Instituto Superior de Engenharia de Lisboa
ENGENHARIA INFORMÁTICA E DE COMPUTADORES

PROGRAMAÇÃO NA INTERNET

TRABALHO PRÁTICO

Autores:

Ricardo Gonçalves, nº 37323

Hugo Barrocas, nº 37330

Hugo Reis,nº38652

Lisboa, 31 de dezembro de 2017

Índice

Índice

Introdução

Foi solicitado a criação de uma aplicação web que deverá ser desenvolvida em trem na que primeira fase possibilitasse a consulta de filmes e o seu cast Atores e directores, e também a consulta de Atores com biografia e filmes em que participaram.
Na segunda fase foi requisitado a adição da funcionalidade de utilizador em que este teria uma área pessoal que permitisse criar listas de filmes.
A terceira fase consiste na adição da funcionalidade de comentários de utilizadores sobre um filme.
Para execução do trabalho foi utilizado a linguagem Javascript suportada em Node.js

Fase 1

Modelo de Dados (DTO)

De acordo com a solicitação implementamos o Modelo de dados ilustrado abaixo.
Onde temos 3 objectos principais

Movie

Objeto que é usado quando são feitas as pesquisas por um nome de um filme e que contém informação simples sobre o filme

MovieDetails

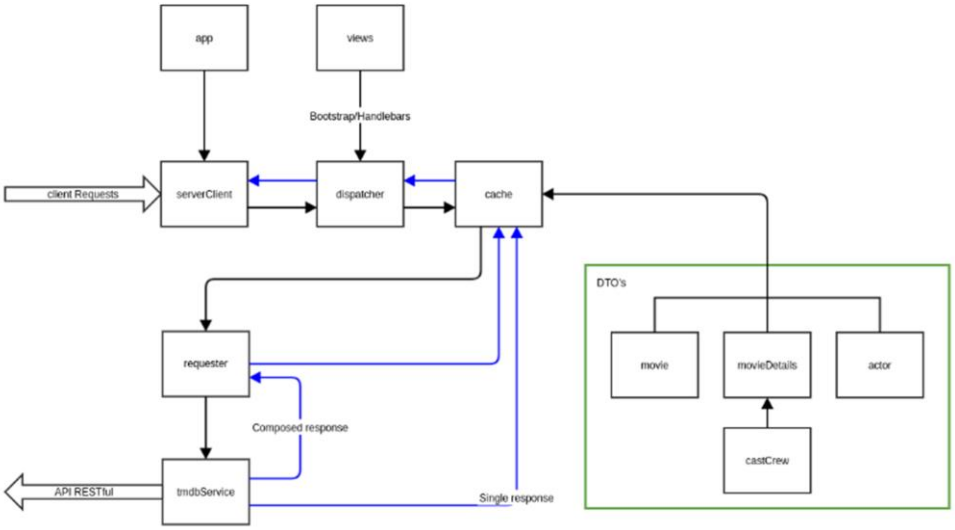
Este objeto contem informação detalhada sobre um filme em concreto e contem também os atores que nele participam e os seus diretores.

Ator

Este objeto é utilizado para guardar a informação detalhada de um ator, a sua biografia e também em que filmes esteve presente

