INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

INTEGRAÇÃO DE SISTEMAS DE INFORMAÇÃO

Internet Movie Database - Processos ETL

LICENCIATURA EM ENGENHARIA INFORMÁTICA

Jorge Eduardo Quinteiro Oliveira - 15475

Leandro Viana Faria - 16537

Barcelos, Portugal

15 de Novembro de 2020

Resumo

[Dada a importância do papel das tecnologias de informação, é imposta a comunicação entre serviços ou aplicações permitindo retrocompatibilidade, controlo, comunicação e integração. Neste contexto, a solução passa por integrar serviços ou sistemas a nível de dados, quer com recurso a ferramentas de Extração-Transformação-Carregamento (*sigla ETL, em inglês*) ou recorrendo à criação de novas ferramentas de raíz. O presente relatório descreve o processo de Extração-Transformação-Carregamento de dados com recurso à ferramenta Kettle, ou com recurso à linguagem de progranação C#. Inicialmente, será descrito o estado da arte, abordando as tecnologias existentes, prós e contras, e possiveis métodos de melhoria de eficiência a nível de transformação de dados. De seguida será detalhada a implementação de todo o trabalho desenvolvido, desde o desenho da solução e da arquitetura até à solução final. Por fim, no capítulo de testes será realizado um conjunto de testes à aplicação, de modo a validar o trabalho desenvolvido.]

Palavras-Chaves: [comunicação, serviços, integração, controlo, transformação]

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

No capítulo introdutório será discutido o contexto do problema, sendo efetuado também um enquadramento à integração de sistemas de informação. De seguida é apresentada a motivação e objetivos do projeto e por fim é descrita a estrutura do documento.

1.1 Contextualização

É imperativo em grande parte dos sistemas de informação a existência de troca de informação, como numa transação bancária ou no simples preenchimento e envio da declaração de *Imposto Sobre o Rendimento de Pessoas Singulares (IRS)*. No entanto, essa tarefa pode ser dificultada pela incorreta sequência de informação, um formato de dados errado, inconsistência dos dados ou informação errada.

Surgiu então a àrea de integração dos dados, para responder a este tipo de problemas. A solução passa pela análise das regras e processos de negócio e, a partir daí, construir uma solução que permita a comunicação entre dois ou mais sistemas. Um dos processos da integração de sistemas passa pela extração, transformação e carregamento (*sigla ETL, em inglês*) dos dados:

- Extrair dados de fontes externas;
- Transformar dados de modo a ir de acordo às necessidades do negócio;
- Carregar os dados;

Deste modo, conseguimos integrar um conjunto de sistemas baseado em regras, coeso e fácil de interpretar. Este tipo de ferramentas é bastante útil, pois funcionam de forma interativa

através do arraste de objetos numa interface, não sendo necessária praticamente qualquer programação.

1.2 Objetivos

O principal propósito deste projeto é a transformação e extração de dados de um *dataset*, proveniente da interface da *Internet Movie Database (IMDB)*, nomeadamente o ficheiro comprimido *title.basics.tsv.gz*.

Dada a inexistência de uma *Application Programming Interface (API)* por parte da *Internet Movie Database*, apenas são disponibilizados ficheiros *tab-separated-values (.tsv)*, com os respetivos conteúdos.

No final, devem resultar três ficheiros *Extensible Markup Language (.xml)*, separados pelo tipo de título (Filme, Curta Metragem ou Série Televisiva). As soluções serão desenvolvidas em duas plataformas completamente distintas, nomeadamente numa ferramenta *Extract Transform Load (ETL)*, e de seguida com recurso à linguagem de programação *C#* e à *.NET Core*.

Será, por fim, efetuada uma análise comparativa das respetivas tecnologias em relação à transformação e extração em *bigdata*.

1.3 Estrutura do documento

O documento encontra-se organizado em cinco capítulos, detalhados de seguida.

O capítulo Introdutório detalha o projeto, onde é feita uma abordagem ao contexto do problema, motivação e objetivos.

O capítulo de Estado da Arte, que visa uma abordagem ao processo de extração, transformação e carregamento (*sigla ETL, em inglês*) e soluções existentes no mercado.

O capítulo de Implementação, onde é descrito o problema abordado e como o projeto foi implementado na vertente de programação e arquitetura, de modo a alcançar os objetivos.

O capítulo de Testes, contendo toda a informação relativa à deteção e correção de erros existentes na solução criada.

Por fim, o capítulo de Conclusão, onde são discutidos os resultados obtidos e as conclusões retiradas com o desenvolvimento da solução.

2 Estado da Arte

No Estado da Arte do presente trabalho é abordado o processo de extração, transformação e carregamento, bem como a existência de aplicações existentes no auxílio de extração e transformação de dados. É também abordada a interface providenciada pela *IMDB - Internet Movie Database* para disponibilização de dados sem atribuição de licença comercial.

2.1 Extract, Transform, Load (ETL) Tools

O armazenamento de dados pode ser considerado uma abordagem à integração de dados. Os dados são extraídos das fontes, transformados e depois carregados para uma base de dados ou simplesmente guardados em ficheiro. As ferramentas *ETL* têm como propósito auxiliar e automatizar essas etapas, de modo a potênciar a integração de dados. O método de armazenamento de dados é atuzlizado periódicamente, ocorrendo normalmente em horários fora de pico, dada a baixa carga nos servidores, por exemplo. Hoje em dia, a periocidade de atualização dos dados é, normalmente diária. O desafio consiste numa abordagem de atualização em tempo real. Vários servicos *Enterprise Resource Planner (ERP)* possuem infraestruturas, nomeadamente *cloud*, que permitem esta integração em tempo real, ou mesmo ferramentas de apoio à indústria, tirando partido da *Internet of Things (IoT)* e da *Indústria 4.0.*

2.2 Internet Movie Database (IMDB) - Interface

O facto da interface de programação de aplicações da *IMDB - Internet Movie Database* ainda ter um acesso bastante restrito por estar em construção é um constrangimento ao desenvolvimento de aplicações que a queiram integrar. Neste sentido, aquando da aquisição da *IMDB*

por parte da *Amazon* foi dado início à construção de uma interface de programação de aplicações recorrendo a soluções da própria, estando o processo por concluír.

De modo a não deixar de apoiar os desenvolvedores, é disponibilizado pela *IMDB* um interface com *datasets*, sete no total, contendo dados desde títulos a avaliações. Estes ficheiros, alguns contendo mais de dez milhões de linhas, estão comprimidos de modo a facilitar o *downlaod* dos mesmos. Os dados são tratados como *bigdata*, sendo atualizados diariamente. O tipo de licença é somente para uso pessoal e não comercial. Existem outras soluções que integram estes mesmos dados, mas alojados em *cloud*, estando restritos a um certo número de consultas diárias sem qualquer custo, sendo as restantes taxadas de acordo com o número de pesquisas efetuadas. De referir que são serviços completamente alheios à *IMDB*.

