código apresentada na secção anterior, esta secção descreve, objetivo a objetivo, como a solução se concretiza em *software* executável. Cada subsecção contém exemplos com diversas capturas de tela da interface, que ilustram como as diferentes funcionalidades foram implementadas no projeto.

6.4.1 Configuração e execução de pesquisas

A componente de pesquisa dinâmica foi concebida para que utilizadores com diferentes perfis pudessem configur e executar consultas complexas sem escrever SQL. A configuração da pesquisa/ *query* decorre num único ecrã composto por várias áreas que substituem de forma visual os diversos comandos da construção de uma pesquisa SQL (SELECT, ORDER BY, etc). Posto isto, os principais passos para a criação de uma *query* são: escolher os tipos de entidade (que em um esquema relacional correspondem às tabelas), selecionar as propriedades a apresentar (as colunas), definir os filtros quando necessário, e, por fim, executar. A ideia é manter o foco no que se pretende obter e deixar a tradução técnica para o sistema.

Antes de apresentar a interface completa, importa falar onde se localiza a componente de pesquisa dinâmica no DISME. No menu lateral esquerdo, na secção "Componentes", existe o bloco "Pesquisa Dinâmica". Ao clicar, surgem duas opções: "Nova Pesquisa", que abre o construtor com uma configuração em branco, e "Carregar pesquisas guardadas", onde são carregadas todas as pesquisas previamente guardadas para execução, edição ou remoção.

A Figura 16 apresenta a visão geral da interface no modo "Nova Pesquisa", com uma pesquisa simples configurada para ilustração. As anotações numeradas identificam as áreas de trabalho:

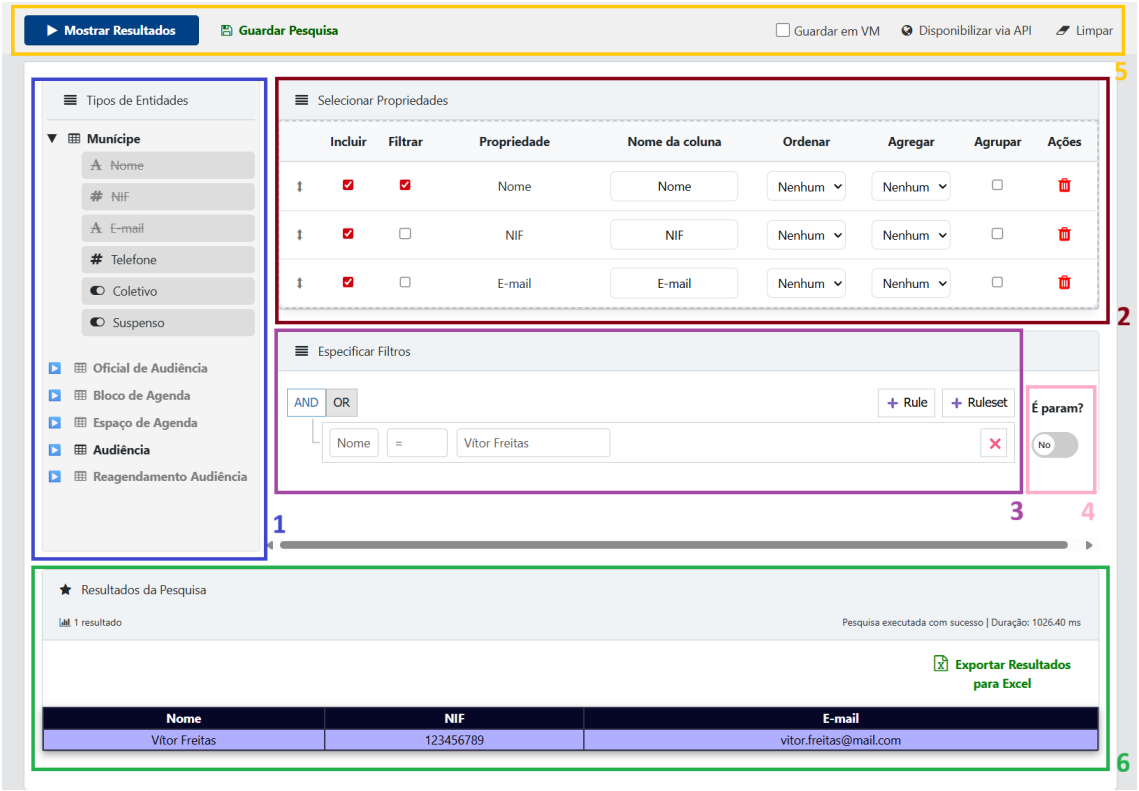


Fig. 16: Visão geral da interface da componente de pesquisa dinâmica: 1) seleção dos tipos de entidade, 2) seleção das propriedades (colunas), funções de agregação e *group by*, 3) especificar os filtros, 4) definir se o valor do filtro é parâmetro ou não, 5) barra de ações, 6) tabela de resultados

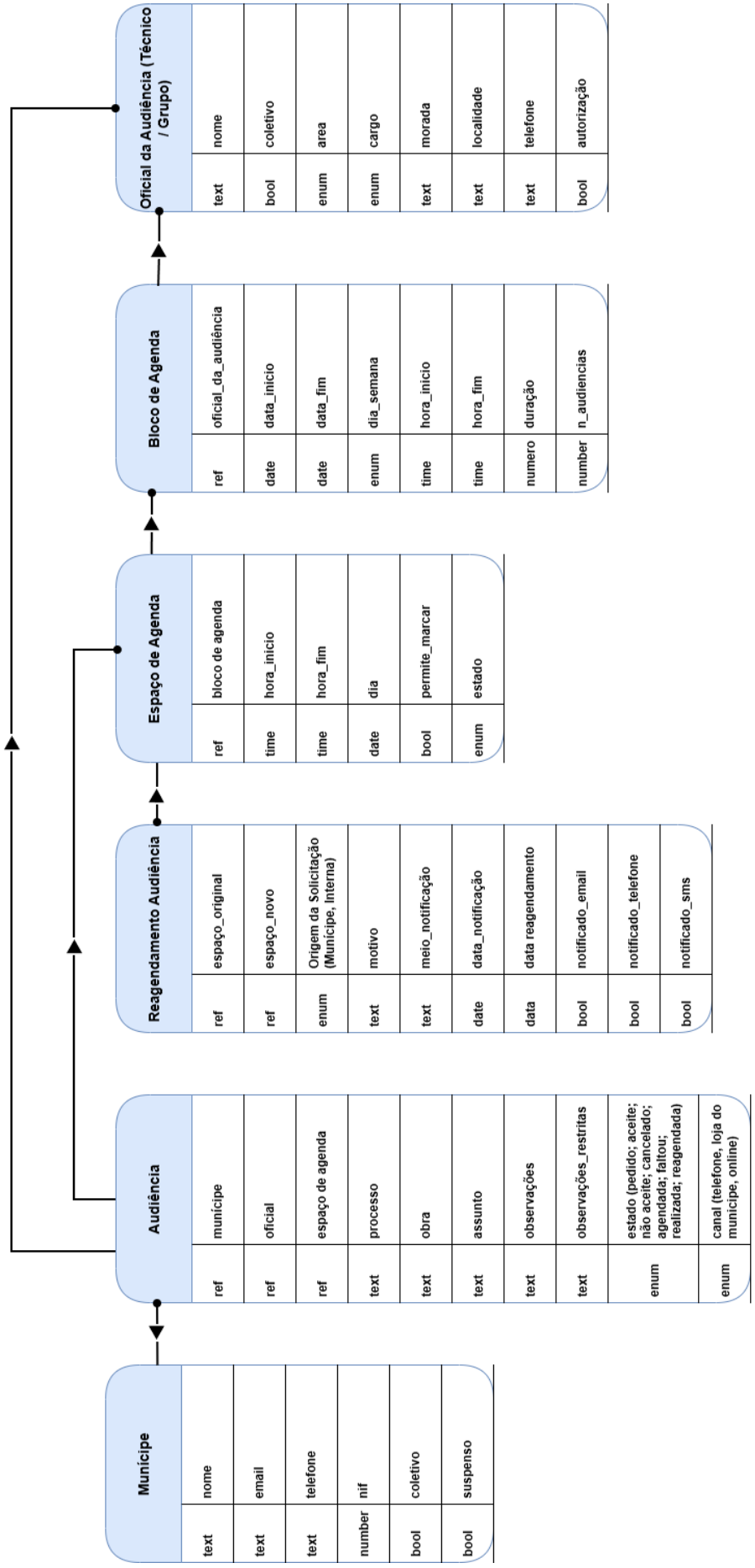


Fig. 17: Diagrama de Factos (*Fact Diagram*) da base de dados da Câmara Municipal do Funchal.

A seguir descreve-se, com detalhe, o processo completo de execução de uma *query* e a forma como as mesmas são guardadas e recuperadas, destacando o que o código faz em cada etapa para simplificar a experiência.

Antes de avançar, apresenta-se na Figura 17 o diagrama de factos (*fact diagram*) da base de dados da Câmara Municipal do Funchal, para que o leitor reconheça as entidades, as relações e as propriedades usadas nos exemplos seguintes. O diagrama centra-se na "Audiência", que liga o pedido do munícipe ao "Oficial da Audiência" responsável e ao "Espaço de Agenda" onde a marcação ocorre. A tabela "Munícipe" guarda os dados de contacto. O "Bloco de Agenda" define as datas de início e fim e a janela de horas e o "Espaço de Agenda" fixa o dia e o intervalo concreto usado. O "Reagendamento Audiência" regista alterações de marcação com o novo espaço, o motivo e as notificações. Sendo assim, para orientar a leitura de cada etapa, definem-se dois exemplos de referência que acompanharão a explicação:

Exemplo A (simples) - Lista de munícipes cujo nome contém Freitas:

- Tipos de entidade: "Munícipe", como tipo de entidade base.
- Propriedades: "Nome" e "E-mail" como colunas a incluir no resultado.
- Filtros: "Nome" com operador *contains* e valor "Freitas".
- Ordenação: ordenação por "Nome" ascendente.

Exemplo B (mais complexo) - Número de audiências por dia da semana e hora de início:

- Tipos de entidade: "Audiência" como tipo de entidade base, "Espaço de Agenda" e "Bloco de Agenda".
- Propriedades: COUNT("Processo") (de "Audiência"), "Dia da Semana" (de "Bloco de Agenda"), e "Hora de Início" (de "Espaço de Agenda") como colunas a incluir no resultado. E *group by* em "Dia da Semana" e "Hora de Início".
- Filtros: "Dia da Semana" igual a "Quinta-feira".
- Ordenação: ordenação por "Hora de Início" ascendente.

O Exemplo A é o fio condutor em todos os passos. O Exemplo B aparece apenas quando acrescenta comportamento que o Exemplo A não demonstra, mantendo o foco e evitando repetição.

1. **Selecionar os tipos de entidade (tabelas):**

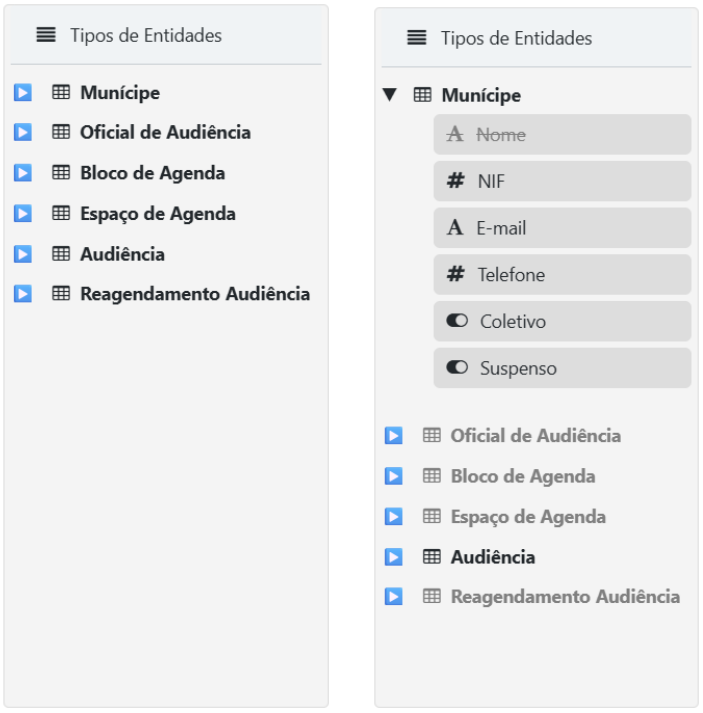
   Ao iniciar uma nova pesquisa, o sistema vai buscar à base de dados todos os tipos de entidades (*entity types*), suas respetivas propriedades (*properties*) e tipos de entidade que cada um referencia (a partir dos valores guardados em *fk_entity_type_id* das suas propriedades), pertencentes ao utilizador que pretende realizar a *query*.

   Cada tipo de entidade é apresentado no painel lateral esquerdo como um bloco expansível com as suas propriedades, como ilustra a Figura 18a). Para selecionar uma propriedade é necessário clicar no tipo de entidade a que essa propriedade pertence e arrastá-la para a área de propriedades. Quando isso acontece, o tipo de entidade a que a propriedade escolhida pertencia fica marcado como selecionado e mantém-se visível para continuar a escolha das restantes propriedades.

Quando existe apenas um tipo de entidade selecionado, esse tipo passa a ser a base da pesquisa. O identificador é guardado em `Query.base_ent_type_id`. Se o utilizador escolher propriedades de mais do que um tipo, o sistema procura o melhor candidato a tabela base através de um grafo de referências (a partir dos valores guardados em *fk_entity_type_id* de cada propriedade) e escolhe um tipo de entidade que consiga alcançar todos os outros. Se não existir um caminho válido, remove os tipos de entidade e propriedades que não são alcançáveis e volta a permitir a escolha, de propriedades que podem ser alcançadas a partir do tipo de entidade base.

No exemplo A, o utilizador clica em "Munícipe" e arrasta "Nome" para a área de propriedades. A interface fixa "Munícipe" como tipo de entidade base e regista `base_ent_type_id` com o id correspondente. A partir daí ficam visíveis as propriedades desse tipo e dos tipos de entidade alcançáveis a partir dele, ou que o referenciam (ver Figura 18b)). O utilizador acrescenta "E-mail" sem alterar a tabela base, uma vez que continua a escolher propriedades do mesmo tipo. Se remover todas as propriedades de "Munícipe", a base é limpa.

No exemplo B, a seleção faz-se de forma encadeada. O utilizador escolhe as propriedades a partir de "Audiência", navega no painel para "Espaço de Agenda" e segue para "Bloco de Agenda", respeitando o caminho disponível no modelo. A interface confirma que todas as propriedades são alcançáveis desde a base e mantém "Audiência" como tipo de entidade base no estado da pesquisa. Aqui é importante referir que também era possível selecionar as propriedades dos tipos de entidades "ao contrário" começando por "Bloco de Agenda" e terminando em "Audiência", e a tabela base continuaria sendo "Audiência". O que não era possível aqui é selecionar "Audiência" e depois "Bloco de Agenda" e vice versa.



a) b)

Fig. 18: Interface da pesquisa dinâmica: a) Tipos de entidade antes de qualquer seleção; b) Tipos de entidade após escolher uma propriedade de "Munícipe".