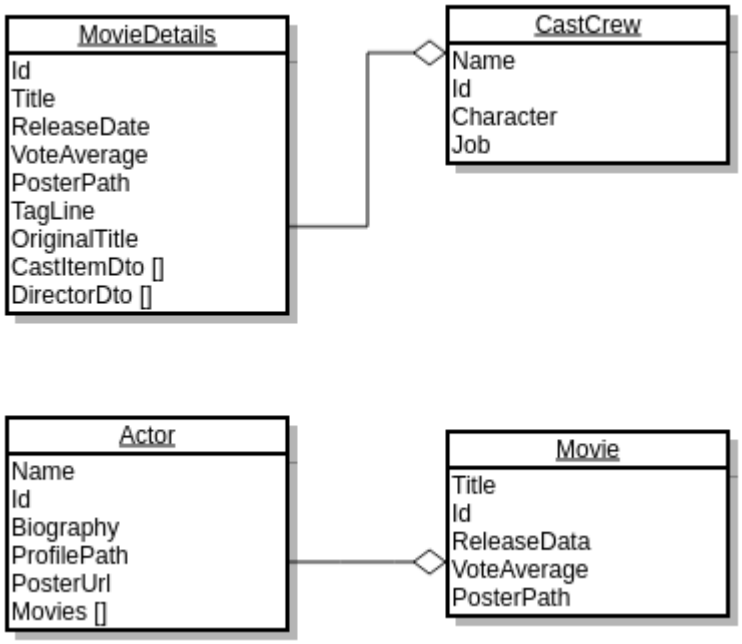
Cast crew

Este objecto é utilizado internamente pelo moviedetails para criar os seus actores e os directores, para poderem ser solicitados pela camada de apresentação.



Descrição funcional

Na nossa aplicação tentamos aplicar o modelo abaixo assinalado, onde tentamos estabelecer um caminho de comunicação por camadas, em que cada camada comunica com um módulo e não existem saltos de comunicação entre módulos de forma a torna-la o mais modular e flexível possível.
Para iniciar a aplicação é necessário correr o comando "Node App.js" que inicia o servidor a escuta na porta 8080
O ponto de entrada para o pedido do cliente é o modulo *serverclient* que recebe valida e trata as url e caso sejam válidas para as nossas pesquisas, inicia a comunicação para o *dispatcher*.
O *dispatcher* têm o papel de efetuar o roteamento do pedido recebido do *serverclient* e dar inicio a execução dos módulos necessários para a construção do mesmo.
No módulo *dispatcher* é também onde são construídas as páginas Html recorrendo as frameworks *handlebars* e *bootstrap*, sendo que nesta fase estão a ser usadas de uma forma simples para o display dos resultados.
O Modulo cache é onde são construídos os nossos "objetos" de acordo com a especificação, para poderem ser armazenados para não serem repetidos os pedidos que já foram feitos anteriormente, seja de filmes , seja de atores.
A cache é também responsável por manter a cache dentro dos limites de forma a não criar problemas de *resource starvation* , neste caso memória, por omissão o valor é de 1000 entradas para atores e filmes.
O modulo *requester* , é responsável e fazer os pedidos a API externa RESTfull *api.themoviedb.org.* quando os objetos não estão presentes na cache.
Para alem dos pedidos a API o *requester* também compõe os pedidos necessários para a composição de objectos internos na nossa aplicação quando os mesmos necessitam de informação de mais que uma fonte.
O modulo *tmdbservice* é quem executa os pedidos ah aplicação externa e compõe os Url's com base nos dados fornecidos pelo requester



Fase 2

Na fase 2 era requisitado uma evolução da fase 1 mais concretamente no método de roteamento dos endpoints disponibilizados ao cliente. Nesta fase ficam disponíveis os seguintes endpoints:

```
/                                Home page
/search?name={query}             Movie search
/movies/{movie id}               Details of a movie
/actors/{actor id}               Details of an actor
/login                           User login page
/users/{username}                User home page containing link for user lists of favorites movies and posted comments
/users/{username}/list           User list of favorite management
/users/{username}/list/{list id} Movies in user favorite list
```

A estrutura de directorias também foi alterada ficando com a seguinte organização:

```
+--+app          Directory containing Javascript Source files
|
+--+views        Directory containing Handlebars template files
|
+--package.json  npm package dependencies
+--server.json   server configurations
+--procfile      heroku process start file
```

