University of Minho
School of Engineering

Representação e Processamento de Conhecimento na Web

Trabalho Prático

- Eduardo Pereira PG53797
- Pedro Parpot PG47560

[1] Universidade do Minho, 4710-057 Braga, Portugal

Índice

1 Introdução

Neste relatório apresentamos o desenvolvimento do sistema de representação e processamento de conhecimento na Web, centrado na modelagem do domínio de videojogos. A aquisição de dados foi realizada sobretudo através da *DBpedia* e da *API Rawg.io*, permitindo-nos obter informações até ao final de 2025. A estrutura da ontologia inclui classes como *Game*, *Platform*, *Developer*, *Engine*, *Genre* e *Series*.

A partir desta ontologia, implementamos um fluxo de povoamento automático que processa ficheiros *Json* e gera múltiplos ficheiros *Turtle* para alimentar um repositório *GraphDB*. Sobre este grafo *RDF*, construímos uma aplicação Web com *frontend* em Jinja e backend em *Flask/Python*, expondo funcionalidades de perquisa básica, perquisa avançada via consultas *SPARQL* e um modo de quiz interativo. No frontend, cada resultado ou entidade pode ser explorado de forma navegável, clicando em relações que carregam páginas de detalhe.

Nos capítulos subsequentes, detalhamos o processo de modelação e povoamento da ontologia, a arquitetura da aplicação (incluindo endpoints REST e templates), bem como os resultados obtidos (pesquisa, quiz e visualização de entidades), concluindo com uma reflexão sobre a eficácia e possíveis extensões deste sistema.

2 Povoamento & Ontologia

2.1 Tema

Para este projeto decidimos criar a nossa própria ontologia e, desse modo, começamos por procurar um tema que satisfizesse as seguintes condições:

- **Riqueza de Dados**: O tema escolhido tinha que ter dados que pudessem formar conexões e abstrações complexas e preencher várias entidades.
- **Disponiblidade de Datasets**: É importante que o tema tivesse vários "datasets" ricos e de várias fontes, para se ter um grande conjunto de dados para povoar.
- **Familiaridade com o Tema**: Usar um tema que tivéssemos alguma familiaridades para saber se houvesse problemas com a aquisição de dados e poder guiar o povoamento melhor.

Tendo estas especificações em mente decidimos escolher o tema de **Video Jogos**. Visto que, é um tema com bastante diversidade e riqueza de dados, com os quais a nossa equipa tem bastante familiaridade.

2.2 Aquisição de Dados

Antes de determinar o tema inspecioná-mos vários *datasets* públicos relacionados e, para o tema de video jogos, não havia falta. As melhores fontes para adquirir os dado de video jogos que encontramos são:

- **IGDB** - Uma base de dados para jogos de computador criada pela empresa *Twitch* que pertence á *Amazon*.
- **Rawg.io** - Uma base de dados independente com uma vasta aquisição de jogos de várias lojas.
- **SteamDB** - Uma base de dados independente com jogos vendidos apenas pela plataforma *Steam*.
- **DBPedia** - A DBpedia é um esforço colaborativo de crowd-sourcing da comunidade para extrair conteúdo estruturado a partir da informação criada em vários projectos da Wikimedia.

Para o nosso projeto decidimos fazer a aquisição primária com o **DBPpedia** a partir do *SPARQL endpoint*, visto que de todas as fontes, tinha uma maior riqueza de informação e uma melhor forma de colecionar os dados, no entanto a secção de video jogos da DBPedia tem uma "cutoff date" de julho de 2023 o que quer dizer que jogos mais recentes não estão incluídos.

Para estender os nossos dados para além de 2023 optamos por usar a API do **Rawg.io** porque, pelo que inspecionamos, era a API com mais versatilidade das mencionadas acima (permitindo 20.000 *API requests*) e

abrangia uma vasta quantidade de dados como a **DBPedia**. Para filtrar apenas pelos dados depois de 2023 usamos a variável `?dates=2023-07-01,2025-12-31` disponibilizada pela API.

Todos os dados adquiridos foram guardados em formato *Json* para, posteriormente, puderem ser mais facilmente processados quando chegar a altura de criar o povoamento.

2.3 Ontologia

Com base na informação recolhida, a nossa ontologia modela o domínio de videojogos, incorporando relações entre jogos, plataformas, estúdios, "game engines", géneros e séries de jogos. Abaixo, apresenta-mos uma síntese das entidades (classes) e das propriedades (objeto e dados).