2. **Selecionar as propriedades (colunas) a apresentar no resultado:**

   A seleção das propriedades é o momento em que o utilizador define o que irá ser mostrado na tabela de resultados.

   A seleção de uma propriedade faz-se por *drag and drop*, arrastando-a a partir da área de Tipos de Entidades, mostrada no ponto 1, para a área que contém a tabela de propriedades, criando assim uma nova linha na mesma Cada linha da tabela expõe os controlos essenciais: incluir no resultado, disponibilizar como propriedade de filtro, editar o nome da coluna (*alias*), definir a direção de ordenação, selecionar a função de agregação e marcar a coluna para agrupar (`GROUP BY`). O nome da coluna assume por defeito o nome da propriedade. As agregações (COUNT, SUM, AVG, MIN, MAX) ficam disponíveis para tipos compatíveis, por exemplo, para o tipo de valor *text* apenas é permitido aplicar o COUNT.

   Caso o utilizador desmarque a inclusão (tornando a propriedade apenas um critério de filtro), as opções de nome da coluna, agregação e `GROUP BY` são desativadas para evitar configurações inválidas.

   A posição das linhas na tabela define a ordem de apresentação das propriedades na tabela final e também, a prioridade entre múltiplas ordenações. A reordenação faz-se por arrasto no símbolo presente no canto esquerdo de cada linha, com resposta visual imediata. Internamente, a lista `resultFields[]` mantém esta ordem, e todos os valores de cada propriedade.

   Do ponto de vista do artefacto lógico, o estado desta tabela é guardado num *array* de objetos com os campos `incProp`, `filterProp`, `alias`, `sortType`, `aggregate` e `groupBy`. A mesma estrutura é reutilizada quando a pesquisa é guardada ou materializada para execução posterior. As ações e respetivos efeitos no artefacto lógico estão sintetizados na Tabela 3.

Tabela 3: Ações na tabela das propriedades e efeito no artefacto lógico da pesquisa

Ação na UI	**Efeito no significado da pesquisa**	**Campo lógico afetado**
Adicionar (arrastar) propriedade	Passa a integrar a lista de propriedades na posição atual	`resultFields[]`
Marcar “Incluir”	A coluna é devolvida no resultado	`resultFields[].incProp`
Marcar “Só filtro”	A coluna não é mostrada no resultado, fica apenas disponível para filtros	`resultFields[].filterProp`
Editar nome da coluna	Ajusta o cabeçalho exibido no resultado	`resultFields[].alias`
Escolher ASC/DESC	Define a direção de ordenação da coluna	`resultFields[].sortType`
Reordenar linhas	Fixa ordem de colunas e prioridade entre múltiplas ordenações	ordem em `resultFields[]`
Selecionar agregação	Aplica função `COUNT/SUM/AVG/MIN/MAX`	`resultFields[].aggregate`
Assinalar “Group By”	Adiciona a coluna à cláusula `GROUP BY`	`resultFields[].groupBy`
Remover coluna	Deixa de estar incluída nos filtros e resultados	remoção em `resultFields[]`

Continuando os exemplos anteriores, no Exemplo A, com "Munícipe" como tipo de entidade base, o utilizador arrasta "Nome" e "E-mail" para a tabela de propriedades, define a ordenação ascendente (ASC) na propriedade "Nome" e a tabela passa a refletir as duas propriedades como incluídas no resultado, sem agregações nem `GROUP BY`, como se observa na Figura 19.

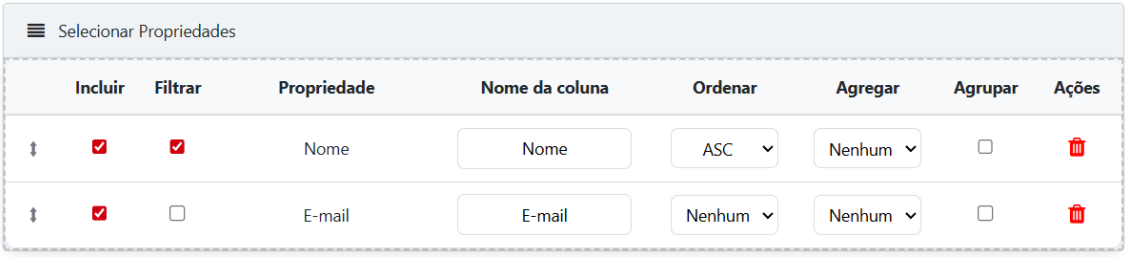


Fig. 19: Exemplo A: Seleção e configuração das propriedades.

E relativamente ao Exemplo B, com "Audiência" como base, o utilizador adiciona as propriedades "Processo" "Hora de Início" e "Dia da Semana" à tabela de propriedades. Como o objetivo é saber o número de audiências por dia da semana e hora de início, seleciona-se a função de agregação COUNT para "Processo" e GROUP BY (agrupar) para as restantes duas propriedades. A configuração final e a ordem aplicada às colunas são visíveis na Figura 20.

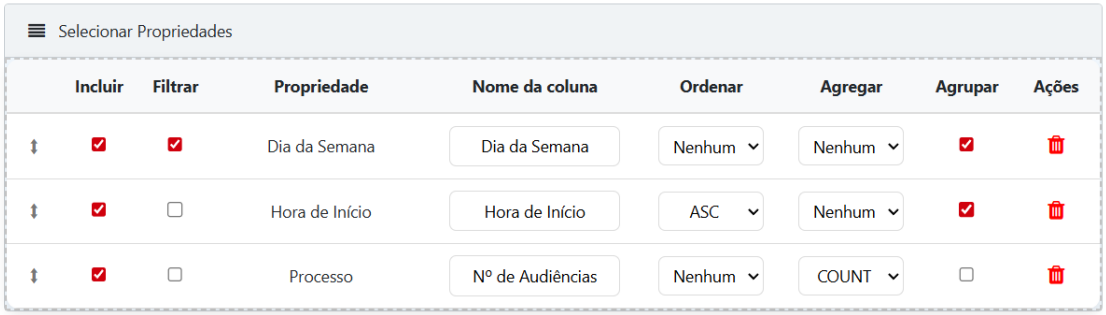


Fig. 20: Exemplo B: Seleção e configuração das propriedades.

3. **Definir os filtros no *query builder*:**

É neste momento que definem-se as entidades que deverão ser consideradas no resultado final. O editor organiza a condição numa árvore lógica n-ária composta por *rules* e *rulesets*, ligados através de operadores `AND` ou `OR`. Cada *ruleset* funciona como um nó interno que pode agrupar um número variável de regras ou de subconjuntos de regras, e cada *rule* corresponde sempre a uma folha. Esta estrutura hierárquica deixa a precedência clara por si só e evita ambiguidades na interpretação das condições. Não se trata de uma árvore binária, porque cada bloco pode conter tantas regras quantas o utilizador necessitar, o que ajusta o formato aos cenários reais de pesquisa e mantém a interface simples. Para prevenir qualquer confusão, o *query builder* só disponibiliza as propriedades que foram marcadas no passo anterior como filtros. Desta forma, o utilizador mantém sempre um conjunto de campos que é consistente com a pesquisa que está a realizar.

Cada *rule* segue o formato: propriedade–operador–valor. Os operadores são validados com base no tipo da propriedade, minimizando erros e tornando a configuração mais rápida:

- **Texto**: =, !=, *contains*.
- **Numéricos, data e hora**: =, !=, <, <=, >, >=.
- **Enumerações e propriedades de referência**: =, !=, *in*, *not in*.
- **Booleanos**: 0, 1.

A versão atual do *query builder* não inclui os operadores "entre", "é nulo" ou "não é nulo", nem operadores "começa por" ou "termina em". Intervalos constroem-se compondo duas regras dentro do mesmo grupo, por exemplo >= e <=.

De forma a preservar a consistência visual e semântica, o campo onde é inserido o valor adapta-se ao tipo de propriedade a que se refere. Sendo assim, para as propriedades do tipo *text* o campo do valor aceita texto livre, para as do tipo *enum*, ou *prop_ref* é apresentado uma *selecbox* com os valores permitidos, para as datas e horas surgem calendários e relógios, e por fim para as propriedades do tipo *bool* uma *checkbox* é suficiente.

Cada *rule* pode ser marcada como parâmetro através do botão *toggle* presente à sua frente, na coluna lateral direita, passando a registar `isParameter = true` e respetivo `ruleNumber` no artefacto lógico do *query builder*. Este mecanismo sinaliza que o valor do filtro será fornecido no momento de execução por quem executa a pesquisa. A forma de passagem desses valores por integração REST é documentada nas subsecções seguintes.

Sendo assim, internamente, a árvore de filtros é transformada num JSON onde cada *rule* guarda o id da propriedade, o operador, o valor, se é parâmetro (`isParameter`) e o seu respetivo número (`ruleNumber`), cujo valor é simplesmente a ordem em que o mesmo aparece no *query builder*.

No Exemplo A, com "Munícipe" como tabela base, o utilizador adiciona uma condição sobre "Nome", escolhe o operador *contains* e introduz o valor "Freitas". O *ruleset* resultante tem como condição `AND`, mas também poderia ser `OR`, porque o resultado final seria o mesmo, já que é apenas um filtro, como pode-se observar na Figura 21.

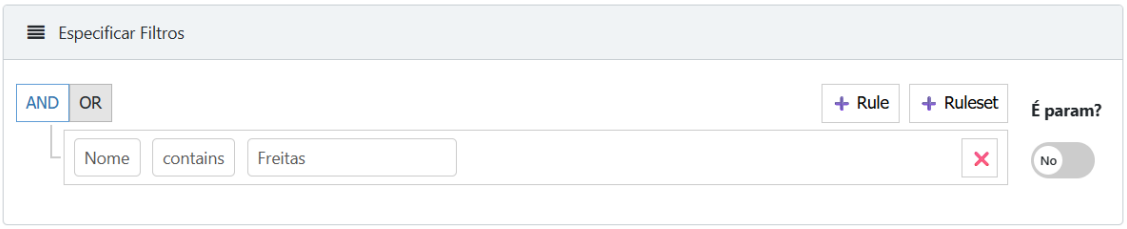


Fig. 21: Exemplo A: *query builder* com o filtro "Nome contém Freitas".

No Exemplo B, com "Audiência" como tabela base, define-se um filtro único sobre a propriedade "Dia da Semana" com operador = e valor "Quinta-feira". As opções de valores aparecem em uma *selectbox* porque "Dia da Semana" é uma propriedade do tipo *enum* (uma vez que só tem sete valores possíveis).

4. **Executar a pesquisa:**

   A execução é direta e dá retorno imediato na própria página.

   No topo, a barra de ações concentra os botões com as funções principais da componente: executar a pesquisa, guardar a configuração na base de dados, guardar os resultados em uma vista materializada (botão "Guardar em VM"), disponibilizar os resultados em interface REST API e a opção para recomeçar a pesquisa do zero. Estes botões estão ativos apenas quando faz sentido, por exemplo, o botão de execução é desativado se não existir pelo menos uma coluna incluída na tabela de propriedades, prevenindo execuções vazias.

   Ao clicar em "Mostrar Resultados" (botão azul com o ícone "*play*"), a interface mostra um *spinner* de carregamento e, quando termina, apresenta a tabela de resultados, com o número de linhas e a duração em ms da execução no cabeçalho.

   O pedido para a API leva o estado completo da *query*: o tipo de entidade que serve de base para a pesquisa (`base_ent_type_id`), os campos do resultado (`resultFields`), o `queryBuilder` e outros valores importantes. Na interface, o "Guardar em VM" é uma *checkbox* que modifica o valor de `Query.saveMV`. Para exemplificar, tem-se o Exemplo A que manda o pedido da seguinte forma:

```
{
  "selectedEntityTypes": [],
  "filterProperties": {
    "1": [2]
  },
  "includedProperties": {
    "1": [2, 4]
  },
  "resultFields": [
    {
      "entTypeId": 1, "propertyId": 2, "propertyName": "Nome", "incProp": true,
      "filterProp": true, "alias": "Nome", "sortType": "ASC", "aggregate": "None",
      "groupBy": false
    },
    {
      "entTypeId": 1, "propertyId": 4, "propertyName": "E-mail",
      "incProp": true, "filterProp": false, "alias": "E-mail", "sortType": "None",
      "aggregate": "None", "groupBy": false
    }
  ],
  "queryBuilder": {
    "condition": "and",
    "rules": [
      {
        "field": 2, "operator": "contains", "value": "Freitas", "entity": "1",
        "isParameter": false, "ruleNumber": 1
      }
    ]
  },
  "base_ent_type_id": 1,
```

```
  "saveMV": false
}
```

No servidor, o `DynSearchController@getQueryResults` constrói a pesquisa sobre o modelo EAV, operando sobretudo nas tabelas *"entity"*, *"property"* e *"value"*, aplica os filtros configurados no *query builder* e só seleciona as propriedades marcadas para serem incluídas no resultado. O resultado é então reordenado para respeitar a ordem das propriedades, agregado quando há agregações/`groupBy`, ordenado conforme `sortType` e, no final, os *aliases* são aplicados ao cabeçalho antes de retornar. Deste modo, a execução segue a ordem típica do SQL e produz o mesmo resultado ao de uma consulta SQL tradicional.

A resposta do Exemplo A que chega à interface é a seguinte:

```
{
    "success": true,
    "data": {
        "header": ["Nome", "E-mail"],
        "resultRows": [
            ["Ana Sofia Freitas", "anasofiafreitas@mail.com"],
            ["André Freitas Correia", "andrefreitascorreia@mail.com"],
            ["Beatriz Freitas Silva", "beatrizfreitassilva@mail.com"],
            ["Bruno Freitas Sousa", "brunofreitassousa@mail.com"],
            ["Catarina Freitas", "catarinafreitas@mail.com"],
            ["Helena Freitas Martins", "helenafreitasmartins@mail.com"],
            ["Inês Freitas Pires", "inesfreitaspires@mail.com],
            ["João Pedro Freitas", "joaopedrofreitas@mail.com"],
            ["Luís Freitas Gomes", "luisfreitasgomes@mail.com"],
            ["Maria Freitas Rodrigues", "mariarodrigues@mail.com"],
            ["Ricardo Freitas Almeida", "ricardofreitasalmeida@mail.com"],
            ["Sílvia Freitas Nunes", "silviafreitasnunes@mail.com"],
            ["Tiago Miguel Freitas", "tiagomiguelfreitas@mail.com"],
            ["Vítor Freitas", "vitor.freitas@mail.com"]
        ]
    },
    "message": null,
    "blocked": false
}
```

E é mostrada ao utilizador numa tabela simples e colorida, sem paginação, como pode-se observar na Figura 22.

