UNIVERSIDADE D
COIMBRA

Bases de Dados

Relatório final de projeto

[iDEI: plataforma de *streaming* de áudio]

2022-2023

João Pinheiro 2017270907

Joel Oliveira 2021215037

Johnny Fernandes 2021190668

Índice

INTRODUÇÃO **3**

DIAGRAMA ER **4**

Modelo conceptual 4
Modelo físico 6

API **6**

PLANO DE DESENVOLVIMENTO **7**

CONCLUSÃO **8**

MANUAL DE INSTALAÇÃO **9**

Instalação do Python 9
Instalação do DBMS (PostgreSQL) 11
Instalação do Postman 13
Instalação do projeto no DBMS 13

MANUAL DE UTILIZAÇÃO **14**

Colocação de token [Transversal] 14
Registo de novos utilizadores 15
Login de utilizador 15
Adicionar uma música 15
Adicionar um álbum 16
Pesquisar uma música 16
Pesquisar um artista 16
Subscrever ao Premium 16
Criar uma playlist 17
Reproduzir uma música 17
Gerar um cartão pré-pago 17
Comentar uma música ou responder a comentário 18
Criar um relatório mensal 18

Índice de figuras do relatório (figuras dos apêndices não incluídas - acessórias)

Figura 1 – Diagrama ER (conceptual) 4
Figura 2 – Diagrama ER (físico) 5
Figura 3 - Tabela de tarefas 8

Introdução

O presente relatório incide sobre a versão final do projeto de Bases de Dados, o qual envolve o processo de design e conceção de um modelo entidade-relação que permita originar uma estrutura de base de dados adequada ao tratamento de informações de uma plataforma de *streaming* de áudio – a qual se apelidou de iDEI. Para além disso, o projeto envolve também o desenvolvimento dos *endpoints* da API com o qual a plataforma deverá interagir para implementar as mais diversas funcionalidades descritas em maior profundidade nas próximas secções deste relatório.

Para o efeito, a base de dados foi desenvolvida no DBMS PostgreSQL, com recurso ao ONDA (onda.dei.uc.pt) para o design e criação do modelo ER. Por outra via, no que diz respeito à API, utilizou-se um *framework* de *webserver* em *Python* – o *Flask* – para a criação dos nossos *endpoints*, cuja instalação, configuração e utilização estarão figurados em apêndice ao presente relatório. De ressalvar que no desenvolvimento do projeto, não incidindo diretamente numa interface web, se fez uso da implementação da API através do *Postman* para a submissão das diferentes *requests* necessárias.

Com o relatório pretendemos assim, seguindo as *guidelines* definidas no enunciado fornecido, demonstrar e explicar as diferentes fases do processo bem como a tomada de decisão e ponderação essencial aos bons métodos de aplicação dos conhecimentos adquiridos na disciplina durante o semestre tentando sempre com máximo rigor aplicar soluções baseadas em SQL (*queries* e *triggers*).

Ainda, anexo ao relatório estará presente o conjunto de ficheiros relativos ao código fonte da API bem como os ficheiros SQL necessários à criação e inicialização da base de dados.

Em suma, o desenvolvimento deste projeto envolveu não só competências técnicas de design e programação, mas também de análise e pensamento crítico, com vista a garantir uma estrutura de base de dados eficiente e uma API funcional e intuitiva.

Diagrama ER

Modelo conceptual

Neste projeto foi desenvolvido um modelo Entidade-Relação (ER) que permitiu representar de forma clara e concisa as entidades e as relações presentes na plataforma iDEI. A partir deste modelo ER, foi possível derivar um modelo de dados relacional que possibilitou a implementação da base de dados propriamente dita.

O modelo ER desenvolvido para a plataforma iDEI incluiu as seguintes entidades: *usr*, *consumer*, *admin*, *artist*, *label*, *album*, *song*, playlist, *comment*, *card*, *purchase*, *subscription* e log; bem como relações entre estas entidades, tais como a *relação* de "ser" entre [*usr* e *consumer*|*artist*|*admin*], a relação de "pertencer" entre [*song* e *album*] ou [*song* e playlist], entre outras descritas abaixo na figura 1. Este modelo permitiu uma representação visual intuitiva e organizada das entidades e relações da plataforma iDEI.

