PROGRAMMING CHALLENGE

Robson Cruz

**Introdução**

Primeiro passo foi clonar o repositório após o fork na máquina local. Usando o comando checkout foi criado dois branches complementares ao master: test e dev. A estratégia a ser adotada é de usar o /dev para as atualizações de desenvolvimento e /test com as versões prontas para o teste. O branch principal, /master, receberá as atualizações após testadas. Assim, em relação a estabilidade, /master > /test > /dev.

O desenvolvimento seguiu as etapas:

- Analisar os dados fornecidos

- Criar o modelo estrutural do banco de dados

- Criar uma Rest API genérica em Django

-Criar script para salvar dados de arquivos no banco de dados

- Adaptar a Rest API para fornecer os dados de acordo com o demandado

- Desenvolver um cliente para o consumo de dados

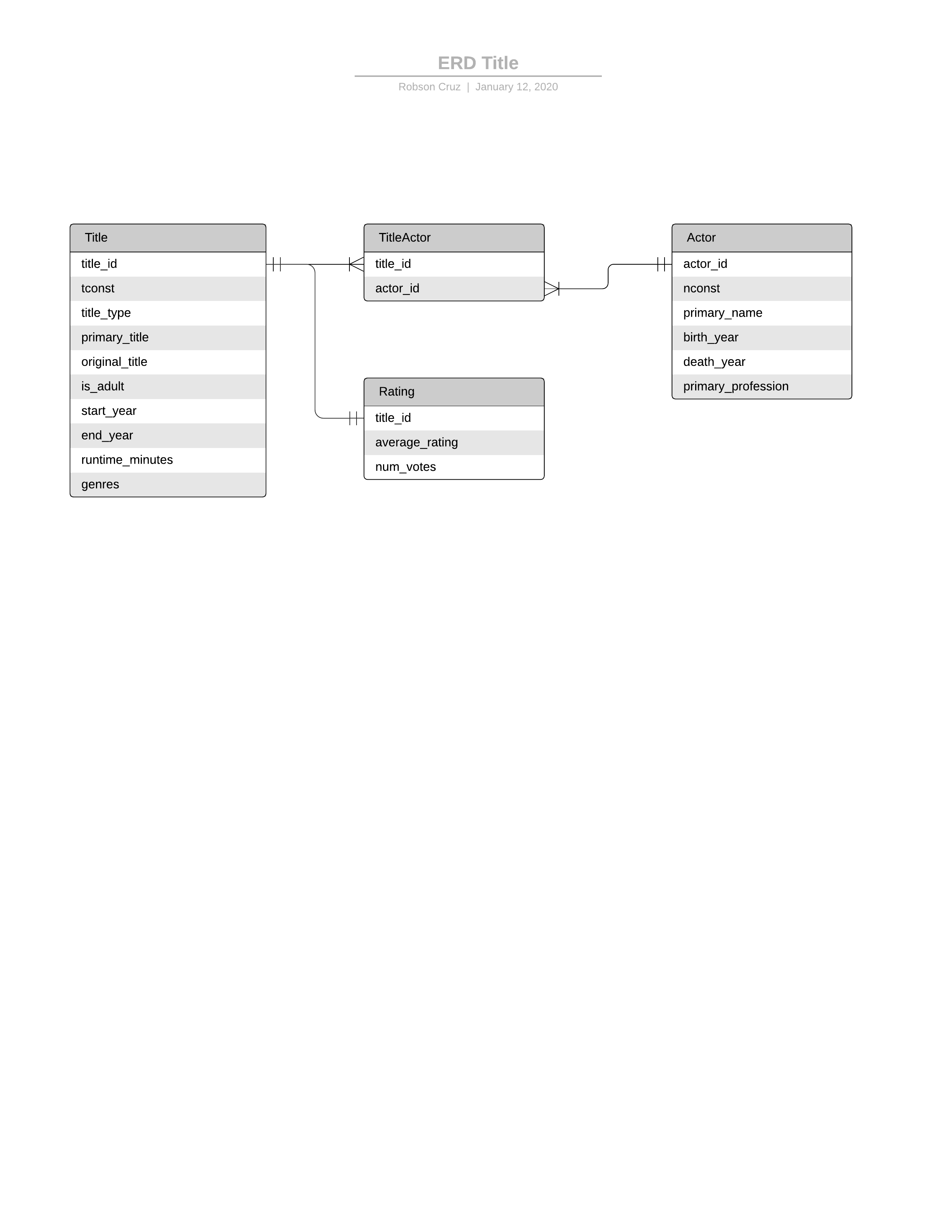
A familiaridade com o desenvolvimento web e com a linguagem Python levou a decisão de explorar uma possível solução com a framework Django e React.js. [Por que React?] Em Django/python, seria implementado a Rest API (Step 1 e 2) enquanto que o cliente, em React, acessaria os dados (Step 3). Porém, o uso do React depende do quão rápido poderá ser feita a integração com o Django. Se necessário, outras alternativas serão exploradas.