3 Implementação

No capítulo de Implementação é efetuada uma descrição do problema bem como a solução encontrada, descrevendo sucintamente todos os passos tomados na resolução do mesmo.

3.1 Descrição do problema

Hoje em dia é bastante simples obter informação sobre um filme ou uma série, tais como o nome, data de criação, a avaliação efetuada por alguma entidade ou o(s) género(s). Para o efeito, foi criada uma base de dados chamada *Internet Movie Database (IMDb)* pelo britânico *Col Needham*, com o intuito de centralizar numa plataforma *online* informação acerca de música, cinema, filmes, programas televisivos e até mesmo jogos de computador.

Ao mesmo tempo, comecaram a surgir as primeiras plataformas de *streaming* como a *Netflix*, que permitem a visualização *online* de filmes e séries. Seguindo este exemplo das plataformas de *streaming*, como é que podemos integrar estas duas plataformas e tirar proveito disso?

A resposta reside na integração de sistemas de informação. Para um determinado filme da plataforma de *streaming* é necessário um conjunto de de informação acerca desse filme, até porque o utilizador necessita de informação acerca do que possivelmente vai visualizar.

Assim, a *Internet Movie Database (IMDb)* possui uma interface em que disponibiliza todos os dados dos filmes que possui em base de dados, de modo a permitir por via de extração e transformação de dados, uma integração com a sua interface.

Dada a complexidade de cada ficheiro disponibilizado pela *Internet Movie Database (IMDb)*, é necessário um tratamento de dados prévio, e uma posterior transformação de dados

para que possam ser carregados na plataforma que pretende tirar proveito. Podemos seguir várias abordagens, duas delas por exemplo seriam a transformação dos dados em ficheiro *.xml*, ou carregamento para base de dados. Deste modo, é possivel criar uma integração, embora um pouco abstrata dada a inexistência de uma interface de programação de aplicação oficial por parte da *Internet Movie Database (IMDb)*.

O ficheiro a utilizar consiste na compilação e posterior compressão da seguinte informação acerca dos títulos presentes na *Internet Movie Database (IMDb)*:

- titleId (string) - um tconst, um identificador único alfanumérico do título;
- titleType (string) – o tipo/formato do título (e.g. movie, short, tvseries, tvepisode, video, etc);
- primaryTitle (string) – o nome mais popular do título / usado pelos produtores em material de promoção no momento do lançamento;
- originalTitle (string) - nome do título original, na língua original;
- isAdult (boolean) - 0: título não-adulto; 1: título adulto;
- startYear (YYYY) – representa o ano de lançamento do título. No caso de séries televisivas, representa o ano de início de transmissão;
- endYear (YYYY) – Representa o ano de fim das séries televisivas. O caractere '\N' é usado em todos os outros tipos de títulos;
- runtimeMinutes – tempo de duração primário, em minutos;
- genres (string array) – Inclui até três tipos de género associados ao título;

O ficheiro possui um formato característico, denominado *Tab-separated values (TSV)*, formatado de acordo com a codificação binária de comprimento variável *UTF-8*. Qualquer campo em falta ou nulo contém o caractere '\N'.

3.2 Solução

A solução, desenvolvida com recurso a duas tecnologias diferêntes, nomeadamente a ferramenta *Pentaho Data Integration (PDI)* e de seguida com recurso à linguagem de programação *C#* e à *.NET Core*, possui a seguinte sequência de funcionamento:

- Acessar à interface da *Internet Movie Database (IMDb)* e efetuar o *download* da respetiva compilação;
- Leitura dos dados, por linha e por campo, sendo efetuada limpeza por via de expressões regulares a dados desnecessários;
- Transformação dos campos nulos '\N' numa string "null"e remoção de caracteres *UTF-8* nulos existentes, susceptíveis de causar interrupções no processo;
- Ordenação dos dados por tipo de título;
- Separação dos dados por tipo de título: Filme (com lançamento superior a 2007 e duração superior a 75 minutos), Curta Metragem (com lançamento superior a 2007 e duração inferior a 15 minutos) ou Série Televisiva (com lançamento superior a 2007);
- Exportação dos dados para os respetivos ficheiros *.xml*;
- Envio de email para o administrador do sistema com o respetivo resultado da operação (Efetuada ou Não Efetuada);

A solução contempla também um *frontend* para visualização dos dados, necessitando de ambos os dados e a página estarem a correr num servidor local (e.g. XAMPP). Adicionalmente, pode ser definido um periodo de atualização dos dados (e.g. diário, semanal), sendo que de acordo com a *Internet Movie Database (IMDb)*, os dados da interface são atualizados diáriamente.

3.2.1 Arquitetura da Solução

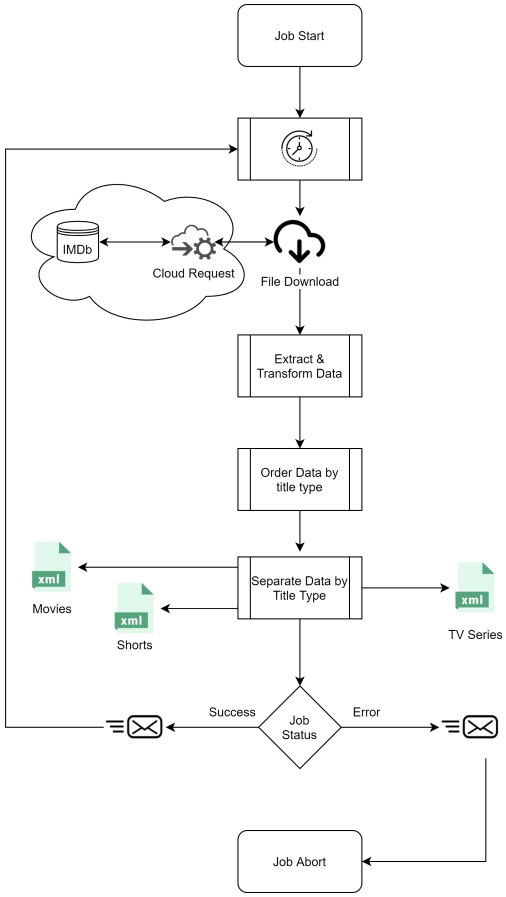

Figura 1: Arquitetura do Sistema

3.2.2 Solução em *Pentaho Data Integration (PDI)*

A solução desenvolvida em *PDI* envolve a criação de *Jobs* e *Transformations*, existindo um *Job main* responsável pela sequência lógica da extração e transformação, e consequente atualização de dados à data e hora definidos.