2.3.1 1. Classes Principais

- **Game**: Representa um videojogo individual.
- **Platform**: Plataforma genérica onde um jogo pode correr (por exemplo, consola, PC, móvel).
  - **Device** (subclasse de **Platform**): Refine **Platform** para dispositivos específicos (e.g., PlayStation, Xbox).
- **Developer**: Estúdio ou entidade responsável pelo desenvolvimento de jogos.
- **Engine**: Motor de jogo utilizado para construir e executar jogos (e.g., Unity, Unreal).
- **Genre**: Gênero ou estilo de jogo (e.g., ação, RPG, estratégia).
- **Series**: Série ou franquia a que um jogo pode pertencer (e.g., "The Legend of Zelda").

2.3.2 2. Propriedades de Objeto (Relações entre instâncias)

Para brevidade, descrevem-se pares de propriedades inversas juntos, indicando domínio (D) e intervalo (I):

1. **hasGenre / appliesTo**
   - **hasGenre** (D: Game → I: Genre): Indica o género de um jogo.
   - **appliesTo** (D: Genre → I: Game): Inversa de **hasGenre**.
2. **availableOn / hosts**
   - **availableOn** (D: Game → I: Platform): Plataforma onde o jogo está disponível.
   - **hosts** (D: Platform → I: Game): Inversa de **availableOn**.
3. **builtOn / engineFor**
   - **builtOn** (D: Game → I: Engine): Motor utilizado para desenvolver o jogo.
   - **engineFor** (D: Engine → I: Game): Inversa de **builtOn**.
4. **developedBy / hasDeveloped**
   - **developedBy** (D: Game → I: Developer): Estúdio que desenvolveu o jogo.
   - **hasDeveloped** (D: Developer → I: Game): Inversa de **developedBy**.
5. **partOf / includes**
   - **partOf** (D: Game → I: Series): Especifica a série à qual um jogo pertence.
   - **includes** (D: Series → I: Game): Inversa de **partOf**.
6. **engineMadeFor / deviceForEngine**
   - **engineMadeFor** (D: Engine → I: Device): *Device* suportado por uma *Engine*.
   - **deviceForEngine** (D: Device → I: Engine): Inversa de **engineMadeFor**.
7. **developedDevice / deviceDevelopedBy**
   - **developedDevice** (D: Developer → I: Device): *Developer* desenvolveu um *Device*.

- **deviceDevelopedBy** (D: Device → I: Developer): Inversa de **developedDevice**.

8. **isDeviceFrom / isPlatformOf**

- **isDeviceFrom** (D: Device → I: Platform): *Device* faz parte da família de plataformas *Platform*.
- **isPlatformOf** (D: Platform → I: Device): Inversa de **isDeviceFrom**.

Cada propriedade herda implicitamente **owl:ObjectProperty**, permitindo inferências OWL (por exemplo, se um jogo **G hasGenre** "RPG", pode-se deduzir que "RPG" **appliesTo G**).

2.3.3 3. Propriedades de Dados (Atributos)

- **Name** (xsd:string): Nome legível de cada instância.
- **Abstract** (xsd:string): Descrição textual de uma entidade (normalmente usada em **Game**, mas sem domínio fixo).
- **Release_Date** (xsd:string): Data de lançamento (formato texto; p. ex., "2023-11-15").
- **ESRB_Rating** (xsd:string): Classificação etária (p. ex., "Teen", "Mature").
- **Metacritic** (xsd:string): Pontuação agregada de crítica.
- **Playtime** (xsd:float): Tempo médio de jogo (em horas).

Essas propriedades de dados permitem anexar características descritivas e numéricas às instâncias, facilitando pesquisas e agregações (e.g., filtrar jogos por pontuação ou tempo médio).

2.4 Povoamento

Para povoar a ontologia com os dados em *Json* usámos o script `povoamentoJson.py` que realiza o povoamento automático da ontologia a partir de ficheiros *Json* que contem os dados de videojogos, como foi mencionado acima.