Resultados da Pesquisa

14 resultados

Pesquisa executada com sucesso | Duração: 1238.10 ms

Exportar Resultados para Excel

Nome	E-mail
Ana Sofia Freitas	anasofiafreitas@mail.com
André Freitas Correia	andrefreitascorreia@mail.com
Beatriz Freitas Silva	beatrizfreitassilva@mail.com
Bruno Freitas Sousa	brunofreitassousa@mail.com
Catarina Freitas	catarinafreitas@mail.com
Helena Freitas Martins	helenafreitasmartins@mail.com
Inês Freitas Pires	inesfreitaspires@mail.com
João Pedro Freitas	joaopedrofreitas@mail.com
Luís Freitas Gomes	luisfreitasgomes@mail.com
Maria Freitas Rodrigues	mariarodrigues@mail.com
Ricardo Freitas Almeida	ricardofreitasalmeida@mail.com
Sílvia Freitas Nunes	silviafreitasnunes@mail.com
Tiago Miguel Freitas	tiagomiguelfreitas@mail.com
Vítor Freitas	vitor.freitas@mail.com

Fig. 22: Exemplo A: resultado da pesquisa.

E ao executar o Exemplo B temos o resultado abaixo:

Resultados da Pesquisa

6 resultados

Pesquisa executada com sucesso | Duração: 1260.30 ms

Exportar Resultados para Excel

Dia da Semana	Hora de Início	Nº de Audiências
Quinta-feira	09:30	3
Quinta-feira	10:15	2
Quinta-feira	11:00	1
Quinta-feira	12:30	1
Quinta-feira	13:15	1
Quinta-feira	14:00	2

Fig. 23: Exemplo B: resultado da pesquisa.

No final, a Figura 24 oferece uma visão geral do Exemplo B ao reunir, num só ecrã, a configuração e o respetivo resultado. Mostra as propriedades selecionadas, os filtros aplicados e as regras de ordenação, bem como a agregação por grupos com contagem do número de audiências. A tabela de resultados inclui a indicação do tempo de execução, confirmando a coerência entre o que foi configurado e o resultado obtido.

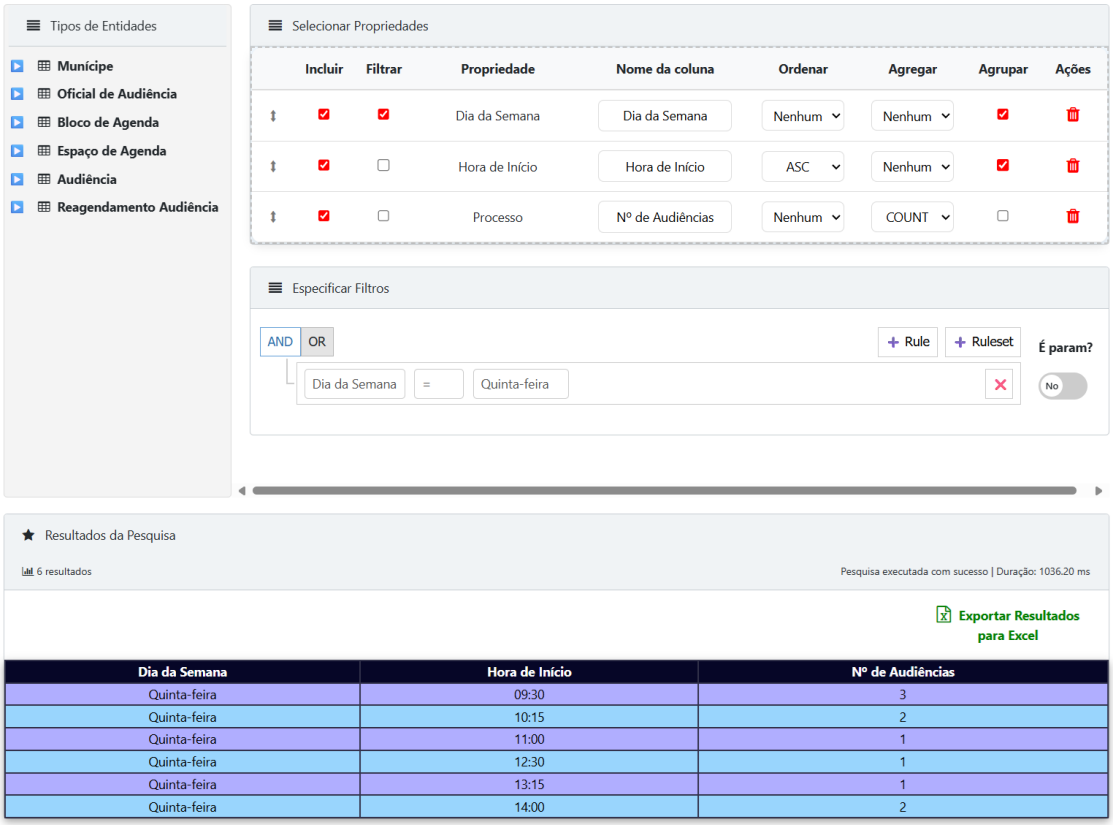


Fig. 24: Exemplo B: visão geral da pesquisa.

Após receber o resultado, se o mesmo devolver mais de 100 linhas *ou* se o *toggle* "Guardar em VM" estiver ativo, o servidor guarda/atualiza uma vista materializada. Esta decisão é sempre realizada no servidor após a execução.

5. **Exportar os resultados:**

A interface disponibiliza a exportação dos resultados para Excel em `.xlsx`. Ao clicar em "Exportar Resultados para Excel", começa automaticamente o *download* de um ficheiro com o nome Results.xlsx. O documento contém uma única folha com o cabeçalho na primeira linha, exatamente como em `header`, e cada elemento de `resultRows` é escrito numa linha seguinte. A largura das colunas é ajustada automaticamente ao conteúdo. Valores em falta são exportados como células vazias e não é aplicada formatação regional adicional, por isso números e datas são exportados tal como chegam do servidor.

Para fechar o ciclo, a Figura 25 mostra a exportação para Excel do resultado do Exemplo A.

	A	B	C	D	E	F	G
1	Nome	E-mail					
2	Ana Sofia Freitas	anasofiafreitas@mail.com					
3	André Freitas Correia	andrefreitascorreia@mail.com					
4	Beatriz Freitas Silva	beatrizfreitassilva@mail.com					
5	Bruno Freitas Sousa	brunofreitassousa@mail.com					
6	Catarina Freitas	catarinafreitas@mail.com					
7	Helena Freitas Martins	helenafreitasmartins@mail.com					
8	Inês Freitas Pires	inesfreitaspires@mail.com					
9	João Pedro Freitas	joaopedrofreitas@mail.com					
10	Luís Freitas Gomes	luisfreitasgomes@mail.com					
11	Maria Freitas Rodrigues	mariarodrigues@mail.com					
12	Ricardo Freitas Almeida	ricardofreitasalmeida@mail.com					
13	Sílvia Freitas Nunes	silviafreitasnunes@mail.com					
14	Tiago Miguel Freitas	tiagomiguelfreitas@mail.com					
15	Vítor Freitas	vitor.freitas@mail.com					
16							

Fig. 25: Exemplo A: resultado exportado para Excel.

6. **Guardar para reutilizar:**

Guardar uma pesquisa transforma uma configuração momentânea num artefacto reutilizável, pois permite que a mesma seja executada e editada quantas vezes forem necessárias.

No modo normal, depois de configurar a pesquisa no ecrã, clica-se em "Guardar Pesquisa" na barra de ações. A aplicação abre automaticamente o *modal* "Guardar Pesquisa", que solicita apenas o nome da pesquisa que irá aparecer na tabela com a lista de *queries* guardadas (Figura 26). Confirmado que o nome não possui caracteres especiais, o sistema prepara o pedido com o estado completo da *query* e envia um `POST/queries` para a API. Em caso de sucesso, a interface mostra uma notificação de sucesso e fecha o *modal*, caso contrário, apresenta a mensagem de erro correspondente.

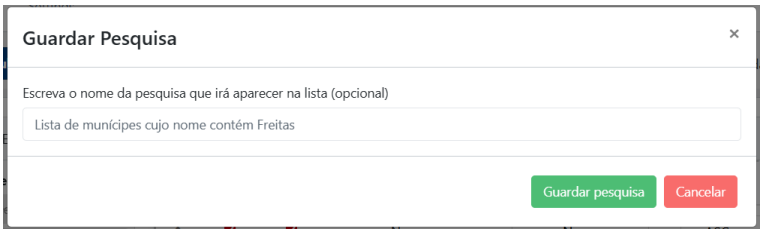


Fig. 26: Exemplo A: *Modal* "Guardar Pesquisa"

No servidor, o `QueryController@store` valida a inexistência de nomes duplicados, e guarda os valores conforme descrito no Capítulo 5.4. A resposta regressa com a consulta pronta a aparecer na tabela de pesquisas guardadas.

A tabela de pesquisas guardadas pode ser acedida através do separador "Carregar pesquisas guardadas" no menu lateral esquerdo. A tabela oferece pesquisa por texto e ordenação por colunas. Cada linha apresenta o nome da pesquisa com um resumo automático das propriedades e filtros (o que permite reconhecer o propósito sem a abrir), quando a pesquisa foi criada e

atualizada, e as ações disponíveis (Mostrar/Editar e Apagar), como é possível constatar na Figura 27.

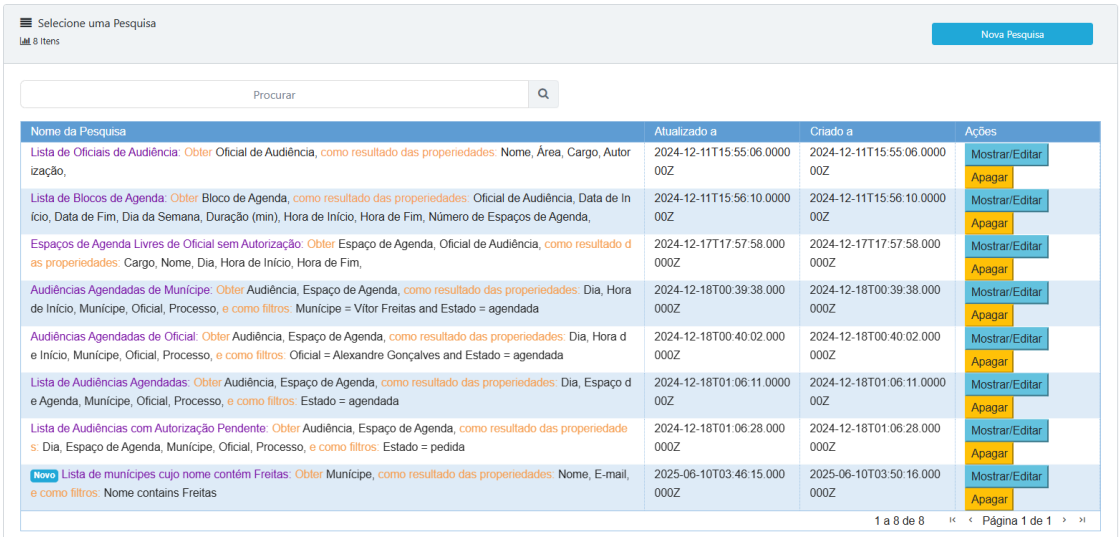


Fig. 27: Tabela de pesquisas guardadas com pesquisa por texto e ações rápidas.

Reabrir uma pesquisa refaz exatamente o que foi guardado. A interface carrega as `includedProperties`, os `resultFields`, as `filterProperties`, os `propertyParameters`, os *entity types* selecionados e o `queryBuilder` original, permitindo executar de imediato ou continuar a edição.

A barra superior adapta as ações apresentadas, substituindo a opção de "Guardar", pelas opções "Guardar como nova" e "Atualizar". Ao clicar em "Guardar como nova", o sistema faz um `POST/queries` e mantém a anterior, já ao clicar em "Atualizar", chama `PUT/queries/{id}` e, no servidor, é criada uma nova versão, os *endpoints* associados a essa *query* são removidos, ajustam-se ligações existentes para a versão atual e a versão anterior é desativada por *soft delete*.

Eliminar uma pesquisa é uma ação que exige confirmação por parte do utilizador, através do *modal* para esse fim. Confirmada a remoção da pesquisa, a interface chama `DELETE/queries/{id}`. No estado atual, a API realiza *soft delete* do registo e não aplica bloqueios por dependências. A Figure 28 mostra o diálogo de confirmação.

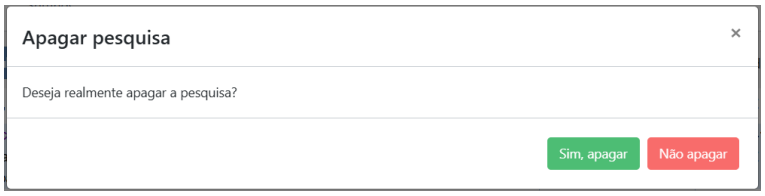


Fig. 28: Confirmação de eliminação de pesquisa.

6.4.2 Disponibilização de resultados de *query* por REST API

Por questões de praticidade, nesta secção tratam-se em conjunto dois objetivos do trabalho: o desenvolvimento de interfaces para disponibilização de resultados de *query* por REST API e o desenvolvimento de funcionalidade para as *queries* receberem parâmetros de forma dinâmica. A ideia é simples, a mesma pesquisa que o utilizador constrói e valida na componente pode ser publicada como *endpoint* `GET` e chamada por clientes externos, devolvendo JSON estável no formato `header/resultRows`. O contrato é o mesmo que a UI consome, o que reduz acoplamento e evita mapeamentos adicionais.

Na interface, a ação de publicar encontra-se na barra superior do ecrã de pesquisa. Ao clicar em "Disponibilizar via API", abre-se automaticamente um *modal*, onde o utilizador encontra no topo uma pré-visualização da rota `GET` já com o padrão da URL e um campo para definir o identificador do *endpoint* (*Query Action*). À medida que o utilizador escreve, a URL é atualizada em tempo real. Existe ainda um campo de descrição opcional que não interfere no contrato técnico. O *modal* organiza as opções em três separadores, que podem ser visualizadas na Figura 29:

- Em "Parâmetros" é mostrada uma tabela com três colunas: nome original do filtro, nome a expor na API e valor corrente de referência. O utilizador pode renomear cada parâmetro e cada alteração reflete-se de imediato na URL pré-visualizada. Se a pesquisa não tiver parâmetros, o *modal* informa o utilizador de forma explícita.
- Em "Autorização" escolhe-se o prefixo da rota, público (`dynamic/query/`) ou protegido com *Bearer Token* (`auth/dynamic/query/`).
- Em "Teste" são introduzidos valores simulados para cada parâmetro e é possível executar um teste que devolve o corpo de resposta, apresentado no próprio modal para conferência rápida.

No rodapé, o botão "Guardar Endpoint" realiza validações para impedir identificadores vazios e nomes de parâmetros em falta, e só então envia o pedido para guardar na base de dados. Em caso de sucesso, o *modal* muda para o estado de confirmação e exibe a *URL final* pronta a usar. Todo este fluxo segue o padrão `/api/{auth|}dynamic/query/{nome}?param=` e evita decisões ambíguas ao guardar a URL canónica com nomes de parâmetros e valores vazios, o que garante a identificação inequívoca do *endpoint*.