O *Job main* contém algumas variáveis, válidas em toda a máquina virtual do Java, de modo a manter a aplicação coesa. A sequenciação cria também as pastas necessárias para desenvolver o processo, caso não existam.

De seguida, um segundo *Job* é chamado para solicitar à *IMDb* o ficheiro correspondente para a pasta temporária do *windows*. É depois movido para a pasta da aplicação, e é iniciada uma *Transformation*, responsável pela extração e transformação dos dados, de acordo com as definições descritas na secção 3.2. Numa fase inicial, o ficheiro é carregado via *GZip*, ficando a extração a cargo do *PDI*.

Como estamos a lidar com grandes quantidades de dados, a máquina virtual do Java não dispõe de memória virtual suficiente para lidar com o ficheiro, sendo necessário aceder às definições do *PDI* e editar a seguinte variável de ambiente atualizando os valores:

```
PENTAHO_DI_JAVA_OPTIONS="-Xms4096m" "-Xmx8192m"
```

Após terminado o carregamento do ficheiro para memória, é efetuado um tratamento aos dados, por via de expressões regulares, de modo a transformar os *ID's* dos títulos em inteiros e remover caracteres capazes de gerar erros no processo, como é o caso dos caracteres nulos da codificação *UTF-8*. O campo *Genres* possui até três géneros diferêntes, separados por vírgula, sendo sepadados em 3 *tag's* com o mesmo nome *genre*, de modo a facilitar a integração com

frontend. Os dados rejeitados pela sequência são alvo de *Memory Dump*; Geram-se então os três ficheiros *.xml*, contendo os dados necessários para uma posterior integração (e.g. aplicação mobile, aplicação web).

Mediante o sucesso ou insucesso desta operação, é enviado um email ao administrador do sistema com o resultado da mesma, sendo que em caso de erro, o *Job main* é abortado. É de referir que todo o processo do *Job main* consegue ser completado em menos de dois minutos, dependendo da largura de banda disponível no momento do *download*.

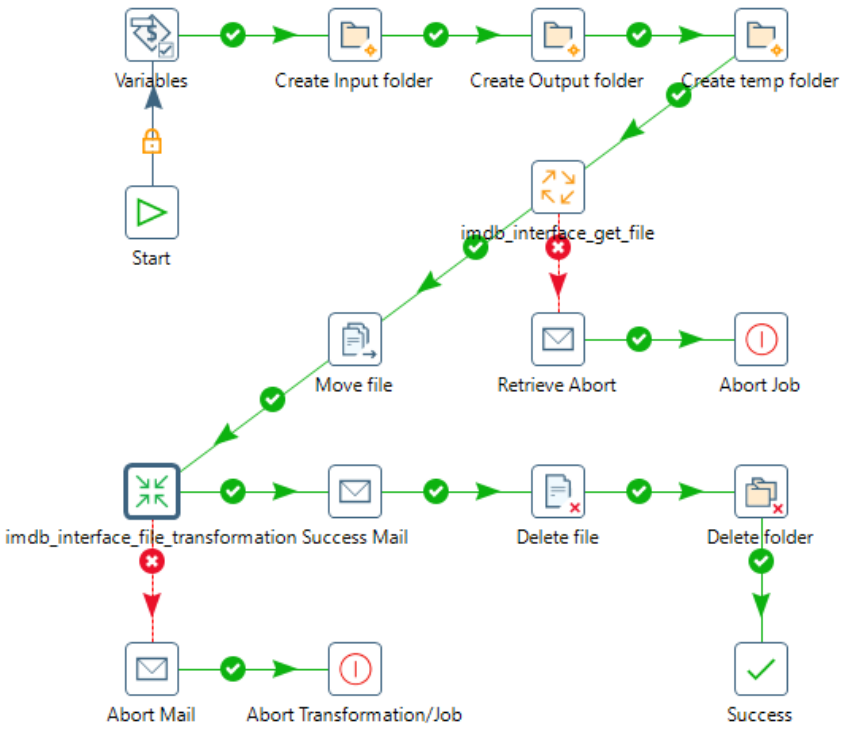

Figura 2: PDI - Main Job

Figura 3: PDI - IMDb Download Request Job

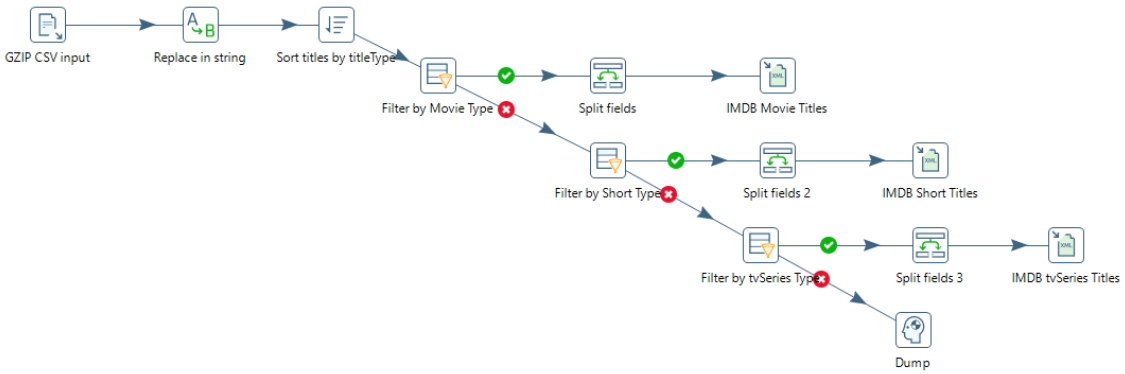

Figura 4: PDI - Data Transformation

Excerto do resultado obtido no ficheiro *imdb-tvSeries-titles.xml*:

```
<?xml version='1.0' encoding='UTF-8'?>
<Titles>
    <Title>
        <id>9916380</id>
        <type>tvSeries</type>
        <tName>Meie aasta Aafrikas</tName>
        <isAdult>0</isAdult>
        <year>2019</year>
```

```
        <endYear>null</endYear>
        <rtMin>43</rtMin>
        <genre>Adventure</genre>
        <genre>Comedy</genre>
        <genre>Family</genre>
    </Title>
    <Title>
        <id>9916216</id>
        <type>tvSeries</type>
        <tName>Kalyanam Mudhal Kadhal Varai</tName>
        <isAdult>0</isAdult>
        <year>2014</year>
        <endYear>2017</endYear>
        <rtMin>22</rtMin>
        <genre>Romance</genre>
        <genre/>
        <genre/>
    </Title>
    <Title>
        <id>9916206</id>
        <type>tvSeries</type>
        <tName>Nojor</tName>
        <isAdult>0</isAdult>
        <year>2019</year>
        <endYear>null</endYear>
        <rtMin>20</rtMin>
        <genre>Fantasy</genre>
        <genre/>
        <genre/>
    </Title>
</Titles>
```