Para cada jogo, cria instâncias da classe `Game` e associa propriedades como nome, data de lançamento, classificação ESRB, playtime, Metacritic, plataformas disponíveis (`Device`) e géneros (`Genre`). Evita duplicação de instâncias usando conjuntos (`set`) e divide a serialização em múltiplos ficheiros `.ttl` por cada 2500 jogos, garantindo melhor gestão da memória e organização dos dados. Utiliza a biblioteca `rdflib` para criar e manipular o grafo RDF.

3 Aplicação Web

A aplicação Web para a exploração de ontologia de jogos integra frontend responsivo com Jinja, backend em Flask/Python e um repositório GraphDB acedido com via SPARQL.

É organizada com a arquitetura *MVC* (Model-View-Controller), onde:

- **Model**: a ontologia em GraphDB acedida via SPARQL;
- **View**: templates Jinja2 e scripts JavaScript que consomem APIs REST;
- **Controller**: rotas Flask que processam as requisições, validam entradas, executam SPARQL e retornam dados para exibição.

3.1 1. Estrutura de Frontend com Jinja

A interface do usuário é gerada a partir de templates Jinja, que combinam HTML, CSS e JavaScript para fornecer uma experiência interativa.

- **Template principal (`jinja_template.html`)**: contém a barra de navegação ("Search", "Advanced Search", "Quiz Mode"), secções ocultáveis para pesquisa básica, SPARQL e modo quiz. O layout usa classes CSS com estilos modernos (gradientes, tipografia e responsividade) para garantir coesão visual e usabilidade em diferentes tamanhos de tela.

- **Template de detalhe de entidade (`entity_detail.html`)**: exibe as informações específicas de uma instância de uma classe da ontologia, organizadas em seções de parâmetros. Cada parâmetro é agrupado por propriedade e representado em "value-tags" clicáveis, permitindo navegação em grafo sem recarregar toda a página.

3.2 2. Backend em Flask

O servidor Flask atua como orquestrador de requisições e comunicação com o repositório. A aplicação (`app.py`) inicia a aplicação, configura uma chave secreta e define o endpoint SPARQL para o GraphDB. Os principais componentes são:

- **Cliente GraphDB (`GraphDBClient`)**: classe que encapsula chamadas HTTP POST ao endpoint SPARQL hospedado localmente em `http://localhost:7200/repositories/videogames`. Recebe consultas SPARQL, envia para o GraphDB e retorna resultados JSON. Lida com erros de conexão e formatação de resposta.
- **Construção de consultas de pesquisa (`build_search_query`)**: função que monta dinamicamente cláusulas SPARQL com base em termos de pesquisa, categoria (jogo, gênero, plataforma, desenvolvedor ou geral) e ordenação (por nome ou data de lançamento). Retorna uma string com prefixos RDF, padrões de pesquisa e cláusulas `ORDER BY`, conforme a necessidade.
- **Validação de SPARQL customizado (`validate_sparql_query`)**: rotina simples que bloqueia operações destrutivas (DROP, DELETE, etc.) e garante que a consulta contenha `SELECT`. Fornece mensagem de erro caso seja inválida.
- **Formatação de resultados para frontend**:

  `format_search_results` converte bindings SPARQL em uma lista de objetos JSON com título, tipo (Game, Genre, Platform, Developer) e URI relativa para navegação interna. `format_custom_sparql_results` adapta resultados de consultas livres em uma estrutura com cabeçalhos (`headers`) e linhas (`rows`), identificando URIs clicáveis para navegação detalhada.

3.3 3. Endpoints e API REST

A aplicação define rotas principais para atender ao *frontend* e para expor APIs JSON:

- **`GET /`**: carrega o template principal (`jinja_template.html`), responsável pelo *render* da página inicial com as abas de pesquisa básica, avançada e quiz.
- **`POST /api/search`**: recebe JSON com `query` (texto de perquisa), `category` e `sort`. Invoca `build_search_query`, executa SPARQL e retorna resultados formatados. Este endpoint é consumido por JavaScript no frontend, que exibe resultados em tempo real sem recarregar a página.
- **`POST /api/sparql-query`**: recebe consulta SPARQL livre do usuário, valida via `validate_sparql_query`, executa no GraphDB e devolve resultados para renderização em tabela, com suporte a enlaces nas células.
- **`GET /entity/<entity_id>`**: endpoint dinâmico que perquisa parâmetros de uma entidade específica. A função `get_entity_details` monta e executa uma consulta SPARQL para coletar pares propriedade-valor (incluindo nomes de entidades relacionadas) e envia dados ao template `entity_detail.html`, que então exibe propriedades agrupadas por rótulo.
- **`GET /api/quiz-questions`**: retorna um conjunto de perguntas geradas aleatoriamente a partir da ontologia. Cada função de geração de pergunta (`random_data_question`, `random_platform_question`, etc.) faz consultas SPARQL específicas (como selecionar aleatoriamente um jogo e obter ano de lançamento ou plataformas) e monta opções de múltipla escolha, incluindo a resposta correta.