Routing

Para efectuar o routing foi usado o modulo *express* que permite associar uma função a um URI (caminho) de uma forma simples e concisa tal como *express.get('/',homePage)*. Este exemplo efectua uma chamada à função *homePage* para todos os pedidos http com o caminho '/' e método *GET* efectuados pelo cliente. O modulo express também permite associar um router a um caminho assim é possível definir diferentes rotas para o mesmo caminho base

O modulo express permite chamadas a funções encadeadas, ou seja uma função associada a um caminho pode delegar o controlo à próxima função na cadeia de endpoints efectuando uma chamada à função *next()* que é dada como parâmetro.
O cliente pode requisitar dados que não estão disponíveis localmente e neste caso a aplicação tem de aceder a uma API externa, no fim da cadeia de endpoints foi colocado o *midleware cache* desenvolvido da primeira fase que efectua os pedidos ao exterior e na resposta devolve os dados a função que efectuou o pedido.
A devolução dos dados é efectuada usando o padrão *decorator* que consiste em adicionar uma função ou redireccionar uma existente de um dos objectos dados como parâmetros.

```
app.use('/', function(req, resp, next){
    let ori_send = resp.send
    resp.send = function renderView(...args){
        resp.send = ori_send
        resp.end()
        return
    }
   next()
})
```

Neste exemplo a função *resp.send* é substituída pela função *renderView* e o controlo é passado à próxima função com a chamada *next()*. No final da cadeia a função que detém os dados efectua a chamada *resp.send(responsedata)* que corresponde à chamada *renderView(responsedata)*, dentro desta função é restaurada a função *send* original para devolução dos dados ao cliente.

Gestão de utilizadores

Para gestão de utilizadores e área pessoal de cada utilizador, mais uma vez o modulo *express* permite o registo de outros módulos para essa funcionalidade tais como o módulos *passport*, *express-session* e *body-parser*.

Com estes módulos é possível autenticar um utilizador e manter a sessão sem recurso ao uso de cookies.
O processo de autenticação é efectuado pelo MidlewareLogin que exporta um router para o endpoint "/login" que suporta os métodos GET e POST.

Quando o utilizador submete um pedido de autenticação o midleware faz primeiro uma validação para novo utilizador e se for o caso é criado um utilizador com o username e password fornecidos (valores disponibilizados como propriedades no objecto *req* devido ao modulo *body-parser*) e guarda na base de dados de utilizadores, após este processo o utilizador encontra-se automaticamente autenticado e é redireccionado para a sua área pessoal.

Para o caso de ser um utilizador existente, é obtido um objecto utilizador da base de dados e validada a password entrada com a existente na base de dados em caso de sucesso é criada uma sessão com a chamada à função *logIn(user, callback)*, na função de callback é feito o redireccionamento para a pagina pessoal.
Para a aplicação distinguir vários utilizadores autenticados, o modulo passport necessita de duas funções para efectuar a serialização/desserialização de um utilizador.

Após este processo fica disponível um objecto *"user"* no parametro *"req"* de cada pedido efectuado pelo cliente, a validação de autenticação é efectuada pela chamada à função *"req.iSAutenthicated()"* que retorna *true* em caso de sucesso.

Para guardar os dados de utilizadores foi escolhida a base de dados NoSQL couchDB que permite criar/aceder a objectos JSON, ou seja quando é registado um novo utilizador é guardado um objecto JSON na base de dados com o mesmo formato que um objecto utilizador.

Uma vez autenticado o utilizador é redireccionado para a sua homepage '/users/{username}' e enquanto a sessão estiver activa será redireccionado no caso de fazer um pedido a '/login'. Nesta pagina é apresentado um link para efectuar *logout*, um para ver as listas pessoais e um para comentários que o utilizador escreveu (3ª fase).