Quanto ao frontend, foi criada uma solução bastante simples apenas para exemplificar uma procura por um determinado filme, sabendo à partida o tipo de título. Dada a limitação de

segurança dos *browsers*, é extremamente complexo e dispendioso trabalhar com ficheiros *.xml ou .json*, complexidade essa que aumenta conforme o tamanho dos ficheiros. A solução para este problema passaria por inserir os dados numa base de dados. No entanto, a título exemplificativo, a imagem seguinte corresponde à solução de pesquisa de filmes nos ficheiros gerados:

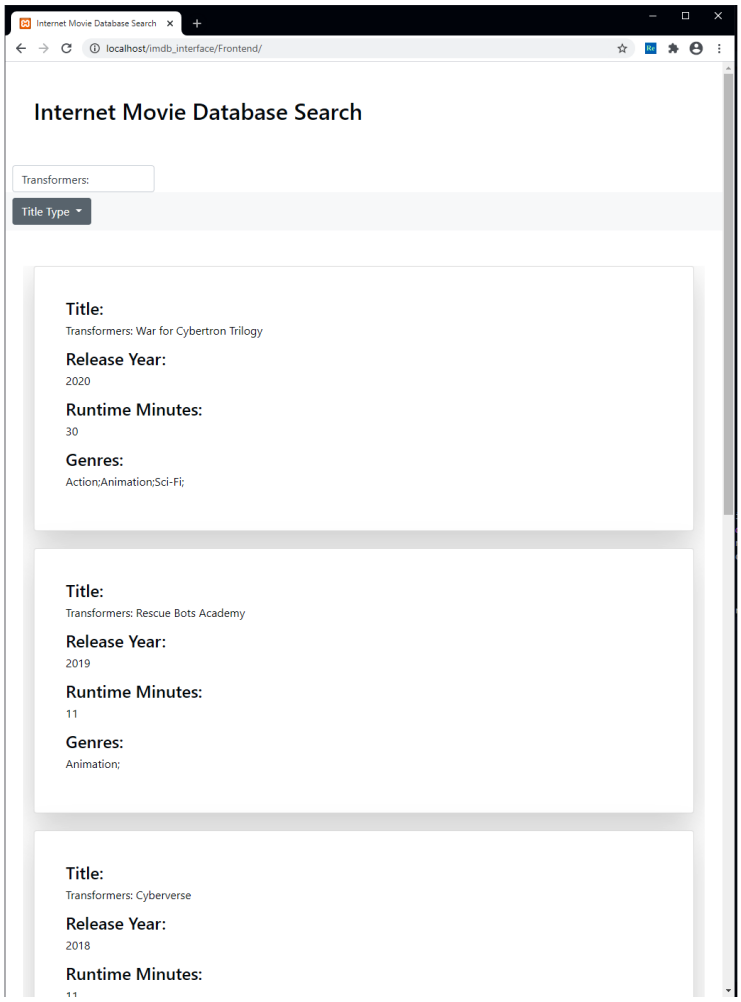

Figura 5: Aplicação de pesquisa de filmes

3.2.3 Solução em *.NET Core*

Na solução *IMDB_ISI* optamos por desenvolver dois projetos em *Asp .Net Core*.

De entre os projetos implementados, o primeiro, *DATA_to_API_Job* foi realizado recorrendo a um template denominado *Worker Service*, isto é, um caso de uso perfeito para qualquer processamento a decorrer em segundo plano. Deste modo, é possível destacar os componentes ETL.

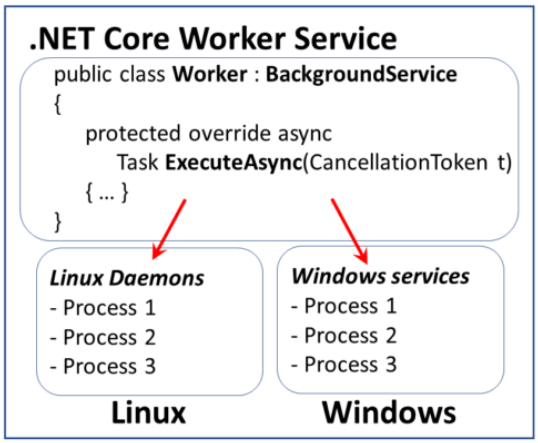

Figura 6: Diagrama Worker

Program and BackgroundService

A classe *Program.cs* é constituída pelos métodos *Main()* e *CreateHostBuilder()*, como é possível visualizar na seguinte imagem:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHostedService<Worker>();
            }); // IHostBuilder
}
```

Figura 7: Classe Program

O Main, é responsável por chamar o método *CreateHostBuilder()*, que o constrói e executa. Relativamente ao método *CreateHostBuilder()*, tem como intuito criar o *Host* e posteriormente efetuar a sua configuração chamando *AddHostService<T>*, onde T é um *IHostedService*, por exemplo, uma classe de trabalho que é "filho"de BackgroundService.

A classe de trabalho, *Worker.cs*, é definida conforme mostra a imagem seguinte: A classe *Worker*, tem como objetivo implementar a classe *BackgroundService*, que vem do *namespace Microsoft.Extensions.Hosting*. Esta poderá substituir o método *ExecuteAsync()*, de modo a realizar qualquer tarefa de longa duração.

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;

    public Worker(ILogger<Worker> logger)
    {
        _logger = logger;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {

    }
}
```

Figura 8: Classe Worker

Na função ExecuteAsync, iniciamos o processo declarando o destino do nosso ficheiro, *imdb.titles.tsv.gz*, que será descarregado do website da *IMDB*. Criada a pasta Download entraremos num ciclo, até que o *Token* seja cancelado. Dentro desse ciclo é inicializado um *WebClient()* que tem como funcionalidade receber o endereço do ficheiro, para posteriormente efetuar o *download* do mesmo para a pasta declarada acima.

Realizado o download do ficheiro, prosseguimos com a descompressão do mesmo, pelo que foi necessário criar uma classe chamada *Descompress* com o método *GZ*.