O consumo externo faz-se por `GET` para a URL publicada. Se o *endpoint* incluir *auth* a chamada requer autenticação e herda o contexto da linguagem do utilizador. O controlador reconstrói a URL a partir do pedido, encontra o registo na tabela "*rest_api*", recupera a definição guardada e aplica os valores recebidos nos filtros assinalados como parâmetros. A execução é efetuada da mesma forma abordada no objetivo anterior, e portanto, o resultado regressa à aplicação cliente num JSON simples com `header` e `resultRows`, exatamente como a interface espera.

Este desenho permite evoluir a capacidade de integração sem perturbar a experiência na UI. Quem publica não sai do fluxo natural de construção da pesquisa. Quem executa recebe sempre o mesmo formato compacto e estável.

Para demonstrar o funcionamento, retoma-se o Exemplo A, da secção anterior, e marcamos o filtro "Nome contém Freitas" como parâmetro, mantendo “Freitas” como valor por omissão. Ao abrir o *modal* de publicação, a interface carrega a forma corrente da pesquisa e identifica automaticamente os filtros elegíveis para parametrização, que neste caso, é apenas um sobre a propriedade "Nome". Define-se a "Ação da pesquisa" como "municipes-por-nome", o nome público

do parâmetro ("Nome - Rule 1") como "nome", e o valor por defeito surge já preenchido com "Freitas", herdado do *query builder*. A imagem abaixo mostra este estado inicial do *modal* com a pré-visualização da URL e a lista de parâmetros:

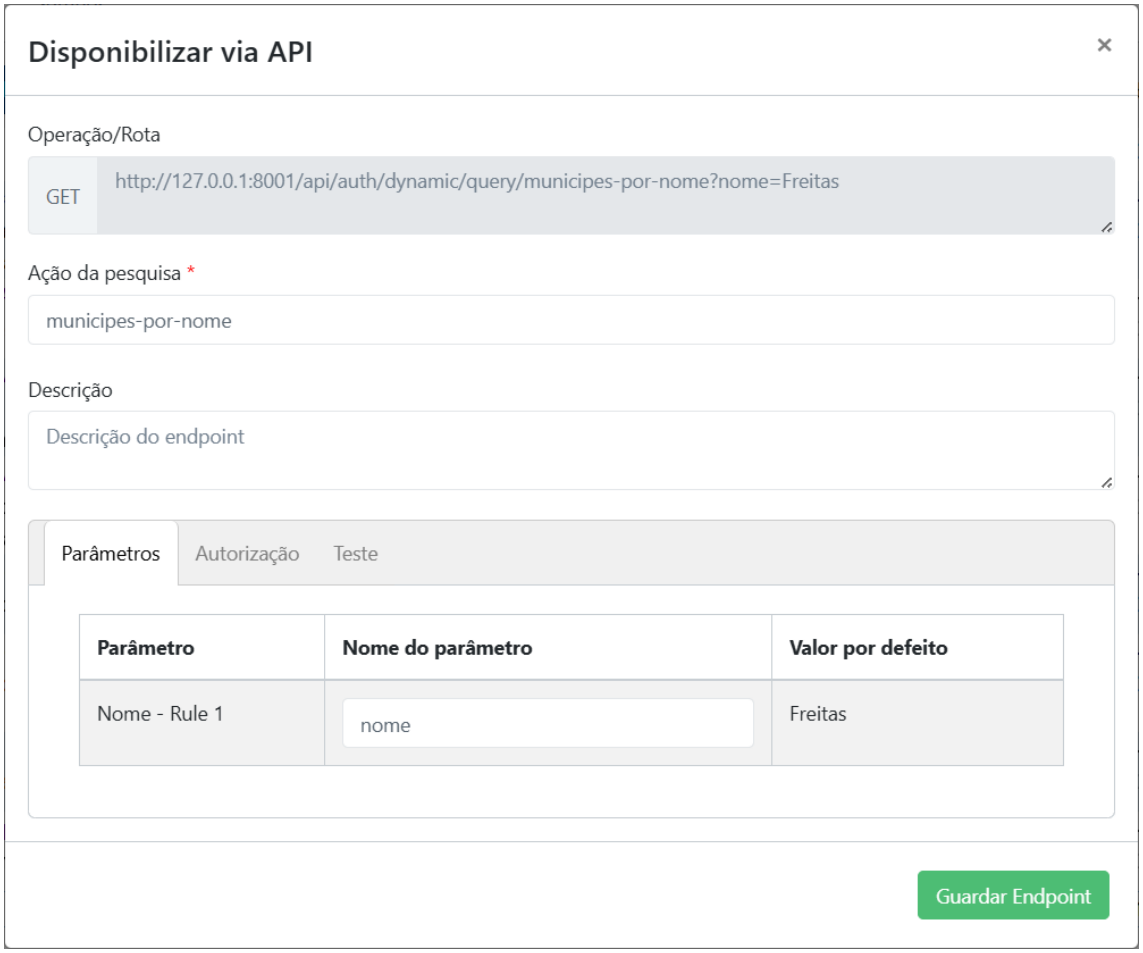


Fig. 29: *Modal* REST: vista geral com pré-visualização da URL e lista de parâmetros.

No separador de autorização escolhe-se entre *endpoint* protegido ou público. Para este exemplo manteve-se protegido, o que resulta numa rota do tipo `/api/auth/dynamic/query/{nome}`.

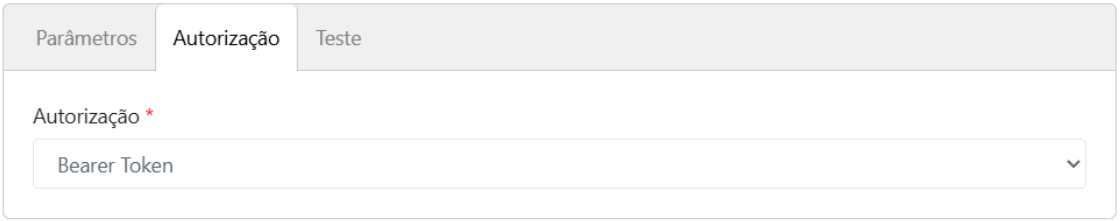


Fig. 30: Separador "Autorização": escolha da opção "Bearer Token".

No separador de teste, introduz-se valores simulados para os parâmetros e pede-se a execução da pesquisa carregando no botão "Teste". Aqui, mantém-se `nome=Freitas` como valor por omissão. O *modal* apresenta o corpo da resposta em formato JSON para conferência rápida, garantindo que o *endpoint* criado retorna valores válidos.

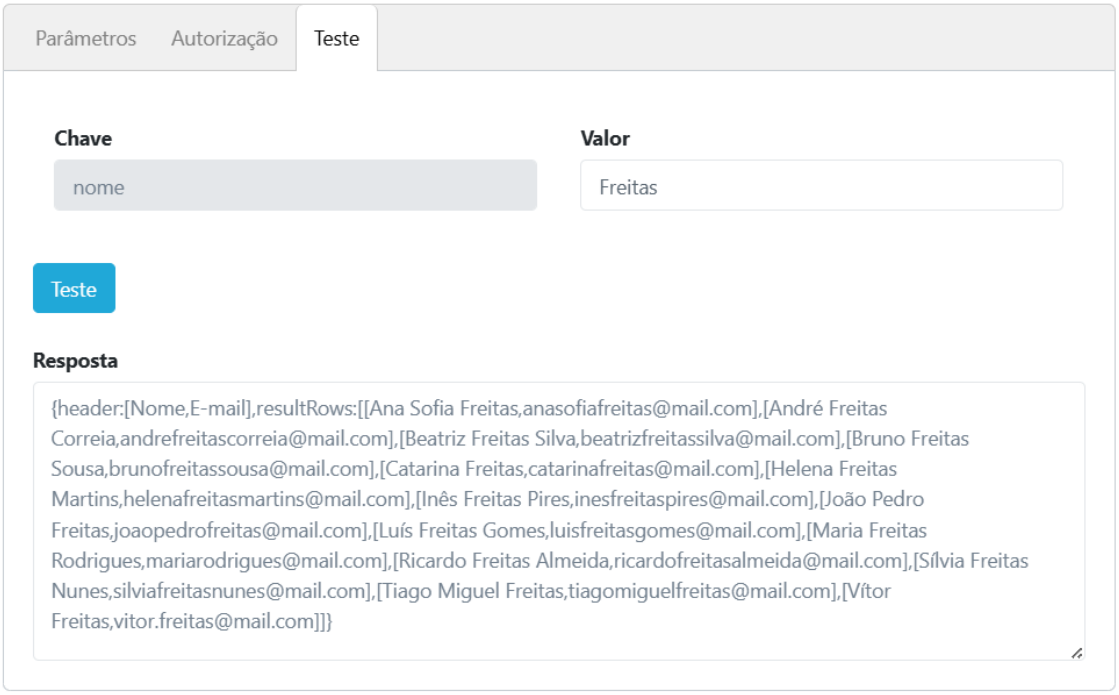


Fig. 31: Separador "Teste": simulação local com `nome=Freitas` e pré-visualização da resposta.

Validada a resposta, guarda-se o *endpoint* clicando no botão "Guardar Endpoint". A aplicação envia `POST /api/dyn-search/save-url` e o servidor guarda o registo na tabela *"rest_api"* com a cópia da *query* e a URL guardada numa forma estável onde os parâmetros aparecem com valores vazios. Este detalhe é o que permite identificar o *endpoint* sem depender de valores concretos. A interface confirma a criação e mostra a URL final pronta para usar, por exemplo `/api/auth/dynamic/query/lista-municipe?nome=`.

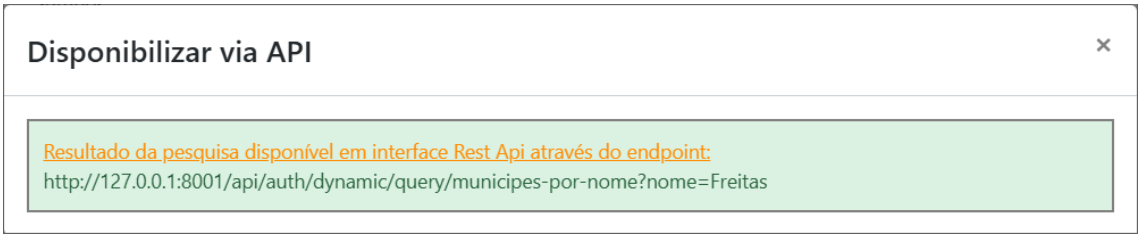


Fig. 32: Confirmação de publicação: URL final registada e pronta a consumo.

Com o *endpoint* criado, passou-se ao teste externo no POSTMAN. Aqui decidiu-se trocar o valor do parâmetro para verificar que a parametrização está a funcionar corretamente e que a execução

respeita a mesma forma de pesquisa, mantendo tudo igual, apenas alteramos `nome=Almeida` na *query string*. A chamada usa `GET` e *token* de acesso.

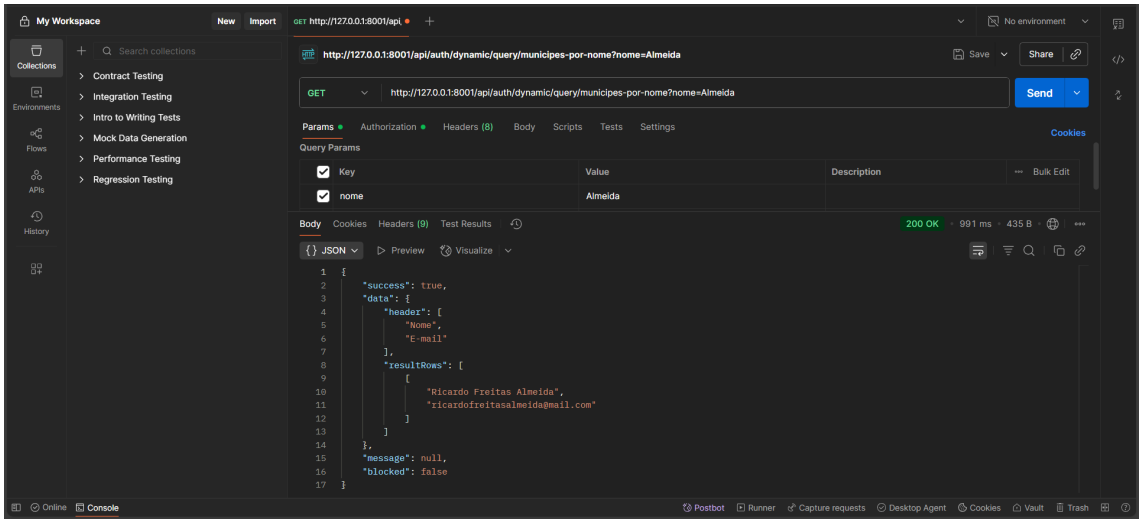


Fig. 33: Teste no Postman com `nome=Almeida`.

6.4.3 Integração com MongoDB e vistas materializadas

A integração com MongoDB tem como objetivo reduzir os tempos de resposta em pesquisas repetidas, principalmente com grande número de linhas. Não migrou-se o núcleo relacional nem alterou-se o modelo EAV, apenas acrescentou-se uma coleção dedicada para registrar resultados materializados no formato `header`/`resultRows`.

Antes de integrar, necessitou-se de instalar o MongoDB no projeto seguindo os passos presentes aqui [59]. Para isso, começou-se pela extensão nativa do PHP para MongoDB `ext-mongodb v1.13.0`. No ambiente Windows com XAMPP descarregou-se o `php_mongodb.dll` compatível com a versão do PHP. Copiou-se o ficheiro para `xampp/php/ext` e ativou-se em `php.ini` com `extension=php_mongodb`. Reiniciou-se o Apache e confirmou-se a presença do módulo tanto com `php -m` no terminal como em `phpinfo()` no navegador.

Seguiu-se a instalação das bibliotecas de aplicação. No PhpStorm executou-se `composer require mongodb/mongodb:^1.12` e `composer require jenssegers/mongodb:^3.5`, que estende o Eloquent para trabalhar com coleções MongoDB lado a lado com os *models* MySQL do DISME.

No MongoDB Atlas criou-se um *cluster* de desenvolvimento, definiu-se um utilizador de base de dados apenas com permissões necessárias e abriu-se o acesso de rede ao endereço da máquina de desenvolvimento.

Depois disto, configurou-se a ligação em `config/database.php`. Criou-se uma conexão `mongodb` sem mexer na ligação `mysql` por defeito:

```
'mongodb' => [
    'driver' => 'mongodb',
    'dsn' => env('MONGO_DB_URI', ''),
    'database' => env('MONGO_DB_DATABASE', 'forge'),
]
```

Os parâmetros sensíveis ficaram no `.env`. Como a base de dados está guardada no MongoDB Atlas usou-se a *connection string* `mongodb+srv`:

```
MONGODB_URI=mongodb+srv://<user>:<password>@<cluster>.mongodb.net/?retryWrites=true&w=
    majority
MONGODB_DATABASE=disme
```

Do lado da UI nada muda em termos de semântica. Se o resultado tiver mais de cem linhas ou se a opção tiver sido assinalada na *checkbox* "Guardar em VM", o sistema guarda os resultados em uma vista materializada. Esta regra foi intencionalmente simples, pois apenas pretende-se materializar uma pesquisa quando o custo de reconstrução no EAV poderá ser alto, ou quando o utilizador assim o desejar. A materialização acontece em paralelo e não introduz variações no resultado que o utilizador vê.