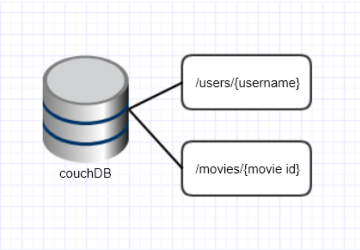
Listas de utilizador

Para implementação da funcionalidade de listas de pessoais cada objecto *user* contem um array de objectos, em que cada objecto contem duas propriedades (id, nome) e um array para os id's dos filmes dessa lista.
De modo a disponibilizar estas listas ao utilizador é criado um *cookie* no *browser* durante o processo de autenticação que contem o objecto *user* e consequentemente as suas listas.

Estas listas encontram-se disponíveis na página de detalhes de um filme e só se o utilizador se encontrar autenticado. Esta funcionalidade é conseguida devido ao servidor validar a autenticação e na resposta permitir que o botão que alterna a visibilidade das listas fique activado.
Assim que o utilizador pressione o botão de listas é chamada uma função que alterna a visibilidade de um quadro que contem os nomes das listas do utilizador.

O processo de adicionar um filme a uma lista também é efectuado no lado do cliente efectuando pedidos http com o método "PUT" com recurso a API *XMLHttpRequest*, esta API tem a vantagem de efectuar pedidos ao servidor sem que a página actual sofra uma actualização total. Com este método é possível efectuar operações CRUDE sobre uma lista.

Como referido é utilizada uma base de dados NoSQL e para esta fase definiu-se o seguinte modelo :



Dentro da base de dados foram criados directórios separadas para utilizadores e filmes, desta forma é possível um acesso aos documentos fazendo pedidos directos com o caminho do documento dentro da base de dados.

Objecto "user"

```
user {
    "name": "user name",
    "password" : "secret pass",
    "lists" : [
        {
            "id" : "array index",
            "name" : "listname",
            "movies" : []
        }
    ]
}
```

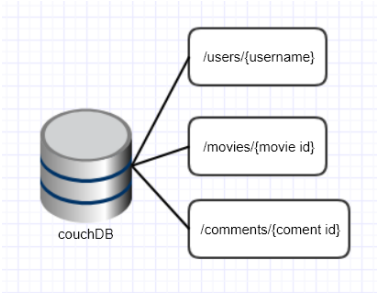
Objecto "movie"

```
movie{
    "id" : "movie id",
    "name" : "movie Title"
}
```

Fase 3

A terceira fase consiste na adição da funcionalidade que permite que os utilizadores efectuem comentários a um filme ou respondam a comentários realizado. O carregamento dos comentários de um filme deve de ser dinâmico ou seja não deve de ser efectuada qualquer actualização completa da página.
Um outro requisito é o da visualização de todos os comentários efectuados por um utilizador na sua página pessoal em que cada comentário contem um link para a pagina do filme em que foi colocado.
Por fim é requisitada a funcionalidade de paginação sobre as listas de filmes de um utilizador.

Modelo BD



Sendo esta fase a adição de funcionalidades à fase anterior a implementação começa por modificar o modelo de dados com a adição de mais uma base de dados para os comentários e a extensão do objecto *movie* com a adição de um array que contem o id de cada comentário a esse filme.
Da mesma forma um objecto *comment* também contem um array com id's de comentários que se destinam ao próprio comentário.

Do lado da a aplicação foram adicionados dois endpoints para gestão dos comentários.

```
/users/{username}/comments          Show all comments posted by user
/movies/{movie id}/comments         GET/POST comments for a movie
```

```
user {
    "name": "user name",
    "password" : "secret pass",
    "lists" : [
        {
            "id" : "array index",
            "name" : "listname",
            "movies" : []
        }
    ]
}
```

```
movie{
    "id" : "movie id",
    "title" : "movie Title",
    "comments" : []
}
```

```
comment{~
    "id" : "auto generated",
    "movie" : "movie id",
    "user" : "username",
    "text" : "user comment",
    "comments" : []
}
```

Comentários a filmes

Com este modelo podemos então apresentar os últimos comentários efectuados a um filme na sua página de detalhes.