O método assíncrono *GZ* recebe de parâmetros, a informação do ficheiro (FileInfo), que efetua a leitura do mesmo. De seguida, é declarada, o nome do ficheiro atual e final em variáveis distintas. Depois é criado um ficheiro vazio com o nome do novo ficheiro declarado anteriormente, bem como, feita a descompressão do mesmo com a função *GZipStream()*.

```
public static class Descompress
{
    public static async Task GZ(FileInfo fileToDecompress)
    {
        await using (var originalFileStream = fileToDecompress.OpenRead())
        {
            var currentFileName :string = fileToDecompress.FullName;
            var newFileName :string = currentFileName.Remove(currentFileName.Length - fileToDecompress.Extension.Length);

            await using (var decompressedFileStream = System.IO.File.Create(newFileName))
            {
                await using (var decompressionStream = new GZipStream(originalFileStream, CompressionMode.Decompress))
                {
                    await decompressionStream.CopyToAsync(decompressedFileStream);
                }
            }
        }
    }
}
```

Figura 9: Classe Descompress

Prosseguimos com a desserialização do ficheiro, descompressado anteriormente. Neste sentido, foi necessário enviarmos como argumento o caminho para dentro do método *ReadFile*, localizado dentro da classe *Deserialize*.

No método *ReadFile*, começamos por ler o ficheiro descompressado, sendo posteriormente enviado para a função *CsvReader* (função mencionada é externa, teve de ser instalada pelos NuGet). De seguida, informamos que o presente ficheiro tem um cabeçalho, bem como , que o seu delimitador é “\t” (tab).

Após todo o procedimento mencionado, guardamos tudo num *IEnumerable* do tipo "Data"(Classe declarada na pasta Models), percorrendo, de seguida, o ficheiro recorrendo a um “foreach” que receberá por parâmetro a variável *IEnumerable*.

No *foreach* , realizamos pesquisas nas linhas e procedemos às substituições necessárias utilizando o Regex. Depois, adicionanamos tudo a uma nova lista que será guardada em memória.

Efetuado o carregamento total do ficheiro para a memória, este é libertado dos recursos,

retornando a lista dos filmes.

```
public static class Deserialize
{
    public static async Task<List<Data>> ReadFile(string path)
    {
        var textReader = System.IO.File.OpenText(path: path + "\\imdb.tsv");
        var csv = new CsvReader(textReader);

        csv.Configuration.HasHeaderRecord = true;
        csv.Configuration.Delimiter = "\t";

        var listFilmesTemp :IEnumerable<Data> = csv.GetRecords<Data>();

        var listFilmes = new List<Data>();


        foreach (var filme :Data in listFilmesTemp)
        {
            filme.tconst = Regex.Replace(input: filme.tconst, pattern: "([a-zA-Z]+)", replacement: "");
            filme.primaryTitle = Regex.Replace(input: filme.primaryTitle, pattern: "[\\u0000]+", replacement: "");
            filme.originalTitle = Regex.Replace(input: filme.originalTitle, pattern: "[\\u0000]+", replacement: "");
            filme.startYear = Regex.Replace(input: filme.startYear, pattern: "([\\\\][N])", replacement: "null");
            filme.runtimeMinutes = Regex.Replace(input: filme.runtimeMinutes, pattern: "([\\\\][N])", replacement: "null");
            filme.genres = Regex.Replace(input: filme.genres, pattern: "([\\\\][N])", replacement: "null");
            filme.endYear = Regex.Replace(input: filme.endYear, pattern: "([\\\\][N])", replacement: "null");

            listFilmes.Add(filme);
        }

        textReader.Dispose();
        csv.Dispose();

        return listFilmes;
    }
}
```

Figura 10: Desserialização

Após realizado todo processo anteriormente descrito, é possível:

- Guardar todos os filmes em JSON;
- Guardar todos os filmes em XML;
- Guardar todos os filmes em Base de dados usando uma API;

```
await Helpers.File.SaveJSON(listFilme);
await Helpers.File.SaveXML(listFilme);
await API.postFilmeAPI(listFilme);
```

Figura 11: Serialização

Para guardar em JSON/XML criamos uma classe chamada *File* com 2 métodos, *SaveJSON* e *SaveXML*, tanto um como o outro recebe como argumentos a lista de filmes *listFilme*.

Por exemplo o método "SaveJSON"temos de começar por criar o ficheiro JSON vazio e de seguida fazer a Serialização da lista dos filmes para dentro do ficheiro. Para o XML é exatamente igual.

```
public static class File
{
    public static async Task SaveJSON(List<Data> listFilmes)
    {
        await using var json :FileStream = System.IO.File.Create( path: "resultado.json");
        await JsonSerializer.SerializeAsync(json, listFilmes);
        await json.DisposeAsync();
    }

    public static async Task SaveXML(List<Data> listFilmes)
    {
        var fs = new FileStream( path: "resultado.xml", FileMode.Create);
        var xs = new XmlSerializer(typeof(List<Data>));
        xs.Serialize(fs, listFilmes);
        fs.Close();
        await fs.DisposeAsync();
    }
}
```

Figura 12: Task JSON

Para guardar os filmes na base de dados utilizando uma API, foi necessário criar a classe "API". Esta contém um método chamado *postFilmeAPI*, cuja função é receber como parâmetros a lista de filmes.

No início do método, introduzimos os endereços da API, efetuado um ciclo *foreach* à lista de filmes, logo de seguida. Ciclo onde é declarada uma variável "filme"que irá ser transportada para a API, através do método *httpClient*. Método que recebe de instrução um objeto e o url da API.

```
public static async Task postFilmeAPI(IEnumerable<Data> dataFilmes)
{
    var urlFilmes = "http://localhost:59449/api/Filmes";
    var urlGeneros = "http://localhost:59449/api/Generos";
    var urlFilmeGeneros = "http://localhost:59449/api/FilmeGeneroes";

    foreach (var dataFilme in dataFilmes)
    {
        if (dataFilme.startYear == "null")
            dataFilme.startYear = null;

        if (dataFilme.endYear == "null")
            dataFilme.endYear = null;

        if (dataFilme.runtimeMinutes == "null")
            dataFilme.runtimeMinutes = null;

        var filme = new Filme
        {
            TitleType = dataFilme.titleType,
            PrimaryTitle = dataFilme.primaryTitle,
            OriginalTitle = dataFilme.originalTitle,
            IsAdult = Convert.ToBoolean(Convert.ToInt32(dataFilme.isAdult)),
            StartYear = Convert.ToInt32(dataFilme.startYear),
            EndYear = Convert.ToInt32(dataFilme.endYear),
            RuntimeMinutes = Convert.ToInt32(dataFilme.runtimeMinutes),
            Tconst = Convert.ToInt32(dataFilme.tconst)
        };

        await httpClient(filme, urlFilmes);

        var genres :string[] = dataFilme.genres.Split( separator: ',');

        foreach (var genre :string in genres)
        {
            await httpClient(genre, urlGeneros);

            var filGen = new FilmeGeneroTemp
            {
                Genero = genre,
                Tconst = filme.Tconst
            };
            await httpClient(filGen, urlFilmeGeneros);
        }
    }

}
```

Figura 13: POST File API

Este método converte o objeto para JSON, e de seguida, converte o mesmo para *String* realizando um *Post* à API.

```
private static async Task httpClient(object o, string url)
{
    using (var client = new HttpClient())
    {
        var json :string = JsonConvert.SerializeObject(o);
        var data = new StringContent(json, Encoding.UTF8, mediaType: "application/json");

        await client.PostAsync(url, data);
    }
}
```

Figura 14: HTTP Client

API

API, neste projeto, apenas está a estabelecer a ligação há base de dados em *SQL Server*, também modelada por nós.