A partir do modelo ER, foi possível derivar um modelo de dados relacional que utilizou tabelas para representar as diferentes entidades e relações da plataforma. Por exemplo, a entidade *usr* foi representada numa tabela com campos como id, *username*, password, email, *name* e *address*. Já a relação entre *album* e *song* foi representada numa tabela intermédia que liga ambas as tabelas. Este modelo relacional permitiu uma representação lógica da base de dados e a sua implementação em *PostgreSQL*.

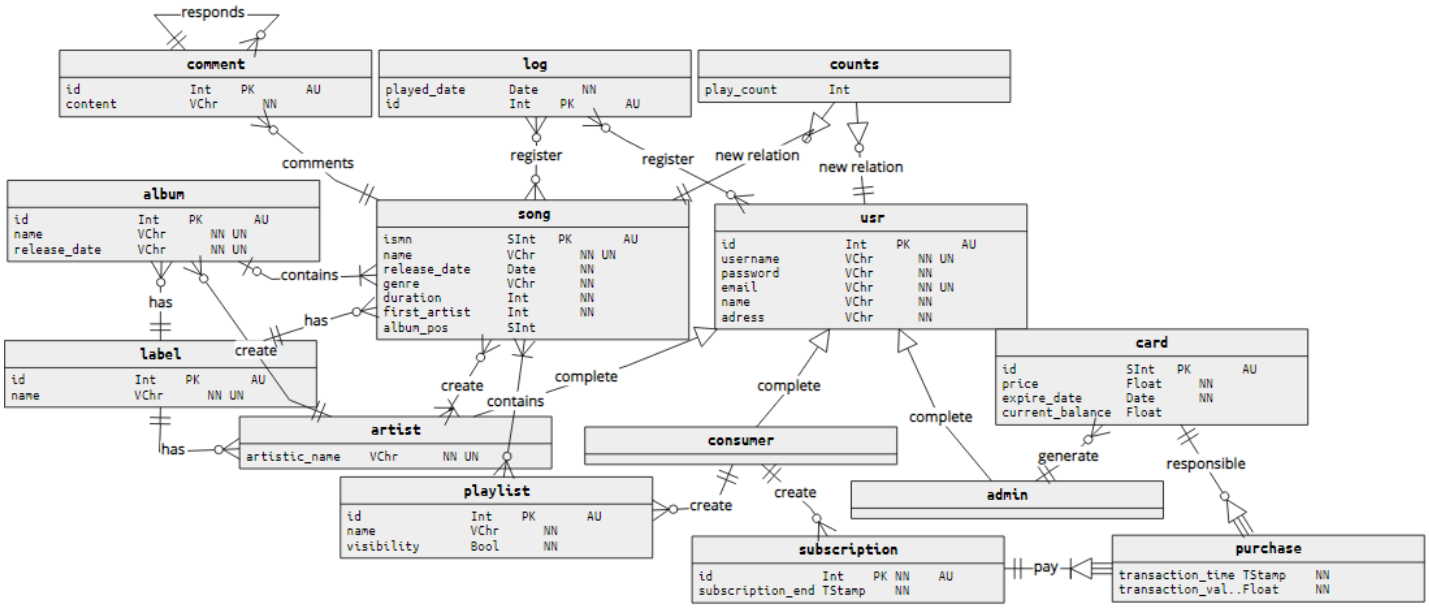


Figura 1 – Diagrama ER (conceptual)

Do ponto de vista do processo de design foi essencial compreender as diferentes necessidades da plataforma em detalhe, para que se pudesse criar uma relação consistente entre as diferentes entidades garantindo três aspetos fundamentais e que foram tidos como base de decisão: a não redundância dos dados, a garantia de armazenamento de todas as informações relevantes e a segurança e integridade dos mesmos.