Após carregamento da página no cliente é executado um script nessa página que efectua um pedido *http* ao servidor com url da página actual com o uri *"/comments"* e método "GET" usando a API *XMLHttpRequest* para obter os últimos comentários efectuados.

O servidor encaminha este pedido para o midleware correspondente, e este começa por extrair o identificador do filme e efectuar um pedido à base de dados.
A base de dados por sua vez responde devolvendo um objecto *movie* que contem um array com os identificadores de todos os comentários efectuados a esse filme.
De seguida o servidor efectua um pedido a base de dados para cada identificador e esta responde novamente devolvendo um objecto *comment* para cada pedido, a medida que as respostas são recebidas são adicionadas num array. Assim que todos os comentários estiverem recebidos o servidor devolve esse array para o cliente.

O cliente assim que recebe o array com as respostas adiciona os elementos necessários para cada comentário ao DOM, desta forma os comentários surgem na página sem que esta sofra uma actualização.

Para a colocação de um comentário o processo é idêntico ao anterior mas neste caso o utilizador tem de estar autenticado e é efectuado um pedido http mas desta vez usando o método "POST".
O servidor para inserir o comentário necessita dos dados do filme e do utilizador que são enviados no corpo da mensagem no formato JSON que foi especificado no cabeçalho do pedido "Content-Type : application/json".

Após análise do corpo da mensagem é obtido um objecto com o formato

```
{
    "movie_id" : "...",
    "movie_title" : "...",
    "movie_comment" : "{...}
}
```

Com e estes dados o servidor faz um pedido à base de dados para criação “POST” de um comentário, ao qual responde com um objecto que contem um id único.
De seguida é feito um novo pedido à base de dado para obtenção de um filme com o id de filme fornecido, caso não exista a base de dados cria um usando os dados obtidos do corpo da mensagem http.
O passo final consiste em adicionar o id do comentário à lista de comentários do filme e actualizar o filme na base de dados.
Para consolidar a operação o servidor devolve o objecto *comment* que foi criado ao cliente e efectua a manipulação do DOM de modo a mostrar o comentário recém colocado no topo da lista.

Comentários de utilizador

Para os comentários de utilizador foi aplicada uma solução idêntica à de obtenção dos primeiros comentários na pagina de detalhes de um filme. Neste caso como é usada uma base de dados NoSql não existe a possibilidade de efectuar interrogações de modo a obter todos os comentários de um utilizador em particular. A solução para este problema consiste em criar uma vista dentro da base de dados que aplica uma função de filtro para utilizador a cada comentário.

```
function(doc) {
    emit(doc.username, doc);
}
```

Com esta função definida para a vista na base de dados é possível obter os dados de um utilizador efectuando um pedido http ao url

```
/comments/_design/filters/_view/_user?key={username}
```

Paginação

A paginação criada para as listas pessoais de utilizador usa o conceito de api *REST* ou seja quando o utilizador selecciona a próxima pagina e efectuado um pedido ao servidor para a pagina de listas do utilizador que se encontra autenticado mas é adicionado ao caminho o recurso “/page/{num}”.

A resposta deste pedido gera uma actualização da página inteira, pela qual o servidor envia os links para a página anterior e pagina seguinte.

Optimizações e problemas conhecidos

1 – Melhoramento do tratamento de excepções
2 – Optimização dos módulos “requester” e “tmdbService” usando conhecimentos adquiridos.
3 – Implementação opcional das funcionalidades de modificação de listas pessoais.
4 – Implementação de comentários sobre comentários, visualização de comentários de utilizador
5 – Modificação para implementação dinâmica para gestão de listas
6 – Parar apresentação dinâmica dos comentários quando já não existem mais
7 – Testes unitários

Funcionalidades extra

A aplicação COIMA dispõe de uma linha de comandos com os seguintes comandos

```
COIMA > help

help    this message
quit    quit application
initdb   initialize couchdb
COIMA >
```