Para o desenvolvimento foram utilizadas a versão 3.7.3 da linguagem Python e a versão 3.0.2 do framework Django. O Python foi instalado a partir do instalador baixado no site oficial. No caso do Django foi usado um sistema de gerenciamento de pacotes, pelo qual a instalação foi realizada ao executar o comando **pip install django** no prompt de comandos do Windows. Quanto a especificação do SO: Windows 10 x64, versão 1809 e compilação do SO 17763.

O banco de dados PostgreeSQL instalado possui a versão 12.1, compilado com Visual C++ x64 build 1914. O pacote de instalação incluía outras aplicações de suporte, que também foram instaladas.

1. **Análise dos dados e modelo conceitual**

A partir da análise dos dados fornecidos, foi elaborado um diagrama ER para mapear as entidades e estabelecer um modelo relacional, uma vez que o Django permite estabelecer os relacionamentos durante a criação dos modelos base para posterior migração da estrutura para o banco de dados configurado. O diagrama é apresentado na Figura X.



Os dados apresentam as entidades relacionadas a Title e Actor, originários dos arquivos title.basics.tsv.gz e name.basics.tsv.gz, apresentam campos como identificadores únicos. Porém, como esses dados são do tipo String, optou-se por incluir outro campo como numérico, pois esse tipo permite melhor desempenho para o banco de dados. O campo de identificação original também será incluído para manter a referência aos dados de origem.

O campo knownForTitles, que consistia de um array de tconsts, deu lugar a uma tabela que auxilia na representação da relação N:N existente entre as entidades Title e Actor. Cada uma terá um relacionamento 1:N com a tabela auxiliar, a qual manterá o vínculo entre as tabelas Title e Actor. Dessa forma, é possível recuperar a informação proporcionada pelo campo original knownForTitles.

1. **Criação de Rest API em Django**

O material de referência usado para desenvolver a Rest API foi a documentação do Django disponível em https://docs.djangoproject.com/en/dev/. Para facilitar a transição entre da fase de desenvolvimento e a produção (deploy), optou-se por utilizar o PostgreSQL como banco de dados ao invés do SQLite que é default do Django, permitindo usar o mesmo banco de dados do ínicio ao fim do projeto. Além disso, a partir da criação dos modelos baseados nos dados fornecidos, esse framework facilita a criação das tabelas do banco de dados.

Para acessar o banco de dados PostgreeSQL com o python e Django, é necessário a configurar os dados de acesso no script settings.py, conforme observado na documentação. Esses dados devem estar de acordo com a instalação do PostgreSQL:

'default': {

        'ENGINE': 'django.db.backends.postgresql', #informado na documentação

        'NAME': 'titles', #nome do banco de dados da aplicação

        'USER': 'djangoapp', #usuário, senha, host e port configurados no PostgreSQL

        'PASSWORD': '@01123581321',

        'HOST': 'localhost',

        'PORT': '5432',

    }

O banco de dados (titles) foi criado manualmente usando a ferramenta pgAdmin4, que foi instalado junto com o PostgreSQL. Também foi criado o usuário djangoapp para a aplicação acessar o banco de dados. Os dados de host e port foram definidos durante a instalação do SGBD. A criação das tabelas ocorre durante o processo de desenvolvimento dos modelos.