O `MaterializedViewController` recebe o `Request` e os resultados já estruturados, e guarda-os na coleção "*materialized_views*" no MongoDB. Abaixo, pode-se constatar um exemplo de documento guardado contento a vista materializada dos resultados do Exemplo A:


```
{
  "_id": {"$oid":"68c2fd614b2018824e09be62"},
  "user": 2,
  "base_ent_type_id": 1,
  "query_builder": {"condition":"and","rules":[{"field":2,"operator":"contains","value":"
    Freitas","entity":"1","isParameter":false,"ruleNumber":1}]},
  "aggregations": [
    {"entTypeId":1,"propertyId":2,"propertyName":"Nome","incProp":true,"filterProp":1,"
    alias":"Nome","sortType":"ASC","aggregate":"None","groupBy":0},
    {"entTypeId":1,"propertyId":4,"propertyName":"E-mail","incProp":true,"filterProp":0,"
    alias":"E-mail","sortType":"None","aggregate":"None","groupBy":0}
  ],
  "data": {
    "header": ["Nome","E-mail"],
    "resultRows": [["Ana Sofia Freitas","anasofiafreitas@mail.com"],["André Freitas
    Correia","andrefreitascorreia@mail.com"],["Beatriz Freitas Silva","
    beatrizfreitassilva@mail.com"],["Bruno Freitas Sousa","brunofreitassousa@mail.com"],["
    Catarina Freitas","catarinafreitas@mail.com"],["Helena Freitas Martins","
    helenafreitasmartins@mail.com"],["Inês Freitas Pires","inesfreitaspires@mail.com"],["
    João Pedro Freitas","joaopedrofreitas@mail.com"],["Luís Freitas Gomes","
    luisfreitasgomes@mail.com"],["Maria Freitas Rodrigues","mariarodrigues@mail.com"],["
    Ricardo Freitas Almeida","ricardofreitasalmeida@mail.com"],["Sílvia Freitas Nunes","
    silviafreitasnunes@mail.com"],["Tiago Miguel Freitas","tiagomiguelfreitas@mail.com"],[
    "Vítor Freitas","vitor.freitas@mail.com"]]
  },
  "updated_at": {"$date":"2025-06-10T03:48:33.000Z"},
  "created_at": {"$date":"2025-06-10T03:48:33.000Z"}
}
```


Para tornar a gestão das vistas materializadas simples e previsível, definiu-se que uma vista só pode ser reutilizada quando o utilizador autenticado, o tipo de entidade base, o *query builder* e o conjunto de agregações mantêm-se iguais. Esta opção elimina ambiguidades e dispensa heurísticas difíceis de explicar. Qualquer diferença na definição gera uma nova vista, por exemplo mudar a

ordem das colunas faz com que a aplicação trate a pesquisa como distinta. O impacto desta decisão no armazenamento é reduzido à escala do protótipo, por isso privilegia-se a previsibilidade.

Para assegurar que esta política é cumprida na prática, o sistema realiza uma validação no servidor antes de guardar cada nova vista materializada, confirmando se já existe uma configuração idêntica. Quando todos os elementos coincidem, o resultado existente é devolvido imediatamente, evitando duplicados e processamento desnecessário. Esta validação garante que apenas alterações reais originam novas vistas, mantendo a coleção organizada, protegendo a integridade da informação e assegurando que cada materialização se mantém eficiente e alinhada com a forma como o DISME utiliza estas pesquisas.

Contudo pretende-se introduzir políticas de retenção para controlar o volume, como um índice de TTL sobre `updated_at` que apague automaticamente vistas antigas após um período definido.

6.4.4 Tabelas relacionais como alternativa de execução

As tabelas `db_et_{id}` oferecem uma forma derivada e fixa por tipo de entidade (guardado no EAV), para acelerar leituras na interface e complementar as vistas materializadas em MongoDB, sem substituir o EAV que permanece como a base de dados principal e único local de escrita. Mantém-se o mesmo formato JSON enviado para a UI com `header` e `resultRows`. No EAV a informação é guardada nas tabelas *"entity"*, *"property"*, *"value"*, etc e cada pesquisa precisa de realizar inúmeros *joins*, o que acrescenta custo. As `db_et_{id}` organizam os dados de maneira convencional com uma linha por entidade e colunas estáveis por propriedade nomeadas a partir do id das mesmas como `p2` para "Nome" ou `p4` para "Email", reduzindo junções e simplificando pesquisas frequentes.

Existe uma tabela por tipo de entidade e por língua como `db_et_1_pt` e `db_et_1_en`. O nome das tabelas e das colunas mantêm-se estáveis, mesmo com alterações na base de dados, uma vez que são escritas a partir de ids que nunca mudam. A geração destas tabelas é feita percorrendo as tabelas do EAV e inserindo os respetivos valores já com o nome correto. Por exemplo, para as propriedades do tipo *enum*, o valor guardado na tabela *"value"* é o id do valor correspondente na tabela *"prop_allowed_value"*, com estas tabelas relacionais o valor é colocado diretamente na tabela já traduzido. Desta forma, guarda-se os valores como texto por defeito e só converte-se para número ou data quando é preciso filtrar ou ordenar, sem necessidade de alterar o esquema da base de dados. Tal como no EAV, a ordenação, as agregações, agrupamentos e *aliases* são aplicados no final da execução, devolvendo sempre à UI o mesmo formato de resultado, independentemente do caminho.

A atualização faz-se por funções do lado do servidor que removem e criam cada tabela a partir do EAV. Estas tabelas foram adicionadas apenas como uma forma de entender se é uma solução viável para o DISME, como uma solução intermédia entre o EAV e as vistas materializadas, e não pertencem ao fluxo normal do sistema. Por outras palavras, para testar esta abordagem é preciso ativá-la manualmente no código do projeto.

6.4.5 Validação da segurança

A componente de pesquisa dinâmica do DISME foi desenvolvida com uma atenção sistemática à segurança, procurando assegurar que o utilizador pudesse construir pesquisas flexíveis sem expor o sistema aos riscos associados à manipulação de SQL. Esta preocupação tem fundamento claro, pois

a geração dinâmica de consultas é uma das áreas mais sensíveis de qualquer sistema de informação e continua a estar na origem de vulnerabilidades críticas como a injeção de SQL. Por essa razão, a segurança foi integrada desde o início no desenho da solução e não tratada como um elemento secundário.

A injeção de SQL continua a ser uma das falhas mais exploradas porque é simples de executar e pode ter impacto grave. O atacante tenta colocar código SQL num campo que a aplicação julga ser apenas um valor e, em sistemas que constroem *queries* por concatenação de *strings*, basta algo tão simples como escrever "OR 1=1" para transformar um filtro legítimo numa condição sempre verdadeira. Desta forma, uma consulta que deveria ser por exemplo `username = "john"` passa a `username = "john" OR 1=1` e a consulta deixa de filtrar, revelando que o problema não é apenas técnico mas estrutural na forma como a *query* é construída.

O DISME evita este risco logo na forma como o utilizador interage com o sistema. Não existe qualquer ponto onde seja possível escrever SQL, já que todas as escolhas são feitas visualmente através de menus, listas de colunas e campos de valores controlados. A consulta é montada pelo sistema a partir de elementos conhecidos e validados e nunca de fragmentos escritos pelo utilizador, o que reduz de forma significativa a superfície de ataque e impede que instruções inesperadas entrem na construção da *query*.

A geração do SQL está a cargo do Laravel e do seu ORM Eloquent, que garante que todos os valores são parametrizados. Primeiro define-se a estrutura da *query* e só depois se associam os dados através de parâmetros, o que evita que qualquer valor seja tratado como código. Mesmo que alguém tente inserir texto semelhante a SQL, esse texto é sempre processado como dado. Nos poucos casos em que foi necessário usar SQL manual, manteve-se a regra de que todos os valores passam obrigatoriamente pelos *bindings* do Laravel e nunca são concatenados diretamente.

Para confirmar a consistência deste comportamento foram feitos testes práticos onde se introduziram valores malformados, sequências típicas de ataques reais e combinações improváveis de operadores. O objetivo foi verificar se a API mantinha um comportamento previsível e se continuava a bloquear qualquer tentativa de manipulação. A validação no *frontend*, o conjunto reduzido de operadores e a construção programática das *queries* mostraram ser suficientes para impedir qualquer vetor de injeção de SQL.

A segurança estende-se ao lado da API, onde todas as operações sensíveis como criar, atualizar ou remover pesquisas exigem autenticação. A segurança estende-se também ao lado da API porque todas as operações sensíveis como criar, atualizar ou remover pesquisas requerem que o utilizador esteja autenticado. O tratamento de erros é simples e segue uma lógica consistente. O único caso distinguido de forma explícita é o conflito de nomes de pesquisas, que origina um código 409 sempre que o utilizador tenta criar ou atualizar uma pesquisa com um nome já existente. Perante qualquer outra exceção o sistema regista o erro, reverte a operação e devolve o código 500. Este código fica reservado para situações inesperadas no servidor e evita expor detalhes internos que poderiam comprometer a segurança.

O resultado é um ambiente seguro e previsível, onde o utilizador pode construir pesquisas de forma visual e intuitiva, enquanto o sistema garante que essa liberdade não abre espaço a comportamentos maliciosos. O equilíbrio entre flexibilidade e segurança guiou toda a implementação desta componente e está presente desde o desenho até à validação final.

6.5 Comparação da componente de pesquisa dinâmica face ao estado da arte

Esta subsecção apresenta a comparação da componente de pesquisa dinâmica do DISME com o estado da arte, procurando mostrar de forma clara como esta abordagem se distingue das soluções oferecidas por plataformas como OutSystems, Mendix e Skyvia, analisadas no Capítulo 3. Todas estas plataformas procuram reduzir a barreira técnica, permitindo que utilizadores sem conhecimentos de SQL construam consultas completas através de interfaces visuais e intuitivas. O DISME segue esta mesma linha, mas adapta essas práticas a um contexto organizacional específico, garantindo integração direta com os dados e processos empresariais.

Entre as soluções analisadas, o Skyvia serviu de inspiração direta para o *visual query builder* do DISME, sobretudo pela simplicidade, pelo fluxo claro de construção de consultas e pela capacidade de gerar *queries* dinâmicas sem programação. Tal como o Skyvia, o DISME utiliza o mecanismo de *drag-and-drop* e organiza a criação de consultas em etapas lógicas: o utilizador adiciona os campos pretendidos, aplica filtros e define agregações ou ordenações. O sistema suporta *alias*, agrupamentos e múltiplas funções de agregação, refletindo práticas comuns nos construtores modernos.

A diferença principal é que o Skyvia opera maioritariamente sobre fontes externas e assume a lógica de integração de dados como objetivo central, o DISME trabalha diretamente sobre o metamodelo da organização. As entidades que o utilizador vê na interface são as mesmas que sustentam processos, regras e transações reais dentro da plataforma. Por isso, uma consulta criada no *builder* pode ser usada de imediato em *dashboards*, fluxos ou parâmetros de decisão, já que o sistema não se limita a explorar dados, integrando-os na operação da plataforma.

A publicação de consultas através de *endpoints* REST também evidencia esta integração. Cada *endpoint* no DISME é gerado para atender a uma necessidade concreta, com controlo sobre campos expostos e parâmetros definidos pelo utilizador, tornando a integração com outros sistemas mais direcionada e eficiente do que nas três plataformas mencionadas, que geralmente expõem coleções de dados mais genéricas.

A materialização de consultas segue igualmente as tendências atuais, mas com um modelo adaptado à realidade *low-code*. O DISME usa o MongoDB para criar vistas materializadas de forma dinâmica, escolhida pelo próprio utilizador. Quando uma consulta é pesada e repetida, basta um clique para materializar os resultados. As execuções seguintes passam a ser quase instantâneas. Este mecanismo, normalmente reservado a administradores de bases de dados, é aqui colocado nas mãos do utilizador final, permitindo otimizar o desempenho sem profundos conhecimentos técnicos.

A parametrização reforça esta flexibilidade. Qualquer filtro pode ser transformado num parâmetro, permitindo reutilizar a mesma consulta com valores diferentes. Esta abordagem aproxima o DISME do que ferramentas como o Power BI [14] promovem, evitando a duplicação de consultas similares e dando margem para automatizar cenários recorrentes.

Apesar destas aproximações ao estado da arte, persistem limitações. Não existe pré-visualização incremental de resultados, nem gráficos automáticos, e as junções estão limitadas às relações diretas do modelo de dados.

Em termos de usabilidade, o DISME oferece uma curva de aprendizagem reduzida tanto para utilizadores que conhecem o modelo de dados da organização tanto para aqueles que não possuem nenhuma experiência na área. Sendo assim, o DISME é uma solução que não tenta competir pelo

[14] Power BI – `https://app.powerbi.com/`

lado mais técnico do estado da arte, mas sim pela clareza, pela ligação ao contexto de negócio e pela utilidade prática no dia a dia da organização.

6.6 Reprodutibilidade do protótipo

Para replicar o protótipo no desenvolvimento local:

1. Clonar o repositório a partir do GitHub e configurar o `.env` com credenciais de MySQL e MongoDB Atlas.
2. Instalar dependências com `composer install` e `npm install`.
3. Ativar a extensão `ext-mongodb` em `php.ini` e confirmar com `php -m`.
4. Importar a base de dados no MySQL e executar as rotinas que criam as tabelas relacionais `db_et_{id}_pt` e `db_et_{id}_en`.
5. Gerar a chave da aplicação com `php artisan key:generate`.
6. Iniciar o *backend* em `http://127.0.0.1:8001` com `php artisan serve --port=8001`.
7. Iniciar o *frontend* com `ng serve` ou compilar para produção com `ng build`.
8. Validar a ligação ao Atlas no MongoDB Compass, e confirmar se `materialized_views` existe.

7 Avaliação e Resultados

Este capítulo avalia a componente de pesquisa dinâmica e discute o impacto dos resultados na plataforma. Interessa perceber se a solução cumpre o que promete na prática e se a experiência de utilização é clara para quem chega sem grande familiaridade com estas ferramentas. A avaliação segue uma progressão simples. Primeiro confirma-se o funcionamento ponta-a-ponta por testes de integração. Depois medem-se tempos de resposta e variabilidade em três cenários representativos de execução de consultas. Por fim, recolhe-se evidência de usabilidade com *think aloud* e com o *System Usability Scale*, para completar a leitura técnica com a perceção dos utilizadores.

Ao longo do desenvolvimento realizaram-se iterações curtas com testes frequentes. Usaram-se sobretudo os esquemas das bases de dados da Câmara Municipal do Funchal e do projeto MiColec (que é um projeto do EELab com o objetivo de aprimorar um sistema colaborativo de *micro-hub* logístico com economia circular, logística inversa e teoria dos jogos [60]), com dados artificiais controlados para reproduzir padrões reais de seleção de propriedades, filtragem, ordenação e agregação. Em paralelo, recolheu-se *feedback* informal de colegas, o que ajudou a detetar incongruências e a melhorar textos, mensagens e comportamentos da interface.