Para declarar as tabelas da base de dados na API, optamos por utilizar um comando que permita efetuar a migração automática e a criação do contexto da mesma.

```
public partial class IMDBContext : DbContext
{
    public IMDBContext() {}

    public IMDBContext(DbContextOptions<IMDBContext> options) : base(options) {}

    public virtual DbSet<Filme> Filme { get; set; }
    public virtual DbSet<FilmeGenero> FilmeGenero { get; set; }
    public virtual DbSet<Genero> Genero { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
            optionsBuilder.UseSqlServer( connectionString: "Data Source=DESKTOP-B8TO0CN;Initial Catalog=ISI_IMDB;Integrated Security=True");
        }
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Filme>(entity =>
        {
            entity.HasKey(e :Filme => e.Tconst);

            entity.Property(e :Filme => e.Tconst)
                .HasColumnName("tconst")
                .ValueGeneratedNever();

            entity.Property(e :Filme => e.EndYear).HasColumnName("endYear");

            entity.Property(e :Filme => e.IsAdult).HasColumnName("isAdult");

            entity.Property(e :Filme => e.OriginalTitle).HasColumnName("originalTitle");

            entity.Property(e :Filme => e.PrimaryTitle).HasColumnName("primaryTitle");

            entity.Property(e :Filme => e.RuntimeMinutes).HasColumnName("runtimeMinutes");

            entity.Property(e :Filme => e.StartYear).HasColumnName("startYear");

            entity.Property(e :Filme => e.TitleType)
```

Figura 15: Database Context

Foram criados 3 controladores, um para cada tabela: Filmes, FilmesGeneroes, Generos. Em cada controlador podemos contar com o mínimo de 5 métodos, entre eles: *Get; Post; Getid; Putid; Deleteid*.

O método *Get*, tem como funcionalidade retornar todos os elementos registados na base de dados.

```
// GET: api/Generos
[HttpGet]
public async Task<ActionResult<IEnumerable<Genero>>> GetGenero()
{
    return await _context.Genero.ToListAsync();
}
```

Figura 16: GET API Genero

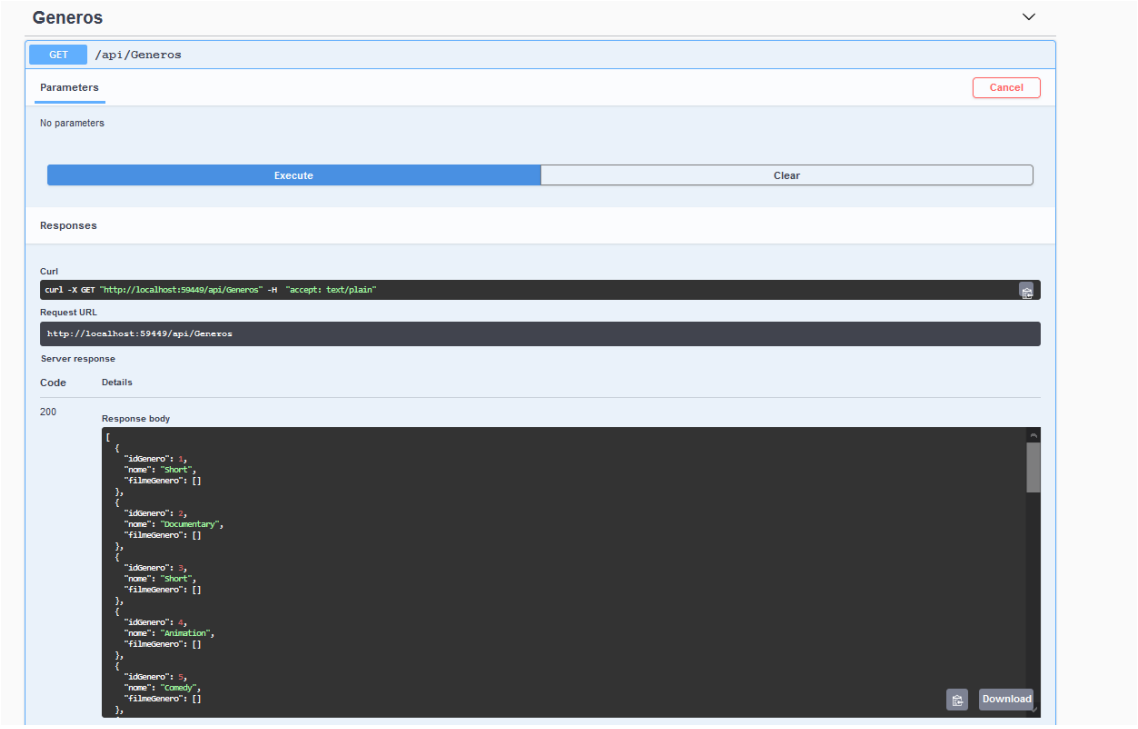

Figura 17: GET API Genero - 2

O método *Post*, tem como intuito receber, por argumento, uma classe/objeto para que possamos inserir na base de dados.

```
// POST: api/Filmes
// To protect from overposting attacks, enable the specific properties you want to bind to, for
// more details, see https://go.microsoft.com/fwlink/?linkid=2123754.
[HttpPost]
public async Task<ActionResult<Filme>> PostFilme([FromBody] Filme filme)
{
    await _context.Filme.AddAsync(filme);
    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateException)
    {
        return Conflict();
    }

    return Ok();
}
```

Figura 18: POST API Filme

O método *Get/id*, retorna apenas 1 elemento, caso exista.