Os modelos (classes) Title, Actor, TitleActor e Rating foram implementadas de acordo com o mapeado no diagrama ER, com campos adequados aos dados e relações. Os índices de Title e Actor são campos autoincrementados em BD. Isso foi possível atribuindo a eles models.AutoField(primary=True). Esses índices são tidos como chave estrangeira na tabela de TitleActor, o que foi realizado no modelo por:

* title\_id = models.ForeignKey('Title', on\_delete=models.CASCADE, to\_field='title\_id')
* actor\_id = models.ForeignKey('Actor', on\_delete=models.CASCADE, to\_field='actor\_id').

Como o relacionamento entre Title e Rating foi modelado como 1:1, o identificador de Title foi utilizado como chave também em Rating:

* title\_id = models.OneToOneField('Title', on\_delete=models.CASCADE, to\_field='title\_id', primary\_key=True).

Outros campos dos modelos receberam a designação mais adequada segundo o disponível no Django e o tipo de dados em questão.

A documentação oficial foi seguida até a disponibilização dos modelos na interface de administração incluída no Django, que é disponibilizada pelo seu próprio servidor. Por meio dela, é possível inserir e consultar dados nas tabelas Actor e Title manualmente. A Figura 1 apresenta a interface administrativa com os links onde é possível adicionar, remover ou editar Actors e Titles.

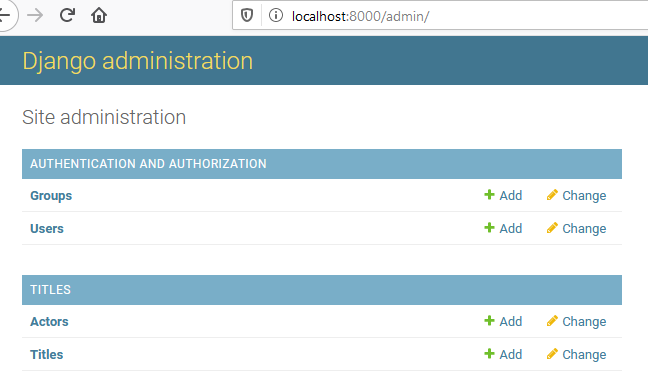


Figura 1: Print da janela com a ferramenta de administração do Django

Antes de completar o desenvolvimento da Rest API, é importante carregar dados dos arquivos no banco de dados para facilitar os testes durante o processo.

Após criados os modelos, foram executados os comandos para que o Django criasse as tabelas no banco de dados de acordo com o definido nos modelos de Title, Actor, Rating e TitleActor, resultando nas tabelas tbl\_title, tbl\_actor, tbl\_rating e tbl\_title\_actor no banco de dados titles, criado anteriormente no PostgreSQL.

1. **Criar script para salvar dados de arquivos no banco de dados**

O acesso ao PostgreeSQL foi realizado utilizando pelo módulo psycopg2 do Python, que pode ser instalado por comando no prompt: pip install psycopg2.

Para cada classe no modelo Django, foi criada uma nova classe de mesmo nome para realizar a transferência dos dados do arquivo para o banco de dados: Title, Actor, TitleActor e Rating. A classe Title é a responsável por carregar os dados do arquivo title.basics.tsv.gz, enquanto Actor e TitleActor carregam do arquivo name.basics.tsv.gz e Rating opera com o arquivo title.ratings.tsv.gz. Os arquivos .tsv compactados devem ser extraídos previamente e colocados na mesma pasta que o script load\_data\_file.py, que contém as classes e realiza as funções. Complementarmente, foi criada a classe DataBaseInterface para uso nas operações com o banco de dados.

De forma geral, os passos para salvar os dados dos arquivos foram de abrir os arquivos com a função open, processá-los e salvar nos formatos de acordo com o banco de dados.

A função open é embutida no Python. Como foram observados alguns caracteres incomuns na base de dados, foi incluído o parâmetro enconding=’utf-8’, garantindo que os dados fossem lidos em um formato apropriado.