4 Resultados

4.1 Basic Search

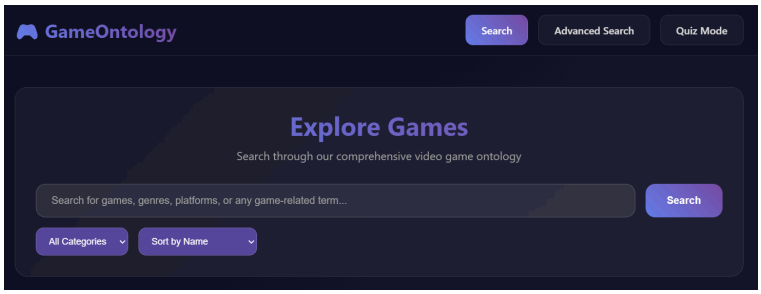

Figura 1. Tab com a Search feature

A *Basic Search* permite a um utilizador pesquisar por uma combinação de carácteres, filtrando por:

- **All Categories**: Lista todas as entidades de qualquer tipo que incluem a *string* a pesquisar no nome.
- **X** : Lista todos as entidades da classe X que incluem a *string* a pesquisar no nome.

E permite ordenar por nome, e data de adição os jogos. Expondo os resultados em formato de lista, no qual cada item incluem o nome da entidade e o seu tipo adjacente. Cada item redireciona para a sua página de entidade quando clicado.

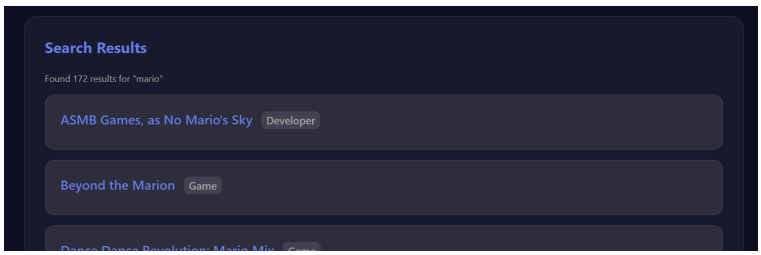

Figura 2. Resultados de uma Search

4.2 Advanced Search

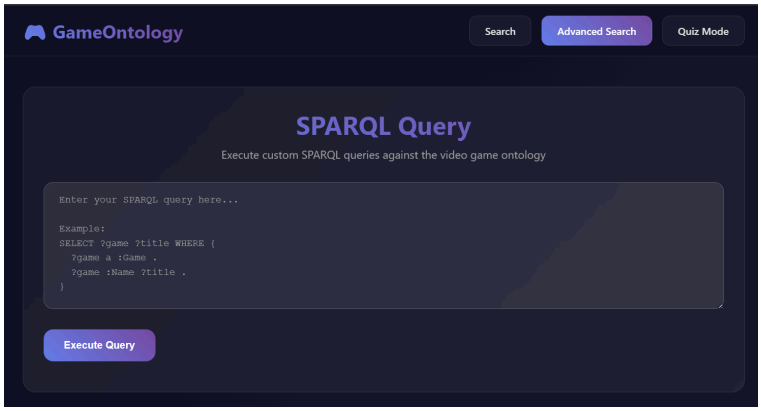

Figura 3. Tab com a Advance Search feature

A *Advanced Search* permite a um usuário executar uma query sparql SELECT, expondo os resultados em formato de tabela em baixo. Se um dos campos conter uma referência para uma entidade da ontologia, pode ser clicado e direcionado para a página da entidade.