7.1 Testes de integração

O objetivo destes testes é verificar a integração entre a interface, a API REST e as bases de dados e confirmar que o que foi implementado cumpre a arquitetura e os requisitos. Sendo assim, os cenários de teste cobriram quatro blocos essenciais da solução: configuração de pesquisas, parâmetros dinâmicos, publicação por REST e vistas materializadas.

Como ambiente de teste, usou-se a aplicação a correr localmente, com a base de dados relacional MySQL para os dados de origem e o MongoDB como repositório de vistas materializadas para *cache*. Carregou-se um conjunto estável de dados para verificação funcional e um conjunto maior para medir tempos.

De forma resumida, os cenários principais de testes foram os seguintes:

- **Configuração e execução:** criar, atualizar, duplicar e eliminar pesquisas, aplicar filtros e ordenações, usar agregações com *group by*, executar e ver a duração.
- **Parâmetros dinâmicos:** marcar filtros como parâmetros, validar nomes e tipos, definir valores por defeito e obrigatoriedade, garantir a mesma ordem do *query builder*.
- **Publicação por REST:** gerar o URL, testar o *endpoint* a partir da interface, escolher a autorização, confirmar que o JSON tem `header` e `resultRows` iguais aos da execução local.
- **Vistas materializadas:** guardar manualmente, reutilizar quando a pesquisa é igual, expirar por tempo, atualizar manualmente, criar automaticamente quando o resultado tem mais de cem linhas.

A componente cumpriu os objetivos de integração. A seguir apresenta-se a síntese dos resultados por cenário:

- **Configuração e execução:** A listagem apresentou nome, descrição e datas, com pesquisa por texto e ordenação operacionais. A criação exigiu nome, tipo de entidade e pelo menos uma propriedade, o que evitou configurações incompletas. A atualização manteve referências corretas. A duplicação criou entradas independentes. O *query builder* bloqueou filtros inválidos.

As agregações funcionaram quando o *group by* estava coerente. A duração apareceu sempre no cabeçalho.

- **Parâmetros dinâmicos:** Os nomes foram validados, os valores respeitaram o tipo, a obrigatoriedade impediu a execução quando faltou preencher e a ordem apresentada seguiu a do *query builder*.
- **Publicação por REST:** O URL gerado respeitou o padrão definido. As respostas foram JSON válido. As datas surgiram em ISO 8601 em UTC e a codificação foi UTF-8. Chamadas sem credenciais falharam como previsto. Chamadas autenticadas devolveram o mesmo `header` e as mesmas `resultRows` que a interface.
- **Vistas materializadas:** Guardar a vista criou o documento com cabeçalho e linhas. Repetir a mesma pesquisa reutilizou a vista e produziu o mesmo resultado. A expiração por tempo forçou recomputação na execução seguinte. A atualização manual substituiu o conteúdo anterior. Quando o resultado ultrapassou cem linhas ocorreu criação automática e a partir daí a leitura foi feita da vista.
- **Erros e mensagens:** As mensagens foram consistentes e ajudaram a corrigir a configuração. Por exemplo quando faltou um parâmetro obrigatório, o *endpoint* listou os parâmetros esperados. A interface manteve-se responsiva após erros.

Para além dos cenários funcionais, estes testes cobriram aspetos não funcionais definidos no Capítulo 5.1.2 como desempenho, segurança, interoperabilidade, fiabilidade, manutenção, interoperabilidade e observabilidade. Segue a síntese do que ficou verificado no contexto dos testes de integração:

- **Desempenho e eficiência:** verificou-se que a duração apresentada no cabeçalho ajudou a comparar execuções repetidas e a confirmar o efeito das vistas materializadas.
- **Segurança de acesso:** os *endpoints* protegidos exigiram *Bearer Token* e as chamadas sem credenciais falharam com mensagens claras.
- **Interoperabilidade:** confirmou-se JSON válido, codificação UTF-8 e datas em ISO 8601 em UTC. As respostas respeitaram a semântica de HTTP.
- **Fiabilidade e consistência:** as mensagens de erro foram uniformes e a interface manteve-se responsiva. Em atualizações, as referências mantiveram-se corretas.
- **Manutenibilidade:** a separação por camadas UI, serviços REST e dados simplificou a repetição dos testes e o isolamento de problemas.
- **Internacionalização e acessibilidade:** os nomes e mensagens apresentaram-se na língua configurada e mantiveram coerência entre UI e API.
- **Observabilidade:** ficaram registados *logs* de chamadas REST e erros com contexto mínimo para diagnóstico e recolha de métricas técnicas.

Os testes de integração confirmam que a componente está pronta para uso em contexto real. Do lado funcional, a criação e a execução de pesquisas mostraram-se estáveis, a publicação por REST funcionou e as vistas materializadas reduziram o tempo de resposta em consultas repetidas. Do lado não funcional, verifica-se desempenho estável, segurança adequada, boa compatibilidade entre sistemas, comportamento fiável, facilidade de manutenção e visibilidade suficiente para diagnóstico.

7.2 Testes de *performance*

Para medir a *performance*, fez-se a comparação da latência entre os três modelos de dados implementados no projeto: EAV como base de dados principal, leitura em tabelas relacionais e leitura a partir de vistas materializadas em MongoDB. Os testes foram realizados num computador pessoal, equipado com processador AMD Ryzen 5 5500U, 8 GB de RAM, disco SSD de 512 GB e sistema operativo Windows 11, garantindo consistência e comparabilidade entre execuções.

A latência foi sempre medida da mesma forma. Executou-se a pesquisa na interface e registou-se o valor apresentado no cabeçalho no momento em que a tabela de resultados fica visível ("Pesquisa executada com sucesso | Duração: X ms"), o que assegura comparabilidade entre execuções.

Em cada cenário, a pesquisa foi repetida seis vezes, descartando-se a primeira execução de forma a eliminar efeitos de arranque. A partir das restantes execuções, foi calculada a média aritmética e o desvio padrão, permitindo quantificar não apenas o tempo médio de resposta, mas também a sua variabilidade. O desvio padrão reflete a consistência das medições: valores reduzidos indicam tempos de execução estáveis entre repetições, enquanto desvios mais elevados evidenciam maior variabilidade, normalmente associada a consultas mais complexas ou à necessidade de múltiplas junções entre tabelas. Usaram-se três cenários para cobrir graus distintos de complexidade e volume de resultados observados na prática. No cenário simples, incluíram-se três propriedades sem filtros e a tabela devolveu poucas linhas. No cenário intermédio, efetuou-se a junção entre dois tipos de entidade e aplicaram-se dois filtros. No cenário avançado, selecionou-se uma propriedade de cada um dos seis tipos de entidade e acrescentou-se ordenação. Em cada cenário, a mesma *query* foi executada nos três modelos para permitir uma comparação justa. No caso das vistas materializadas, confirmou-se que a leitura provinha do documento previamente guardado em JSON e não de uma recomputação e manteve-se o ambiente constante durante as medições para evitar interferências.

Observou-se consistência entre repetições e diferenças claras entre as três abordagens. A leitura a partir de vistas materializadas em MongoDB apresentou o melhor desempenho, com tempos médios por consulta próximos de 600 ms. A leitura em tabelas relacionais ficou em valores intermédios, em torno de 1100 ms. A execução em EAV foi o mais lenta, com média perto de 1700 ms, variando de cerca de 900 ms em consultas simples até cerca de 3000 ms nas mais complexas. Em consultas com mais linhas verificou-se agravamento da latência no EAV, enquanto em MongoDB os tempos se mantiveram próximos. Isto é compatível com a natureza de cada modelo, porque o EAV exige várias junções entre entidades, propriedades e valores, e o modelo documental lê de um único documento. A Tabela 4 resume os valores observados.

Tabela 4: Síntese da latência média e desvio padrão por cenário e por abordagem, medida na interface

	EAV	**Relacional**	**MongoDB com VM**
Cenário simples	939 ± 32 ms	854 ± 91 ms	486 ± 76 ms
Cenário intermédio	1146 ± 124 ms	962 ± 168 ms	563 ± 120 ms
Cenário avançado	2984 ± 185 ms	1514 ± 190 ms	761 ± 157 ms
Média	**1690 ms**	**1110 ms**	**603 ms**

Relativamente aos desvios padrão, no cenário simples, o EAV apresentou o menor desvio (32 ms), refletindo execuções mais previsíveis, enquanto o relacional atingiu 91 ms e o MongoDB 76 ms, indicando que, apesar da sua latência média mais baixa, ainda existe alguma variabilidade. À medida que a complexidade aumenta, o desvio padrão cresce em todos os casos: no cenário intermédio, 124 ms no EAV, 168 ms no relacional e 120 ms no MongoDB, resultado do aumento da carga de processamento, *joins* complexos e volume de dados. No cenário avançado, os desvios sobem para 185 ms, 190 ms e 157 ms respetivamente, mostrando que o MongoDB mantém uma previsibilidade relativamente melhor em cenários complexos, enquanto EAV e relacional apresentam maior dispersão devido à complexidade das *queries* e à estrutura das tabelas. Estes resultados destacam que, embora o MongoDB não seja o mais consistente em cenários simples, a sua arquitetura orientada a documentos oferece vantagens em cenários de maior complexidade, garantindo tempos de resposta mais estáveis e menos sujeitos a variações imprevisíveis.

A leitura dos resultados permite concluir que, quando as pesquisas são repetidas muitas vezes, envolvem várias junções entre tabelas ou incidem sobre volumes maiores com resultados em dezenas de linhas, promover a consulta a vista materializada em MongoDB reduz de forma visível a latência e mantém desvios padrão mais baixos nos cenários intermédios e avançados, o que revela maior consistência entre execuções nesses contextos. Em pesquisas simples, com poucas propriedades e poucos filtros, a leitura em tabelas relacionais oferece um equilíbrio estável entre simplicidade e tempo de resposta, com desvios padrão moderados, além de evitar o custo de manter materializações ativas. O EAV mantém o seu papel como fonte principal e como opção mais flexível para configurar consultas, mas revela limites de desempenho em cenários exigentes, por isso deve usar-se sobretudo em casos simples ou quando os resultados têm de estar sempre atualizados. Nesses casos a execução recorrente deve ficar a cargo de vistas materializadas sempre que o padrão de uso o justifique.

7.3 Testes de usabilidade

Os testes de usabilidade dividiram-se em seis fases: apresentação, familiarização com a componente de pesquisa dinâmica do DISME, execução de tarefas, questionário SUS, *feedback* final e conclusão.

As tarefas cobriram o essencial do percurso de pesquisa e foram definidas para utilizadores sem experiência prévia com *visual query builders*. Excluíram-se conceitos avançados como agregações, *group by* e materialização, por exigirem literacia técnica mais elevada e não serem necessários para avaliar aprendizagem inicial, previsibilidade das funcionalidades e clareza das mensagens. Esta escolha permitiu isolar o que é estrutural no uso diário, como selecionar propriedades, aplicar filtros, ordenar resultados, ativar parâmetros, guardar a pesquisa, voltar a abri-la e publicar por REST.

Sendo assim, realizou-se um teste de pequena escala com cinco participantes sem experiência prévia no DISME ou em *visual query builders*. Três estudantes do ensino superior e dois trabalhadores, um de área técnica e outro de área não técnica. As idades variaram entre dezoito e quarenta e quatro anos. A seleção incidiu propositadamente em utilizadores iniciantes para observar aprendizagem inicial, clareza das mensagens e previsibilidade do fluxo básico sem depender de literacia técnica avançada.

A recolha de dados combinou o método *think aloud* com o questionário *System Usability Scale* (SUS). No *think aloud* cada participante foi convidado a dizer em voz alta o que esperava ver, o que

encontrou e por que motivo escolheu cada opção, o que permitiu registar em tempo real mensagens pouco claras, momentos de hesitação e tentativas de correção [61]. O SUS forneceu uma medida simples e comparável da usabilidade percebida numa escala de zero a cem e foi aplicado no fim de cada sessão [62]. A combinação do *think aloud* com o SUS permitiu cruzar evidência qualitativa e quantitativa e deu uma leitura equilibrada sobre o que funciona e o que precisa de melhoria. Cada teste decorreu individualmente e presencialmente numa sala reservada, sem influências externas.

Na Tabela 5 apresentam-se as fases realizadas nos testes de usabilidade e a duração média de cada atividade. Cada sessão teve uma duração total média entre 35 e 45 minutos.

Tabela 5: Procedimentos dos testes de usabilidade.

Fase	**Tarefa**	**Duração média (min)**
Apresentação	DISME e objetivos da pesquisa dinâmica	3
	Apresentação das funcionalidades do ecrã de pesquisa dinâmica	2
Familiarização	Exploração livre da plataforma	4–5
Tarefas	1. Criar e executar uma pesquisa simples sem filtros	3
	2. Aplicar um ou mais filtros à pesquisa	4
	3. Ordenar uma das propriedades do resultado	1
	4. Ativar um filtro como parâmetro	1
	5. Guardar a pesquisa com um nome descritivo	2
	6. Procurar e reabrir a pesquisa guardada	3
	7. Publicar por REST e testar a URL gerada	4
SUS Google Forms	Questionário SUS	4
Feedback final	Comentários sobre a experiência, clareza das funcionalidades e mensagens	5–8
Conclusão	Agradecimento e encerramento da sessão	1
	Duração total (minutos)	**35–45**

Todos os participantes seguiram a mesma ordem das etapas do teste de usabilidade a seguir:

- **Apresentação:** Iniciou-se cada sessão com uma explicação breve do DISME, da componente de pesquisa dinâmica e o público alvo. Mostraram-se as áreas principais do ecrã de pesquisa e os tipos de ações disponíveis. No final da introdução, esclareceu-se a sequência de passos da sessão para que o participante soubesse o que iria acontecer a seguir.
- **Familiarização com a plataforma:** De seguida, o participante ficou com o computador para explorar à vontade. O site estava a correr localmente e pediu-se que navegasse no ecrã de pesquisa durante alguns minutos sem guião. O participante dizia em voz alta o que esperava ver e o que ia encontrando.

- **Tarefas:** Terminada a exploração livre, passou-se às tarefas. Apresentaram-se uma a uma e só se avançou quando a anterior estava concluída. Não se impuseram valores concretos, podendo cada pessoa escolher os seus exemplos para evitar enviesamentos e para mostrar que a interface lida bem com pesquisas diferentes. Manteve-se o *think aloud* durante toda esta fase. As tarefas foram as seguintes:

  - **1. Criar e executar uma pesquisa simples sem filtros:** Pediu-se para o participante selecionar pelo menos três propriedades de um tipo de entidade à sua escolha e executar a pesquisa, confirmando se o cabeçalho reflete exatamente a seleção efetuada e se a tabela de resultados apresenta linhas coerentes com as propriedades escolhidas.