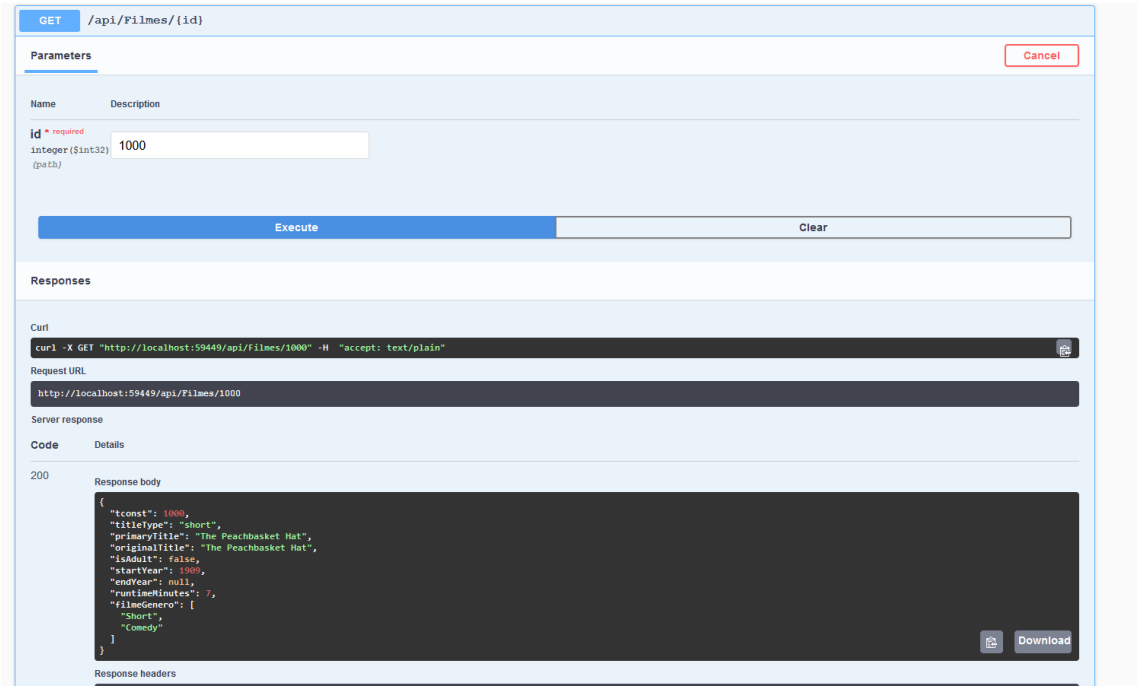

Figura 19: GET API ID

```
// GET: api/Filmes/5
[HttpGet(template: "{id}")]
public async Task<ActionResult<FilmeGet>> GetFilme(int id)
{
    var filme = await _context.Filme.FindAsync(id);

    if (filme == null)
    {
        return NotFound();
    }

    var generos = new List<string>();

    foreach (var filmeGenero in _context.FilmeGenero.Where(x :FilmeGenero => x.Tconst == id).ToList())
    {
        var gen :Genero = await _context.Genero.FirstOrDefaultAsync( predicate: y :Genero => y.IdGenero == filmeGenero.IdGenero);

        generos.Add(gen.Nome);
    }

    var filmeGet = new FilmeGet
    {
        OriginalTitle = filme.OriginalTitle,
        TitleType = filme.TitleType,
        PrimaryTitle = filme.PrimaryTitle,
        IsAdult = filme.IsAdult,
        StartYear = filme.StartYear,
        EndYear = filme.EndYear,
        RuntimeMinutes = filme.RuntimeMinutes,
        Tconst = id,
        FilmeGenero = generos
    };

    return filmeGet;
}
```

Figura 20: GET API ID - 2

O método *Put/id*, recebe por argumento um id e uma classe/objeto, tornando possível localizar, na base de dados, o registo a editar, e editando-o conforme a classe/objeto recebida.

```
// PUT: api/Generos/5
// To protect from overposting attacks, enable the specific properties you want to bind to, for
// more details, see https://go.microsoft.com/fwlink/?linkid=2123754.
[HttpPut(template: "{id}")]
public async Task<IActionResult> PutGenero(int id, Genero genero)
{
    if (id != genero.IdGenero)
    {
        return BadRequest();
    }

    _context.Entry(genero).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!GeneroExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

Figura 21: PUT Method API

O método *Delete/id*, recebe por argumento um id e posteriormente localiza-o, na base de dados, eliminando-o caso não aja outras relações. No caso de existirem, este irá eliminá-las primeiro.

```
// DELETE: api/Filmes/5
[HttpDelete(template: "{nome}")]
public async Task<ActionResult<Filme>> DeleteFilme(string nome)
{
    var filme = await _context.Filme.FirstAsync(predicate: x :Filme=>x.OriginalTitle == nome);
    if (filme == null)
    {
        return NotFound();
    }

    var generos :FilmeGenero[] = _context.FilmeGenero.Where(x :FilmeGenero => x.Tconst == filme.Tconst).ToArray();

    foreach (var genero in generos)
    {
        _context.FilmeGenero.Remove(genero);
        await _context.SaveChangesAsync();
    }

    _context.Filme.Remove(filme);
    await _context.SaveChangesAsync();