Os dados foram processados a cada linha. O objeto gerado pela função open foi lido com a função reader da biblioteca csv do Python, que retorna um objeto com a propriedade de iteração, facilitando o processo seguinte. A iteração linha a linha ocorreu com o uso da estrutura for a partir da segunda linha, pois a primeira é de títulos das colunas. Alguns dos campos já apresentavam o formato adequado, esses foram copiados diretamente para as variáveis da classe para posterior uso.

Da classe Title, os campos ajustados foram os relacionados às colunas isAdult, startYear, endYear e genres. O campo isAdult é representado pelo boolean do Python (0 – false, 1 – true). Os campos startYear e endYear são avaliados se possuem valores válidos ou não, sendo convertidos para inteiros em caso verdadeiro e None no caso falso. O tratamento de genre gera um vetor com os valores separados por vírgula ou None quando possui /N.

Foram detectados o valor ‘/N’ em outras colunas além da deathYear informada. Nesses casos, a variável recebeu o valor None, que é convertido em NULL durante a conversão para o formato adequado do banco de dados.

Nas classes Actor, TitleActor e Rating, os valores tiveram tratamento similar ao que houve em Title, inclusive valores inválidos também seriam convertidos em NULL.

Uma vez processados, os valores das variáveis foram preparados usando a função mogrify, que pode ser acessada do objeto cursor do banco de dados. Esse objeto é obtido durante a conexão com o banco.

Foi observado durante testes para salvar todos os dados dos arquivos no banco de dados, que alguns campos de String extrapolaram o comprimento avaliado inicialmente. Para evitar mais mudanças, foi preparado uma função para determinar o maior comprimento por coluna.

Tabela 1 - Dados divergentes

|  |  |  |  |
| --- | --- | --- | --- |
| tabela | coluna | tamanho | novo tamanho |
| tbl\_title | title\_type | 12 | (mantido original) |
| primary\_title | 408 | 450 |
| original\_title | 408 | 450 |
| tbl\_actor | Primary\_name | 105 | 150 |

Os modelos tiveram ajustes segundo os valores mapeados na tabela 1. Após isso, executou-se novamente os comandos de migração para que as mudanças tivessem efeito no banco de dados. Após as mudanças, os dados foram submetidos ao banco de dados. Para evitar ‘estouro’ de memória durante a submissão, a cada 100 registros processados o script realiza a operação de commit dos dados para o banco.

A transferência de toda a massa de dados seria inviável de ser realizada pelo Python pela via convencional. Assim, foi adotado duas estratégias, a depender da tabela/classe, para assim conseguir testar o máximo possível de dados dos arquivos e ainda conseguir a transferência de todo o conjunto. As tabelas tbl\_title e tbl\_actor foram preenchidas normalmente, pois não precisavam realizar consultas no processo e a velocidade de transferência tornava viável. Assim, os dados eram transferidos conforme a leitura e processamento de cada linha dos arquivos.

Ainda que tivessem disso implementadas as funções para preencher as tabelas tbl\_title\_actor e tbl\_rating da mesma forma que as outras, por fim foram preenchidas de uma forma diferente, com outra função em duas etapas. Primeiro, criou-se as tabelas temporárias tbl\_title\_actor\_temp e tbl\_rating\_temp para receber os dados de referência para as buscas:

* tbl\_title\_actor\_temp: tconst, nconst
* tbl\_rating\_temp: tconst, average\_rating, num\_votes

Uma vez preenchidas as tabelas temporárias, a função executou no SGBD outro statement SQL para preencher as tabelas definitivas. Os statements utilizados são apresentados na Tabela 2 a seguir.