Para garantir a aplicação destes motivadores de decisão – e em concordância com o que é possível observar da figura 1 - foi necessário criar um relacionamento que de certa forma estivesse suficientemente encadeado entre as diferentes entidades por forma a se criarem tabelas únicas sem repetição de parâmetros. Com isso garante-se também indiretamente um relacionamento mais intrínseco entre essas mesmas entidades, uma vez que (por exemplo) para se extrair informações sobre um determinado artista, é necessário verificar várias informações que estão em outras tabelas, nomeadamente os dados pessoais na tabela de utilizador, as músicas a que está associado na tabela *song* (feito com recurso a uma tabela intermédia na figura 2 abaixo), a editora que o representa na tabela *label* ou ainda os álbuns que produziu, na tabela *album*. Ainda na mesma linha de pensamento, parte da segurança é obtida através precisamente dessa relação, uma vez que a remoção de dados poderá não ser possível através da dependência de outras tabelas existentes.

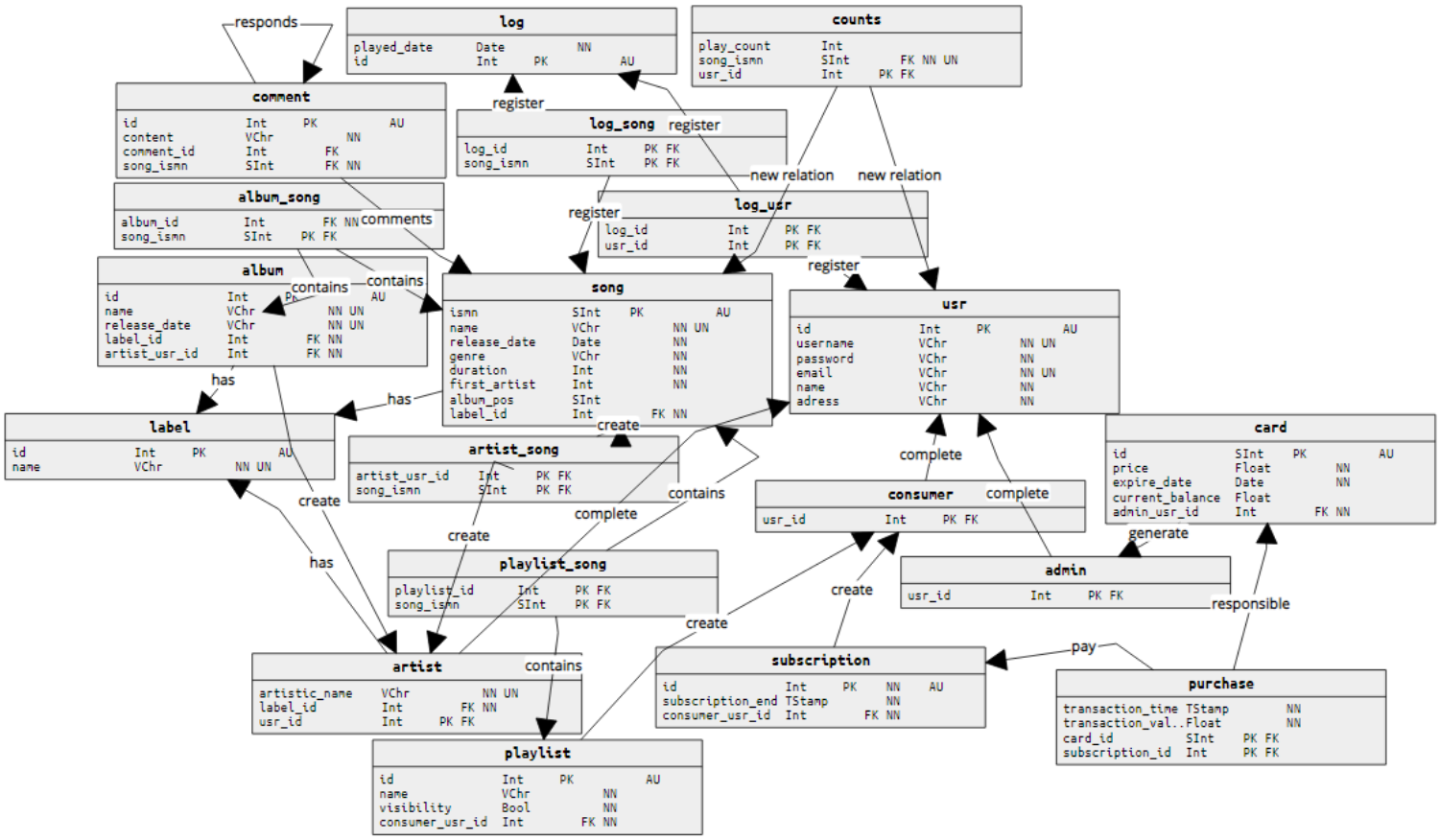


Figura 2 – Diagrama ER (físico)

Modelo físico

Conforme se pode observar pela figura 2, há várias tabelas de apoio que são criadas para o encadeamento entre as entidades, conforme referido, nomeadamente a tabela *log_song* e *log_usr* que se relacionam com a tabela log e com as respetivas tabelas *song* e *usr*. Tal acontece uma vez que a tabela log, a ser uma entidade fraca, acabaria por usar uma chave primária como conjunto de duas chaves estrangeiras que dizem respeito a cada uma das tabelas e surgiria como um problema na medida em que para a mesma música e para o mesmo utilizador, não é possível ter mais do que um registo de música executada, dificultando uma extração de informação mais granular que diz respeito aos gostos musicais de um determinado utilizador para diferentes períodos. Dessa forma, em face à solução apresentada na meta I, decidimos abandonar a ideia de deixar a tabela log como entidade fraca em relação às outras duas. Além disso, temos ainda as tabelas *álbum_song*, *playlist_song* e *artist_song*, que relacionam a tabela música com cada uma das outras, conforme referido em exemplo anterior. Foi criada também uma tabela counts para a contagem do número de reproduções de músicas para a tabela de top 10, que será o caso de uma playlist.