    return filme;
}
```

Figura 22: DELETE Method API

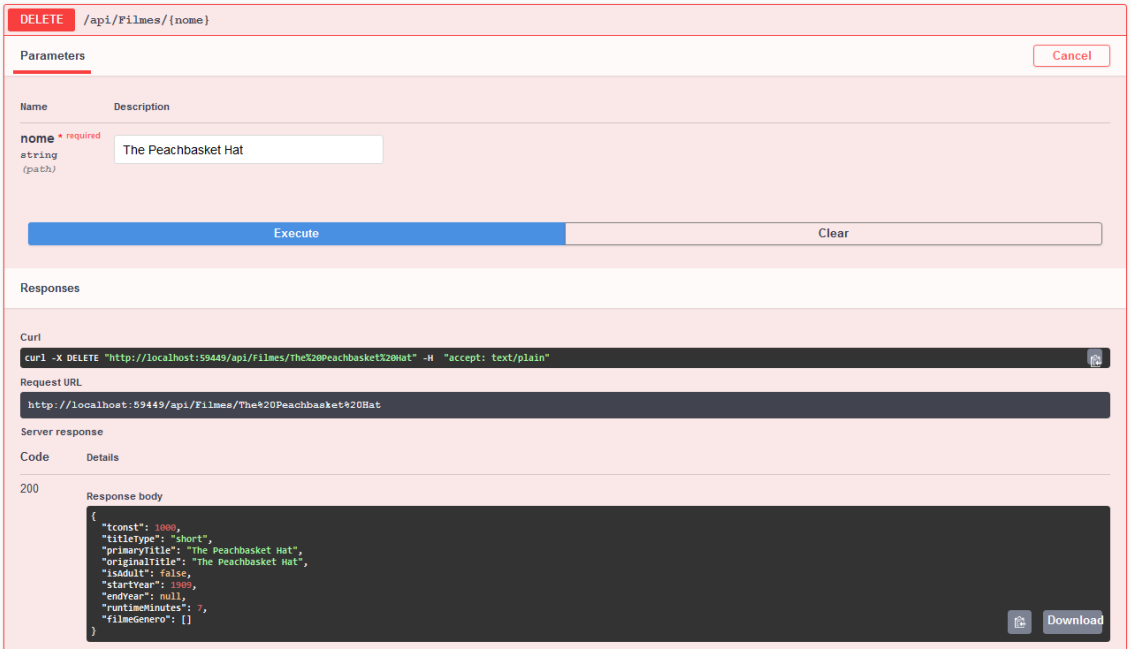

Figura 23: DELETE Method API - 2

4 Análise e Testes

De modo a validar a aplicação é realizado de seguida um conjunto de testes. Estes testes tem por base a transformação e extração dos dados providenciados pela interface da *IMDB - Internet Movie Database*, com base nos critérios definidos e exportação dos mesmos para ficheiros XML. No final, ambas as soluções devem ser capazes de desempenhar similarmente as funções descritas.

4.1 Solução em *Pentaho Data Integration (PDI)*

4.1.1 Casos de teste

Tabela 1: PDI - Casos de Teste

#ID	Passo	Ação	Resultado Esperado	Resultado Obtido	Correção
1	Acesso à interface da IMDb e respetivo download do ficheiro.	Executar o Main Job a partir do PDI.	Download do ficheiro e respetiva movimentação para a pasta Input, dentro da diretoria do projeto.	Ficheiro obtido e movido para a respetiva pasta.	
2	Abertura do ficheiro e carregamento dos dados para memória.	Executar o Main Job a partir do PDI.	Dados do ficheiro obtido compeltamente carregados para memória no PDI, separados por campos obtidos.	Falha ao carregar os dados. A máquina virtual do Java não dispõe de memória suficiente para o completo carregamento dos dados.	Aumentar o limite máximo de memória virtual disponível até 8GB.

3	Correta extração dos dados e posterior transformação, com base em expressões regulares.	Executar o Main Job a partir do PDI.	Tabela com os dados transformados, de acordo com os critérios definidos.	Falha ao efetuar a transformação de uma linha do ficheiro original, devido à existência de um caractere especial não identificado.	Remoção do caractere codificado em UTF-8, correspondente a nulo - \u0000.
4	Filtragem dos dados de acordo com o tipo de título.	Executar o Main Job a partir do PDI.	Obtenção de duas tabelas distintas com a correta listagem de dados.	Listagem completa e correta dos dados.	
5	Exportação dos respetivos ficheiros .xml, contendo os dados filtrados.	Executar o Main Job a partir do PDI.	Um ficheiro respetivo aos títulos cinematográficos e um ficheiro respetivo aos títulos televisivos.	Os ficheiros foram gerados corretamente.	
6	Envio de email ao administrador do sistema com informação acerca do resultado da operação de extração, transformação e conversão dos dados.	Executar o Main Job a partir do PDI.	O administrador recebe um email com o resultado da operação.	Envio correto do email com a informação detalhada acerca de todo o processo.	

5 Conclusão

Neste capítulo final é abordado o problema como um todo, bem como a solução final obtida.

5.1 Lições aprendidas

A integração de dados forneceu uma solução prática para um problema complexo. Hoje em dia conseguimos utilizar uma aplicação, e efetuar o envio de dados para outra aplicação completamente diferente, mediante um conjunto de padrões associados.

No entanto, e dado o elevado rísco de segurança, trabalhar com dados *online* é um trabalho bastante difícil e complexo, uma vez que potencia a exploração de falhas de segurança, *i.e. injeção de código....* Nesse seguimento, as ferramentas *ETL* tornam-se uma necessidade, não só pela redução no tempo dispendido a criar a solução, mas também a extrair informação e transformar a mesma.

Associado à redução do trabalho associado à criação da solução, surge também a automatização do processo *ETL*, podendo este funcionar quase que em tempo real, se a solução assim o requerer.

É de destacar aínda a simplicidade no processo da criação da solução, dada a quase inexistência de necessidade de criação de código, quando comparado com uma solução criada de raíz.

Por fim, o resultado da aprendizagem é bastante satisfatório.

5.2 Trabalho futuro

Como trabalho futuro, seria interessante a possibilidade de inserção dos dados numa base de dados e a criação de um *frontend* que permita a procura de um filme pelo título do mesmo, ao par da utilização da interface de programação de aplicação da *Google* para pesquisa de imagem de capa de título correspondente.

5.3 Apreciação final

Numa última nota é de salientar que o desenvolvimento deste projeto foi concluído com sucesso, tendo ambas as soluções mostrado ótimos resultados.

Numa primeira fase do projeto foram definidos os objetivos do projeto e criada a arquitetura da solução, bem como a definição das tecnologias a utilizar.

Tendo a estratégia já bem delineada, partiu-se para o desenvolvimento de duas soluções recorrendo a tecnologias distintas, de modo a ser possível efetuar uma comparação entre elas.

Na terceira fase, realizamos a deteção e correção de erros por via de testes às soluções.

Referências

[1] Ralph; Caserta, Joe; Kimball. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Wiley, Indianapolis, Indiana, 2004.

[2] Jim; Ras. *Etl - Extract, Transform, Load*. Kim, 2018.

[3] Maria; Meadows, Alex; Roldan. *PENTAHO DATA INTEGRATION COOKBOOK - SECOND EDITION*. Packt Publishing, Birmingham, United Kingdom, 2013.

[4] María; Roldán. *PENTAHO DATA INTEGRATION QUICK START GUIDE*. Packt Publishing, Birmingham, United Kingdom, 2018.

[5] Erik T.; Ray. *Learning XML, Second Edition*. O'Reilly Media, Sebastopol, California, 2003.

[6] Ben; Albahar, Joseph; Albahari. *C-sharp 8.0 Pocket Reference: Instant Help for C-sharp 8.0*. O'Reilly UK Ltd., Farnham, United Kingdom, 2019.

[7] HITACHI. Pentaho Data Integration. `https://help.pentaho.com/Documentation/8.2/Products/Data_Integration`. (accessed: 06.11.2020).

[8] Neil Charles. Batch downloading files with Pentaho Kettle / PDI. `https://www.joyofdata.de/blog/batch-downloading-files-with-pentaho-kettle/`. (accessed: 06.11.2020).

[9] Microsoft Docs. GZipStream Class. `https://docs.microsoft.com/en-us/dotnet/api/system.io.compression.gzipstream?view=net-5.0`. (accessed: 07.11.2020).

[10] IMDb Datasets. IMDb Dataset Details. `https://www.imdb.com/interfaces/`. (accessed: 04.11.2020).