Tabela 2 - Instruções SQL auxiliares

|  |  |
| --- | --- |
| tbl\_title\_actor | INSERT INTO tbl\_title\_actor  SELECT tbl\_title\_actor\_temp.uid, tbl\_title.title\_id, tbl\_actor.actor\_id  FROM tbl\_title\_actor\_temp  INNER JOIN tbl\_title ON tbl\_title\_actor\_temp.tconst=tbl\_title.tconst  INNER JOIN tbl\_actor ON tbl\_actor.nconst=tbl\_title\_actor\_temp.nconst  WHERE tbl\_title.title\_id IS NOT NULL AND tbl\_actor.actor\_id IS NOT NULL |
| tbl\_rating | INSERT INTO tbl\_rating  SELECT title\_id, average\_rating, num\_votes  FROM tbl\_title  INNER JOIN tbl\_rating\_temp ON tbl\_title.tconst=tbl\_rating\_temp.tconst WHERE tbl\_title.tconst <> '' |

Essa última estratégia de preenchimento mesmo quebrada em duas etapas, foi bem mais rápida que na primeira forma. Isso porque utilizou apropriadamente o poder de manipulação e processamento de dados característico de um SGBD.

1. **Adaptar a Rest API para fornecer os dados de acordo com o demandado**

No passo 2, foi solicitado a implementação de pelo menos dois métodos. O primeiro, que listasse os títulos com seus dados relevantes filtrados por categoria. O segundo deveria listar os 10 filmes com maiores avaliações filtrados por ano, ou na falta desse parâmetro, todos os filmes com paginação. Ambos os casos não devem incluir filmes adultos ou com avaliação menor que 6.

A estrutura do sistema foi orientada a objetos. As classes criadas representam as entidades mapeadas anteriormente: Title, Actor, TitleActor e Rating. Com o objetivo de obter modelos mais flexíveis e adaptativos, com abstração de parte das operações do banco de dados, foi utilizado o Django Rest Framework. O desenvolvimento da API Rest se torna mesmo mecânica e mais conceitual, com a preparação dos modelos que representam os elementos necessários ao sistema. O framework apresenta diversas classes que oferecem especificidades que devem ser avaliadas durante a concepção. Foi seguido o procedimento de instalação e configuração mostrado na página https://www.django-rest-framework.org/. A documentação do framework foi uma referência com diversos exemplos.

Durante o desenvolvimento, 4 arquivos foram de maior atenção: views.py, models.py, serializers.py e urls.py. O views.py é onde se concentra os modelos que representam a forma como os dados serão entregues ao cliente. O model.py contém as classes e atributos que representam as entidades modeladas. O script serializers.py é onde foram criados os serializadores e deserializadores dos dados, que fazem a devida formatação tanto dos dados de entrada quanto de saída. O arquivo urls.py é onde ocorrem os vínculos entre endereços http e os dados que serão recebidos ou enviados no acesso, fazendo também o vínculo com o “view” correspondente.

As classes criadas em “models” são o núcleo de todo o restante construído. Essa representação é o que o Django exporta para o banco de dados em um processo chamado de migração. Os modelos e seus relacionamentos se tornam tabelas com as mesmas relações e restrições, ou seja, o schema do banco de dados.

A seguir são apresentadas parte das classes Title, Rating e TitleActor para observar de que forma vinculou-se segundo o modelo concebido.

class Rating(models.Model):

title\_id = models.OneToOneField('Title', on\_delete=models.CASCADE, to\_field='title\_id', primary\_key=True, related\_name='rating', )

class Title(models.Model):

title\_id = models.AutoField(primary\_key=True)

tconst = models.CharField(max\_length=20)

title\_type = models.CharField(max\_length=20)

class TitleActor(models.Model):

title\_id = models.ForeignKey('Title', on\_delete=models.CASCADE, to\_field='title\_id', related\_name='title')

actor\_id = models.ForeignKey('Actor', on\_delete=models.CASCADE, to\_field='actor\_id', related\_name='actor')

Segundo a modelagem conceitual, foi estabelecido uma relação 1:1 entre Title e Rating, que pode ser tratado incluindo a chave de uma das tabelas na outra de forma parecida como ocorre em 1:1 ou criar uma tabela entre as duas, como acontece em N:N. A opção foi incluir uma relação do índice de Title na tabela relacionada Rating (1:1 - models.OneToOneField).