Além destas tabelas adicionais, no modelo físico observam-se também parâmetros adicionais colocados nas diferentes tabelas, fruto das relações com outras tabelas e para as quais é necessário dar indicação das chaves estrangeiras (FK) de entradas em outras tabelas, para que a relação possa permanecer. Para além do mais, o tipo de dados armazenado foi tanto quanto possível ajustado aos atributos em questão.

De uma forma geral, em relação ao apresentado na meta I de entrega, não houve alterações significativas na estrutura do nosso diagrama ER, uma vez que os três princípios fundamentais foram logo considerados desde início.

API

Para o processo de criação da API foi necessário recorrer a uma *framework* web para *Python* – o *Flask*. Por outro lado – e por que o projeto não compreendia uma interface web – foi necessário utilizar o *Postman* para o teste da API, onde se criou uma coleção a ser anexa ao presente relatório, com as diferentes *requests*.

Para a implementação da API tivemos por base o repositório do professor regente da disciplina – João Campos (https://github.com/jrcampos/bd-python-demo-api) - sobre o qual desenvolvemos os diferentes *endpoints* da API. Para se garantir algumas funcionalidades associadas ao próprio código, seja diretamente relacionado com a API, com as questões de segurança ou com as ferramentas de *debug*, fez-se a inclusão de algumas bibliotecas externas (outras estão incluídas mas também foram mencionadas) ao *Python*, nomeadamente.

- *Flask* – Servidor web para a criação dos *URLs*/*endpoints* associados à API
- *Logging* – Para fazer o log das diferentes interações entre o *Postman* e a API
- *Traceback* – Para fazer *debug* e obter mais informações sobre os pontos em que são originados erros na execução da API
- Time e *Datetime* – Para o cálculo dos períodos de subscrição, baseado em datas reais

- Os e *Dotenv* – Para a utilização de ficheiros com variáveis de ambiente, sobre os quais se consegue manter e garantir um grau mais elevado de segurança sobre as credenciais de acesso à base de dados, nomeadamente através de um mais alto nível de restrições nas permissões de acesso ao ficheiro, por utilizadores específicos associados à execução da própria API.
- Psycopg2 – Para a interação entre a base de dados e a nossa aplicação.
- JSON – Para a API, uma vez que o *Postman* interage através de JSON
- JWT – Para a geração de *tokens* únicas de utilizador que serão usadas e transmitidas em detrimento de palavras-passe. Feita a autenticação o utilizador tem acesso a uma *token* com período de validade igual a um dia.
- *Bcrypt* – Para a encriptação das palavras-passe na base de dados.