  - **2. Aplicar um ou mais filtros à pesquisa:** O participante acrescenta um ou mais filtros adequados às propriedades que escolheu, volta a executar e verifica se as linhas devolvidas se ajustam ao filtro definido.

  - **3. Ordenar uma das propriedades do resultado:** Aqui foi solicitado ao participante para escolher uma propriedade para ordenar no sentido que quisesse, para avaliar se a indicação visual do estado de ordenação é fácil de reconhecer.

  - **4. Ativar um filtro como parâmetro em execução:** Solicitou-se ao participante que transformasse pelo menos um dos filtros em parâmetro.

  - **5. Guardar a pesquisa com um nome descritivo:** O participante guarda a configuração atual com um nome válido para perceber se o fluxo de guardar é direto e se a mensagem de confirmação é suficiente.

  - **6. Procurar e reabrir a pesquisa guardada:** Foi pedido ao participante para usar a tabela de pesquisas para localizar a pesquisa criada por nome, abri-la no construtor e executar de novo.

  - **7. Publicar por REST e testar a URL gerada:** Solicitou-se ao participante para ativar a publicação por REST, copiar a URL do *endpoint* e testar no POSTMAN (aqui foi necessário ajudar o participante para indicar onde deveria colar o URL). Para além disso, pediu-se para testar a chamada alterando o valor de algum parâmetro.

- **Questionário SUS:** No final das tarefas, cada participante respondeu ao *System Usability Scale* num formulário em Google Forms. O questionário tinha quatro secções. Primeiro apresentava a informação essencial sobre o estudo, incluindo objetivo, procedimento, critérios de inclusão, riscos, benefícios, eventuais custos e regras de confidencialidade. Depois pedia o consentimento informado, com a explicação dos direitos e os contactos para esclarecimentos. Em seguida, recolhia dados demográficos (idade, género e situação profissional). Por fim, surgiam as dez questões do SUS que geram a pontuação de usabilidade.

  ***Feedback* final:** Depois do SUS, pediu-se um comentário livre sobre a experiência. Convidou-se o participante a referir o que funcionou bem e o que gerou dúvidas com foco em clareza dos textos da interface, previsibilidade das ações, mensagens de validação e perceção de desempenho.

  Conclusão: Para encerrar fez-se uma síntese breve do que seria feito com os resultados, agradeceu-se a participação e terminou-se a sessão.

Embora a amostra seja reduzida, esta revela-se adequada para um teste formativo, cujo objetivo passa por identificar os problemas mais relevantes e validar o percurso essencial de criar, executar, guardar, reabrir e publicar uma pesquisa. Estes cinco casos fornecem indicações suficientes sobre aspetos a melhorar, nomeadamente: clarificar textos, simplificar passos e reforçar mensagens de validação. A evidência recolhida durante o teste permite orientar o trabalho futuro, nomeadamente alargar a amostra a utilizadores com maior experiência em bases de dados, incluir tarefas que envolvam agregações, agrupamento (*group by*) e materialização de dados e repetir o estudo em contexto real, com utilização prolongada do sistema.

7.3.1 Resultados dos testes de usabilidade

Os sete passos do percurso essencial foram concluídos pelos cinco participantes, com autonomia crescente a partir da segunda tarefa. Ainda assim, emergiram pontos de fricção claros na descoberta dos resultados, na organização das propriedades, no modo de seleção por arrastar e largar e na publicação por REST. Estas observações são valiosas porque expõem dúvidas que surgem nos primeiros minutos de uso e orientam melhorias de desenho com potencial de impacto imediato.

7.3.1.1 Avaliação qualitativa

Inicialmente, a descoberta dos resultados não foi imediata. Vários participantes executaram a pesquisa e mantiveram o foco visual no topo do ecrã, à procura de alterações, sem perceber que a tabela de resultados aparecia mais abaixo, fora da área visível inicial. Como a tabela se situa abaixo da linha do ecrã, é necessário fazer *scroll* para a visualizar. Esta dificuldade manifestou-se principalmente nas tarefas 1 e 2, levando a repetições desnecessárias do botão "Executar", numa tentativa de confirmar se a consulta tinha sido bem-sucedida. O indicador presente no cabeçalho dos resultados com a mensagem "Pesquisa executada com sucesso | Duração: X ms" mostrou-se útil, mas apenas quando os participantes já tinham descido na página e visualizavam a tabela.

O seletor de propriedades foi descrito como "sobrecarregado", especialmente quando a entidade base continha um elevado número de propriedades. O painel apresentava demasiada informação em simultâneo, tornando a navegação por listas longas morosa. Na tarefa 1, a seleção inicial demorou mais do que o esperado, uma vez que os participantes não compreenderam, de imediato, o processo de escolha das propriedades.

A interação baseada no arrastar e largar para selecionar propriedades foi considerada pouco intuitiva. A maioria dos participantes tentou primeiro clicar para incluir uma propriedade, só depois percebendo que era necessário arrastá-la para a zona de resultados. Registaram-se erros de precisão na área de largar e momentos de hesitação sobre o destino correto. A perceção geral foi a de que um simples clique para incluir seria mais eficiente. Este padrão afetou a tarefa 1 e gerou frustração sempre que a ação de largar não era bem-sucedida à primeira tentativa.

Na segunda tarefa, dois participantes revelaram dificuldade em perceber como criar um filtro. Procuraram um campo direto com a designação "Adicionar filtro" e não identificaram que era necessário clicar no botão "+ Rule" do *query builder*. A descoberta do caminho correto ocorreu apenas após alguns segundos de exploração. Esta hesitação reforça a necessidade de tornar o *affordance* do botão mais explícito e de alinhar a terminologia utilizada com a linguagem esperada pelos utilizadores, por exemplo, através da adoção do termo "Adicionar filtro".

Na publicação por REST, surgiram duas dificuldades principais relacionadas com os campos do formulário e a descoberta de funcionalidades. Primeiro, o campo "Ação da pesquisa" não foi claro

para três participantes, uma vez que não perceberam o que deveriam escrever, avançando apenas através de tentativas e erro. Segundo, o campo para o nome do parâmetro passou despercebido nas primeiras utilizações. A exploração da funcionalidade foi assim guiada mais pela correção de erros do que por uma compreensão antecipada do funcionamento.

Foram também registados aspetos positivos quando a tabela de resultados estava visível. O cabeçalho com o número de resultados e o tempo de execução transmitiu uma sensação de controlo aos utilizadores, permitindo comparar rapidamente diferentes execuções. A opção de exportar para Excel foi igualmente valorizada, por permitir continuar a análise fora da plataforma.

De modo geral, a navegação foi bem avaliada quando os utilizadores se familiarizaram com a ferramenta. As fases de "Selecionar propriedades", "Especificar filtros" e "Ver resultados" estão bem organizadas na interface, e a tabela final é clara. Os principais obstáculos identificados ocorreram no início do processo de utilização. Quando estes foram superados, a execução das tarefas e a leitura dos resultados tornaram-se mais rápidas e eficazes.

Em síntese, os participantes consideraram a plataforma funcional e com potencial para utilização diária, mas identificaram três aspetos passíveis de melhoria ao nível da usabilidade. Em primeiro lugar, é necessário garantir que a tabela de resultados fique sempre visível após a execução, através de mecanismos como *auto scroll* ou a inclusão de um atalho "ver resultados". Em segundo lugar, foi sugerida a redução da carga visual no seletor de propriedades e de um agrupamento colapsável mais organizado. Por último, foi proposta a disponibilização de uma alternativa por clique para incluir propriedades, mantendo simultaneamente a opção de arrastar e largar, de modo a acomodar diferentes preferências dos utilizadores. Estas alterações apresentam um baixo risco técnico de implementação e alinham-se com os próximos passos planeados para o desenvolvimento da plataforma.

7.3.1.2 Avaliação quantitativa

A análise quantitativa recorreu ao *System Usability Scale* (SUS). O SUS contém dez afirmações avaliadas numa escala de *Likert* de 5 pontos (1="discordo totalmente" a 5="concordo totalmente"). Nas questões ímpares, valores altos são desejáveis, e nas pares o contrário. Após a recolha das respostas, aplica-se a conversão padrão do SUS: nas questões ímpares subtrai-se 1 da resposta $(Q-1)$ e nas questões pares $(5-Q)$. Soma-se tudo e multiplica-se a soma por 2,5 para obter uma pontuação final entre 0 e 100 [62]:

$$\begin{aligned}\text{SUS} = \Big((Q1-1)+(5-Q2)+(Q3-1)+(5-Q4)+(Q5-1)+(5-Q6)+(Q7-1)+(5-Q8)\\ +(Q9-1)+(5-Q10)\Big)\times 2{,}5\end{aligned}$$

O formulário usado encontra-se na Figura 34, e as respostas e respetivas pontuações na Tabela 6.

Na tabela abaixo assinale a resposta que melhor descreve a sua opinião sobre essa afirmação, onde: *

1 = Discordo totalmente
2 = Discordo
3 = Não concordo nem discordo
4 = Concordo
5 = Concordo totalmente

	1 Discordo totalmente	2	3	4	5 Concordo totalmente
Acho que gostaria de utilizar este sistema com frequência.	○	○	○	○	○
Considerei o sistema mais complexo do que necessário.	○	○	○	○	○
Achei o sistema fácil de utilizar.	○	○	○	○	○
Acho que necessitaria de ajuda de um técnico para conseguir utilizar este sistema.	○	○	○	○	○
Considerei que as várias funcionalidades deste sistema estavam bem integradas.	○	○	○	○	○
Achei que este sistema tinha muitas inconsistências.	○	○	○	○	○
Suponho que a maioria das pessoas aprenderia a utilizar rapidamente este sistema.	○	○	○	○	○
Considerei o sistema muito complicado de utilizar.	○	○	○	○	○
Senti-me muito confiante a utilizar este sistema.	○	○	○	○	○
Tive de aprender muito antes de conseguir lidar com este sistema.	○	○	○	○	○

Fig. 34: Questionário SUS realizado via Google Forms.

Tabela 6: Respostas ao SUS por participante e pontuação final

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Pontuação**
Participante 1	3	2	4	4	4	1	4	2	4	2	70
Participante 2	5	2	4	1	4	1	5	2	4	2	85
Participante 3	3	2	3	2	4	1	5	2	3	2	72,5
Participante 4	3	1	4	1	3	2	4	2	3	1	75
Participante 5	3	2	3	2	4	2	4	3	3	2	65

A média das pontuações foi 73,5 pontos, o mínimo 65 e o máximo 85. A literatura sugere que 68 é um valor de referência "médio". Os resultados superiores a 80 tendem a indicar excelente usabilidade, 68–80 indicam "bom/aceitável", 51–68 razoável "OK", e abaixo de 51 "fraco" [63]. Estes valores colocam a componente de pesquisa dinâmica na faixa de usabilidade "bom/aceitável", acima do limiar de referência do SUS, mas ainda com espaço para subir para bom e muito bom. A dispersão é moderada, o que sugere que pequenos obstáculos iniciais afetaram alguns participantes de forma mais evidente.

Os resultados mostram usabilidade globalmente positiva e alinhada com a avaliação qualitativa. A maioria sentiu a plataforma simples, com funcionalidades bem integradas e confiança na utilização depois de "entrar no ritmo". As variações individuais explicam a dispersão entre 65 e 85 pontos. As dificuldades iniciais identificadas, como a descoberta da tabela de resultados abaixo da configuração da *query*, o "+ Rule" pouco intuitivo e o arrastar e largar como mecanismo único de seleção de propriedades, ajudam a entender os casos com pontuação intermédia. Ao melhorar estes pontos, espera-se um ganho direto nas questões do SUS ligadas à facilidade de utilização, necessidade de apoio e aprendizagem inicial, o que deverá elevar a média em futuras iterações.

8 Conclusão

Neste trabalho desenvolveu-se e aperfeiçoou-se a componente de pesquisa dinâmica do DISME, complementada por uma API que garante a sua integração com sistemas externos.

O resultado alcançado permite configurar e executar pesquisas de forma visual sem escrever SQL, através de uma interface intuitiva, permitindo que pessoas não técnicas configurem consultas úteis e reutilizáveis. Através dela, o utilizador pode consultar, guardar, exportar e atualizar *queries* e, ainda, parametrizar consultas de forma dinâmica para reutilização em diferentes contextos. Esta configuração é guardada de forma persistente e executada sempre com a mesma semântica tanto na interface como na API, através de um contrato em JSON simples e estável, que inclui o cabeçalho da tabela de resultados e suas respetivas linhas.

A componente suporta parâmetros dinâmicos e expõe resultados por REST, permitindo que plataformas externas consultem dados guardados na base de dados do projeto.

A arquitetura manteve o modelo EAV em MySQL como base de dados principal e acrescentou a opção de vistas materializadas em MongoDB para diminuir o tempo de execução em consultas repetidas e com muitas linhas. Além disso, foi adicionada e avaliada a leitura por tabelas relacionais convencionais para comparar comportamentos. Os testes realizados destacaram o MongoDB com vistas materializadas como a melhor opção, uma vez que, é a que apresenta o tempo de execução mais curto. As tabelas relacionais tradicionais dão resposta eficiente a pesquisas comuns na interface, sendo portanto uma boa opção de ser integrada de forma definitiva no DISME, principalmente em momentos em que estão sendo realizadas muitas consultas ao mesmo tempo. E por fim, é no EAV que se encontram os dados sempre atualizados, contudo de entre as três abordagens é sem dúvida a mais lenta.

A avaliação de usabilidade, combinando *think aloud* e o método SUS, complementou a análise técnica com a perceção dos utilizadores. Numa amostra de utilizadores iniciantes, a aprendizagem revelou-se rápida e a navegação previsível. A parametrização e a publicação por REST foram compreendidas após um breve período de familiarização. Os resultados do SUS indicaram uma perceção global positiva da solução, salientando a baixa complexidade percebida e a integração coerente das funcionalidades, alinhadas com o objetivo de reduzir a dependência de conhecimentos técnicos em tarefas comuns de consulta e partilha.

O projeto cumpriu com todos os objetivos definidos. A pesquisa dinâmica agora pode ser configurada e reutilizada em casos que antes não era possível, a API REST facilitou a integração, e as vistas materializadas proporcionaram o desempenho que era necessário em cenários mais exigentes. Com isto, o DISME torna-se mais apto a atender equipas que necessitam de agilidade nas respostas, sem comprometer a consistência. Com isto, além de reforçar a posição do DISME dentro do conjunto de ferramentas do *Enterprise Engineering Lab*, a solução criada possibilita futuras ampliações, especialmente em termos de usabilidade e escalabilidade, garantindo que a plataforma se desenvolva de acordo com as necessidades reais de organizações e utilizadores.

Para finalizar, este trabalho deu-me a oportunidade de pôr em prática muitos dos conceitos e técnicas que fui aprendendo ao longo do meu percurso académico. Como o projeto de mestrado envolveu várias tarefas, acabei por desenvolver competências em áreas distintas, desde a gestão do projeto à implementação em *backend* e *frontend*, algo que noutro contexto dificilmente seria tão abrangente.