A relação N:N entre Actor e Model deu origem a classe TitleActor, que recebe como chaves estrangeiras as chaves primárias de Title e Actor. As buscam que envolvam os dois modelos simultaneamente são intermediadas por esta classe.

Em relação aos “views”, a necessidade de dispor os dados em diferentes formas e contextos, levaram a criação de diversas “views classes”. Como ela faz a ponte entre cliente e banco de dados, ela quase sempre inclui um serializador de dados do modelo, o que foi o caso da grande maioria das “views” criadas. As views constroem a instrução SQL sequenciando chamadas de funções.

A seguir mostra-se o roteamento das páginas que são vinculadas aos “views”. Todos os caminhos foram configurados com a aplicação frontend.

urlpatterns = [

path('api/titles/', views.TitleList.as\_view()),

path('api/titles/types/', views.TypesList.as\_view()),

path('api/titles/types/filter/', views.ManyTypesList.as\_view({'post': 'list'})),

path('api/titles/type/<str:title\_type>/', views.TitleTypeList.as\_view()),

path('api/titles/genre/<str:genre>/', views.TitleGenreList.as\_view()),

path('api/titles/year/', views.TopList.as\_view({'get': 'list'})),

path('api/titles/year/<str:year>/', views.TopList.as\_view({'get': 'list'})),

path('api/titles/stat/title/', views.TitleCountView.as\_view()),

path('api/titles/stat/actor/', views.ActorCountView.as\_view()),

path('api/titles/stat/type/', views.TypeCountView.as\_view()),

path('api/titles/stat/completed/', views.CompletedDataView.as\_view()),

path('api/titles/stat/adult/', views.AdultCountView.as\_view()),

path('api/titles/stat/worst/', views.WorstRatingView.as\_view()),

path('admin/', admin.site.urls),

]

urlpatterns = format\_suffix\_patterns(urlpatterns)

Uma outra facilidade proporcionada pelo Django Framework é o gerenciamento da paginação, que melhorou a integração com o frontend e permite que ambos controlem e sincronizem as páginas em relação aos dados em cache da última consulta.

1. **Desenvolver um cliente para o consumo de dados**

O desenvolvimento do frontend fez reuso de template adquirido para desenvolvimento de outros projetos. Assim, a familiaridade poderia agilizar o desenvolvimento. Esse template é baseado em React.

Foi seguido o guia de instalação do fabricante para realizar a configuração inicial:

-instalar a última vesão do NodeJS: foi instalada versão v12.14.1.

-Instalar a última versão do Yarn (versão: 1.21.1).

Devido a alguns problemas de configuração do arquivos package.json , foi necessário instalar também - npm shrinkwrap – para que esse último os gerenciasse. Com a opção por usar no frontend um Template em React, o processo de desenvolvimento fez o reuso de componentes e estruturas de páginas que foram adaptadas para o atual projeto. As funcionalidades demandadas foram proporcionadas por duas páginas, ambas com tabela mais o acesso ao backend correspondente. Também foi implementado uma página Home com diversas informações extraídas automaticamente da do banco de dados povoado com a massa de dados fornecida.

Por ser um template, possui páginas gerais que procuram atender diversos tipos de projeto. Além das adaptações estruturais para preparar os componentes para o projeto atual, o acesso aos dados e o comportamento durante a interação com usuário foram implementados.

As páginas implementadas foram: Home Dashboard, Titles e Top List, sendo as duas últimas as que consomem os recursos da Rest API que constam como objetivos.

O gerenciamento da paginação foi um ponto forte da integração do Framework Django com o React: as páginas foram melhor controladas, não houve acúmulo dos muitos dados no Frontend causando travamentos, entre outras. A página TopList foi implementada de forma que as mesmas páginas corresponderão aos mesmos dados, o que foi proporcionado pela classe CursorPagination. A página de lista de títulos geral utilizou a classe PageView, de forma que os dados das páginas vão alterando, permitindo uma interação mais exploratória na lista.

Em relação as funcionalidades de objetivo, um filtro na primeira página permite filtrar os filmes pelo tipo, o que ocorre com consulta ao banco de dados. Os botões de ordenação e busca ocorrem somente sobre os dados locais (primeira página).