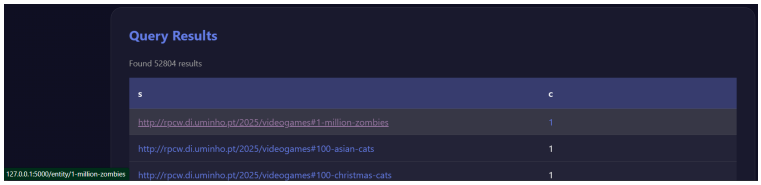

Figura 4. Resultados de uma Advanced Search

4.3 Quiz Mode

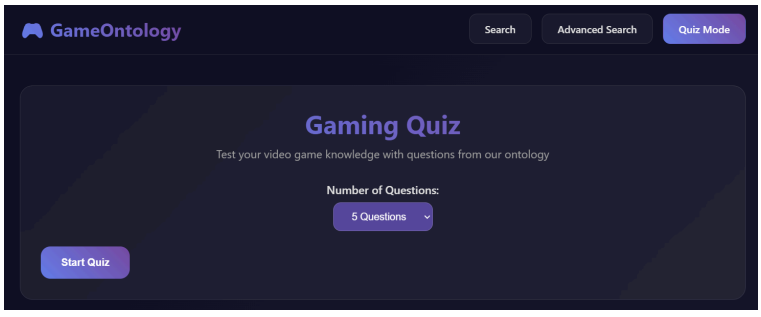

Figura 5. Tab com a Quiz feature

O *Quiz Mode* dá a opção ao utilizador de selecionar um número de questões que serão geradas quando começar o quiz com "Start Quiz". As questões são geradas selecionado entidades aleatoriamente da ontologia e criando questões de seleção de opção com base numa estrutura pré determinada.

Figura 6. Uma pergunta de um Quiz

4.4 Entity Display

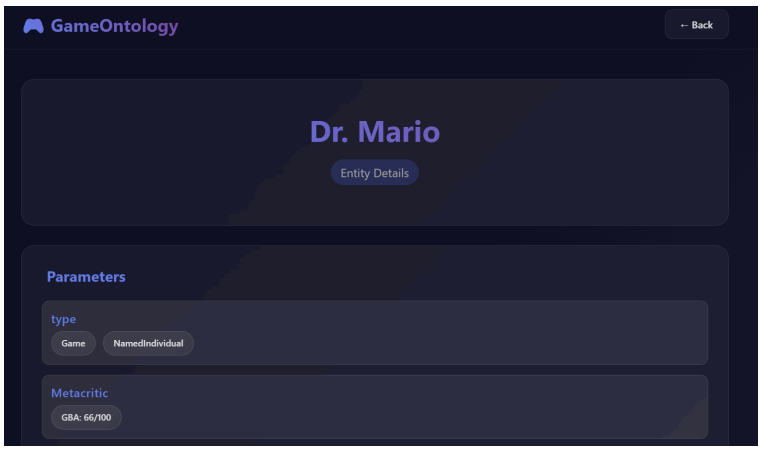

Figura 7. Página da entidade Game "Dr. Mario"

A página *Entity Display* expõe os dados de uma entidade, começando pelo nome como titulo da página. Em baixo, mostra uma lista de parâmetros construída a partir de todos os predicados e objetos relacionados ao sujeito. Se um objeto for uma entidade (como um Game que pertence a uma Series) esse campo pode ser clicado para reencaminhar para a página de entidade do objeto.

5 Conclusão

Em suma, este projeto demonstrou a viabilidade de construir um sistema completo de representação e processamento de conhecimento, desde a definição de uma ontologia abrangente até ao povoamento automático de dados e à disponibilização de interfaces web para pesquisa e quiz.

Através da utilização do *SPARQL endpoint* da *DBpedia* e da *API* da *Rawg.io*, conseguimos enriquecer as entidades com atributos relevantes (datas de lançamento, pontuações, estúdios, motores, géneros, etc.) e estruturar essa informação num grafo RDF eficiente, armazenado no GraphDB. A aplicação desenvolvida em Flask/Jinja/Python permitiu validar as consultas SPARQL e demonstrou a forma como utilizadores podem explorar dinamicamente relações entre jogos, plataformas e desenvolvedores.

Como trabalho futuro, poderíamos apoiar o uso de *queries* de inserção e construção nos pedidos *SPARQL*. Outra linha de evolução poderia ser aperfeiçoar a interface do quiz e a navegação baseada em grafos, estas que podem ser ampliadas com visualizações mais interativas (graph views) e módulos de personalização de perfil, tornando o sistema ainda mais útil tanto para entusiastas de videojogos como para investigadores na área de representação de conhecimento.