8.1 Trabalho futuro

O próximo passo é corrigir, de forma prioritária, os problemas de usabilidade observados nos testes. Implementar as funcionalidades em falta, simplificar passos que geraram bloqueios e clarificar textos e formulários.

Depois é crucial validar a componente em contexto real com perfis de utilizador distintos. O foco será medir usabilidade, taxa de conclusão e satisfação, em sessões mais longas e com tarefas que vão além do percurso básico já testado. Estas tarefas devem incluir agregações, *group by*, parâmetros dinâmicos e materialização, para espelhar o uso previsto em listas e relatórios recorrentes. Com base nos resultados, a interface será ajustada de forma iterativa para manter a configuração simples para quem não é técnico e, ao mesmo tempo, oferecer controlo suficiente a utilizadores experientes.

Posteriormente, também é importante atualizar a gestão das vistas materializadas, para que seja mais rápida e eficiente. Isso, na prática, quer dizer achar um meio-termo entre manter os dados sempre atualizados e acessíveis, mas sem gastar mais recursos do que o necessário.

A API precisa evoluir de maneira estável, com versões claras e uma documentação bem elaborada.

Por último, é indispensável que se acrescentem ao *query builder* operadores úteis como: *between*, *null*, *not null*, entre outros, para viabilizar filtros por intervalos e por valores ausentes, facilitando a configuração de pesquisas mais precisas. Uma vez consolidados estes operadores básicos, o próximo passo será adicionar funcionalidades como as *subqueries* e o uso dos operadores *in* e *not in* em conjunto com essas *subqueries*, porque são estas ferramentas que permitem representar situações que aparecem muitas vezes no uso real quando é preciso comparar resultados entre conjuntos de dados diferentes. As *subqueries* permitem gerar listas intermédias de valores e usá-las dentro da pesquisa principal de forma direta e clara, evitando soluções que obrigariam o utilizador a criar vistas ou tabelas auxiliares, enquanto os operadores *in* e *not in*, aplicados sobre essas listas, tornam possível filtrar resultados de acordo com conjuntos produzidos no próprio momento da pesquisa e dão ao construtor visual a expressividade que falta para lidar com cenários mais complexos.

Referências

[1] N. U. Ain, G. Vaia, W. H. DeLone, and M. Waheed, "Two decades of research on business intelligence system adoption, utilization and success: A systematic literature review," *Decision Support Systems*, Oct. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0167923619301423

[2] J. L. G. Dietz, *Enterprise Ontology: Theory and Methodology.* Berlin Heidelberg: Springer, 2006.

[3] M. R. Krouwel, M. Op 't Land, and H. A. Proper, "Generating low-code applications from enterprise ontology," in *The Practice of Enterprise Modeling (PoEM 2022)*, ser. Lecture Notes in Business Information Processing, B. S. Barn and K. Sandkuhl, Eds. Cham: Springer, 2022. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-21488-2_2

[4] V. Freitas, D. Pinto, V. Caires, L. Tadeu, and D. Aveiro, "The disme low-code platform - from simple diagram creation to system execution," 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)*, ser. CEUR Workshop Proceedings. Leusden, The Netherlands: CEUR-WS.org, Nov. 2022. [Online]. Available: https://ceur-ws.org/Vol-3388/paper4.pdf

[5] What is ad hoc analysis? | Definition from TechTarget. Search Business Analytics. [Online]. Available: https://www.techtarget.com/searchbusinessanalytics/definition/ad-hoc-analysis

[6] T. Svarre and T. Russell-Rose, "Think outside the search box: A comparative study of visual and form-based query builders," *Journal of Information Science*, 2025. [Online]. Available: https://vbn.aau.dk/en/publications/think-outside-the-search-box-a-comparative-study-of-visual-and-fo

[7] W. Gatterbauer, "A tutorial on visual representations of relational queries," *Proceedings of the VLDB Endowment*, no. 12, pp. 3890–3893, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p3890-gatterbauer.pdf

[8] S. S. Bhowmick and B. Choi, "Data-driven visual query interfaces for graphs: Past, present, and (near) future," in *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22).* ACM, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3514221.3522562

[9] W. Gatterbauer, C. Dunne, H. V. Jagadish, and M. Riedewald, "Principles of query visualization," *CoRR*, 2022. [Online]. Available: https://arxiv.org/abs/2208.01613

[10] H. Spengler, C. Lang, and T. Mahapatra, "Enabling agile clinical and translational data warehousing: Platform development and evaluation," *JMIR Medical Informatics*, no. 7, p. e15918, 2020. [Online]. Available: https://medinform.jmir.org/2020/7/e15918/

[11] L. M. Nguyen *et al.*, "Big data for healthcare: Using entity-attribute-value (eav) model to build a national platform for disability management," Preprint

/ working paper, 2023. [Online]. Available: https://www.semanticscholar.org/paper/35d9dd21e1a7f2cdd032b433ba62f267ad64d190

[12] D. Löper, M. Klettke, I. Bruder, and A. Heuer, "Enabling flexible integration of healthcare information using the entity-attribute-value storage model," ResearchGate, 2013. [Online]. Available: https://www.researchgate.net/publication/257884193_Enabling_flexible_integration_of_healthcare_information_using_the_entity-attribute-value_storage_model

[13] J. W. Yoder, R. Johnson, and D. Foote, "The adaptive object-model architectural style," in *Proceedings of WICSA 3*, 2001. [Online]. Available: https://www.adaptiveobjectmodel.com/WICSA3/ArchitectureOfAOMsWICSA3.pdf

[14] F. Katsch, R. Hussein, and G. Duftschmid, "Converting entity-attribute-value data sources to omop's cdm: Lessons learned," *Studies in Health Technology and Informatics*, pp. 356–357, 2024. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/39176749/

[15] W. Khan, T. Kumar, C. Zhang, K. Raj, A. M. Roy, and B. Luo, "Sql and nosql database software architecture, performance analysis and assessments—a systematic literature review," *Data*, no. 2, p. 97, 2023. [Online]. Available: https://www.mdpi.com/2504-2289/7/2/97

[16] A. Margara, G. Cugola, N. Felicioni, and S. Cilloni, "A model and survey of distributed data-intensive systems," *ACM Computing Surveys*, pp. 1–71, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3604801

[17] A. Williams, "Understanding nosql database types: Document," 06 2021. [Online]. Available: https://cacm.acm.org/blogcacm/understanding-nosql-database-types-document/

[18] M. Budiu, T. Chajed, F. McSherry, L. Ryzhyk, and V. Tannen, "Dbsp: Automatic incremental view maintenance for rich query languages," *Proc. VLDB Endow.*, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p1601-budiu.pdf

[19] C. Svingos, A. Hernich, H. Gildhoff, Y. Papakonstantinou, and Y. Ioannidis, "Foreign keys open the door for faster incremental view maintenance," *Proc. ACM Manag. Data*, May 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3588720

[20] P. Lappas, M. Armbrust, M. E. Hossain, M. Li, M. Lu, E.-G. Kim, and R. Yadav, "Announcing the general availability of materialized views and streaming tables for databricks sql," Databricks Blog, 11 2024. [Online]. Available: https://www.databricks.com/blog/announcing-general-availability-materialized-views-and-streaming-tables-databricks-sql

[21] Microsoft Azure Architecture Center. (2025) Materialized view pattern. Microsoft. [Online]. Available: https://learn.microsoft.com/azure/architecture/patterns/materialized-view

[22] R. Soepriadi *et al.*, "Performance improvement of periodic reports with materialized views on oracle database system," *Sistemasi: Jurnal Sistem Informasi*, vol. 14, no. 2, 2025. [Online]. Available: https://doaj.org/article/0e237a662a3f48b9bf6bf5de490f0c4c

[23] D. Olteanu, "Recent increments in incremental view maintenance," arXiv preprint arXiv:2404.17679, 2024. [Online]. Available: https://arxiv.org/abs/2404.17679

[24] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of Cal-

ifornia, Irvine, 2000. [Online]. Available: https://www.researchgate.net/publication/216797523_Architectural_Styles_and_the_Design_of_Network-based_Software_Architectures

[25] Perforce, "What is a rest api?" 2020. [Online]. Available: https://www.perforce.com/blog/aka/what-is-rest-api

[26] Y. Barnard, "Url parameters: What they are and how to use them properly," Backlinko, Mar. 2025. [Online]. Available: https://backlinko.com/url-parameters

[27] R. Fielding, M. Nottingham, and J. Reschke, "RFC 9110: HTTP semantics," 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110

[28] A. Lercher, J. Glock, C. Macho, and M. Pinzger, "Microservice api evolution in practice: A study on strategies and challenges," *Journal of Systems and Software*, 2024. [Online]. Available: https://arxiv.org/abs/2311.08175

[29] H. Mann, "Rest api basics – 4 things you need to know," Mar. 2023. [Online]. Available: https://mannhowie.com/rest-api

[30] J. Bogner *et al.*, "Do restful api design rules have an impact on the understandability of web apis?" *Empirical Software Engineering*, 2023. [Online]. Available: https://arxiv.org/abs/2305.07346

[31] (2025) Mendix documentation. [Online]. Available: https://docs.mendix.com/

[32] (2025) Outsystems documentation. [Online]. Available: https://success.outsystems.com/

[33] (2025) Skyvia documentation. [Online]. Available: https://docs.skyvia.com/

[34] (2025) Data querying support & management (mendix evaluation guide). [Online]. Available: https://www.mendix.com/evaluation-guide/app-lifecycle/develop/data-management/data-querying-management/

[35] (2025) Publish a rest service. Mendix Documentation. [Online]. Available: https://docs.mendix.com/refguide/publish-a-rest-service/

[36] (2025) Mendix pricing benchmarking. [Online]. Available: https://www.vertice.one/vendors/mendix

[37] (2025) Aggregate — outsystems 11 documentation. [Online]. Available: https://success.outsystems.com/documentation/11/reference/outsystems_language/data/handling_data/queries/aggregate/

[38] (2025) Sql queries — outsystems 11 documentation. [Online]. Available: https://success.outsystems.com/documentation/11/reference/outsystems_language/data/handling_data/queries/sql/

[39] L. Fidalgo, "How to optimize your queries in outsystems: the 'max records' property," 2021. [Online]. Available: https://www.osquay.com/knowledge-center/how-to-optimize-your-queries-in-outsystems

[40] (2025) Document an exposed rest api. OutSystems 11 Documentation. [Online]. Available: https://success.outsystems.com/documentation/11/integration_with_external_systems/rest/expose_rest_apis/document_an_exposed_rest_api/

[41] (2025) Expose a rest api. OutSystems 11 Documentation. [Online]. Available: https://success.outsystems.com/documentation/11/integration_with_external_systems/rest/expose_rest_apis/expose_a_rest_api/

[42] (2025) Configuring queries with query builder. [Online]. Available: https://docs.skyvia.com/query/configuring-queries-with-query-builder.html

[43] (2025) Odata endpoints. Skyvia Docs. [Online]. Available: https://docs.skyvia.com/connect/odata-endpoints/

[44] (2025) Sending requests to skyvia connect odata endpoints. Skyvia Docs. [Online]. Available: https://docs.skyvia.com/connect/odata-endpoints/sending-requests-to-skyvia-connect-odata-endpoints.html

[45] (2025) Supported odata protocol features. Skyvia Docs. [Online]. Available: https://docs.skyvia.com/connect/odata-endpoints/supported-odata-protocol-features.html

[46] (2025) Odata endpoints — working with created endpoints. [Online]. Available: https://docs.skyvia.com/connect/odata-endpoints/#working-with-created-endpoints

[47] M. Andrade, D. Aveiro, and D. Pinto, "Demo based dynamic information system modeller and executer," in *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K)*. Seville, Spain: SCITEPRESS, 2018. [Online]. Available: https://www.scitepress.org/Link.aspx?doi=10.5220/0007230003830390

[48] M. M. Bouhamed, G. Díaz, A. Chaoui, O. Kamel, and R. Nouara, "Models@runtime: The development and re-configuration management of python applications using formal methods," *Applied Sciences*, vol. 11, no. 20, p. 9743, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/20/9743

[49] R. Heinrich, "Architectural runtime models for integrating runtime observations and component-based models," *Journal of Systems and Software*, vol. 169, p. 110722, 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110722

[50] H. S. Ferreira, F. F. Correia, J. W. Yoder, and A. Aguiar, "The lazy semantics pattern on the context of meta-architectures," in *2nd Asian Conference on Pattern Languages of Programs (AsianPLoP 2011)*, Tokyo, Japan, Oct. 2011. [Online]. Available: https://www.researchgate.net/publication/228965396_The_Lazy_Semantics_Pattern_on_the_context_of_Meta-Architectures

[51] K. E. Wiegers and J. Beatty, *Software Requirements*, ser. Developer Best Practices. Microsoft Press, 2013. [Online]. Available: https://ptgmedia.pearsoncmg.com/images/9780735679665/samplepages/9780735679665.pdf

[52] N. Silnitsky, "How to choose the right database for your service," Wix Engineering, Aug. 2022. [Online]. Available: https://medium.com/wix-engineering/how-to-choose-the-right-database-for-your-service-97b1670c5632

[53] A. Bayat, "Arangodb features review," Medium, Sep. 2021. [Online]. Available: https://medium.com/@ali.bayat/arangodb-features-review-7db4d72824b3

[54] ScienceDirect, “Neo4j - an overview | sciencedirect topics.” [Online]. Available: https://www.sciencedirect.com/topics/computer-science/neo4j

[55] H. Patel, “Redis — what and why?” Weekly Webtips, Sep. 2020. [Online]. Available: https://medium.com/weekly-webtips/redis-what-and-why-pros-cons-ae2f5bc750fd

[56] Redis. Redisgraph. [Online]. Available: https://redis.io/docs/stack/graph/

[57] A. Haliti, “Zhvillimi i aplikacionit e-forumi nË platformen e ueb-it: Duke pËrdor teknologjitË mongodb dhe laravel,” Theses and Dissertations, Apr. 2019. [Online]. Available: https://knowledgecenter.ubt-uni.net/etd/1391

[58] Mongodb and laravel integration. [Online]. Available: https://www.mongodb.com/compatibility/mongodb-laravel-intergration

[59] Laravel mongodb php driver — quick start: Download and install. [Online]. Available: https://www.mongodb.com/docs/drivers/php/laravel-mongodb/current/quick-start/download-and-install/

[60] Weasy, “Projeto micolec.” [Online]. Available: http://logimade.pt/projeto-micolec

[61] D. W. Eccles and G. Arsal, “The think aloud method: what is it and how do i use it?” *Qualitative Research in Sport, Exercise and Health*, vol. 9, no. 4, pp. 514–531, 2017. [Online]. Available: https://doi.org/10.1080/2159676X.2017.1331501

[62] A. Bangor, P. T. Kortum, and J. T. Miller, “An Empirical Evaluation of the System Usability Scale,” *International Journal of Human–Computer Interaction*, vol. 24, no. 6, pp. 574–594, 2008. [Online]. Available: https://doi.org/10.1080/10447310802205776

[63] J. Sauro, “5 ways to interpret a sus score,” MeasuringU blog, Sep. 2018. [Online]. Available: https://measuringu.com/interpret-sus-score/