A segunda página, por padrão, carrega a lista de filmes ordenadas de forma decrescente pelo campo average\_rating. Porém, a coluna Start Year possui um campo de busca onde é possível filtrar pelo ano.

1. **Requisitos e Procedimentos para utilização da aplicação**

Como é de praxe em aplicações web com recursos como o Node.js, geralmente a pasta node\_modules não é incluída. Para utilizar o cliente a partir da clonagem do repositório, é necessário instalar as versões mais recentes do Node.js e Yarn, ambos gerenciadores de pacotes. Depois, com o prompt de comando na pasta /sidia\_project/frontend/, executar o comando: yarn install. Alguns erros podem ocorrer devido a alguns conflitos entre dependências, porém o necessário para executar a aplicação geralmente é instalado. Também foi instalado outra aplicação por: npm shrinkwrap.

Com relação ao banco de dados, foi adotado o PostgreSQL na versão mais recente baixado no site do fabricante: https://www.postgresql.org/download/. A base de dados utilizada foi criada manualmente na aplicação pgAdmin4, que na versão Windows, vem no mesmo pacote de instalação do PosgreSQL. O nome da base de dados deve ser: titles. Caso o nome da base de dados seja mudada, isso deve ser alterado também no script \programming-challenge\sidia\_project\djangoconfig\settings.py. Dados de acesso em settings:

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.postgresql',

        'NAME': 'titles',

        'USER': 'djangoapp',

        'PASSWORD': '@01123581321',

        'HOST': 'localhost',

        'PORT': '5432',

    }

}

As tabelas do banco serão criadas pela aplicação em Python/Django posteriormente.

Em relação ao backend, são requisitos ter o Python 3.7, o Django e o Rest Framework. O Django pode ser obtido pelo comando: pip install django. O Django Rest Framework, o comando deve ser: pip install djangorestframework.

Antes de testar, ainda devem ser geradas as tabelas no banco de dados. Deve ser executado com o prompt em /sidia\_project/: python manage.py makemigrations, depois python manage.py migrate. Nota: só surtirá efeito se configurado os dados de acesso corretamente, inclusive a base de dados “titles” deve estar criada no PostgreSQL.

Para fazer os testes da aplicação completa, deve-se executar ambos os servidores da aplicação.

-Backend: com Prompt de comando na pasta /sidia\_project/, executar: python manage.py runserver

-Frontend: o prompt deve estar na pasta /sidia\_project/frontend/, onde deve ser usado: yarn start

Se os procedimentos tiveram sucesso, o yarn abrirá no navegador padrão a aplicação cliente, que pode ser acessada em: http://localhost:3000. Os recursos de dados estarão disponíveis em http://localhost:8000.

1. **Extras – Envio de dados BD**

Na pasta /sidia\_project/, há um script desenvolvido para submeter os dados para o banco: load\_data\_file.py.

Como dependência, é necessário o driver de acesso ao PosgreSQL. Ela pode ser obtida por comando no prompt: pip install psycopg2.

As funções foram criadas em separado para permitir submeter os dados às tabelas uma após a outra. Porém, há alguma ordem a seguir. Não deve ser submetido TitleActor e Rating sem os dados de suas chaves, ou seja, as tabelas do modelo Title e Actor. Uma possível ordem seria: Title > Actor > Rating > TitleActor. Na parte inferior do script possui as chamadas organizadas para isso. Devem ser comentadas as linhas correspondentes ao que não será enviado. No início desse documento, é explicado a como enviar as tabelas TitleActor e Rating de forma bem mais rápida por dois passos. As funções correspondentes a isso, possuem os nomes com “temp” no final. Uma instrução SQL, também mancionada naquela seção, deve ser executada diretamente no banco de dados para realizar a segunda parte.

Algumas referências utilizadas:

* https://docs.djangoproject.com/en/
* https://docs.python.org/
* https://pynative.com/python-postgresql-tutorial/
* https://www.postgresqltutorial.com/postgresql-python/
* https://www.django-rest-framework.org/api-guide/generic-views/