Estas importações não serão incluídas em anexo devendo ser importadas através de um conjunto de passos a seguir indicados no manual de instalação.

O desenvolvimento geral da aplicação centra-se então no uso de funções associadas a rotas de *Flask* que serão executadas aquando do recebimento de uma *request* com o método POST, PUT ou GET por parte do *Postman*. As funcionalidades estão condensadas dentro de um *try-catch* que apesar de em casos normais não ser usado como controlo de fluxo neste caso em particular de *acesso* a uma base de dados achamos pertinente uma vez que, para casos incógnitos ou inesperados de payloads passados pelo *Postman* - e também como forma de dar um *return* coeso de resultados – achamos uma boa escolha.

O acesso à base de dados é feito através do psycopg2 acedendo a um ficheiro .env que contém as nossas credenciais bem como ip e porta de ligação à base de dados. A escolha de uso de um arquivo .env prende-se com a maior garantia de segurança uma vez que a palavra passe não se encontra encriptada. Além do mais, para as palavras-passe guardadas na base de dados, as mesmas são codificadas com recurso ao *Bcrypt* antes de serem guardadas.

Para informações mais aprofundadas sobre o funcionamento da API bem como dos *endpoints* criados recomendamos a leitura do manual de utilização que se faz acompanhar no fim do relatório.

Plano de desenvolvimento

O projeto teve desde a sua génese a divisão de tarefas pouco delineada havendo uma colaboração de todos os elementos do grupo em todas as distintas fases de design, conceção e teste da aplicação. Desta forma e, de acordo com o pedido no enunciado sobre uma descrição mais detalhada sobre as diferentes tarefas executadas pelos diferentes membros, já numa fase mais avançada do projeto separaram-se as distintas tarefas ainda que houvesse sobreposição dos diferentes elementos nas diferentes tarefas, seja por motivos de reforço das ideias implementadas no design, bem como melhoria das soluções já implementadas ou por mera dúvida em relação à abordagem correta, motivos pelos quais os restantes elementos sempre colaboraram para levar o projeto a bom porto. Assim, divide-se abaixo em tabela as diferentes tarefas executadas, sendo necessário considerar durante a análise desta secção que todos os elementos acabariam, de uma forma ou de outra, por estar incluídos nos diferentes processos de desenvolvimento.

	João	Joel	Johnny
Design do modelo Entidade-Relação bem como correções			
Criação inicial dos diferentes *endpoints* da API			
Limpeza, otimização e simplificação dos *endpoints* da API			
Melhorar o equilíbrio *Python* vs *queries* SQL nos *endpoints* para recorrer mais a SQL			
Criação das *triggers* que dizem respeito ao TOP10			
Verificação de aspetos de segurança, integridade e concorrência			
Criação do relatório final, bem como dos manuais			
Testes de funcionalidade, verificação do preenchimento correto das tabelas			
Criação e correção/melhoria do ficheiro SQL de criação da base de dados			
Criação de um ficheiro SQL de importação com *dummy* data para inicializar a BD			
Preparação para entrega e submissão final			

Figura 3 - Tabela de tarefas

Apesar de o grupo de trabalho não ter contabilizado as horas na implementação das diferentes funcionalidades, a divisão de tarefas foi equitativamente definida havendo um contacto constante durante as diferentes fases e também em aula e havendo uma distribuição contínua de novas fases de trabalho consoante os avanços e conclusões das etapas. Além do mais, a experiência prévia à disciplina de um dos elementos do grupo permitiu uma melhor noção da dimensão, carga e responsabilidade de cada tarefa, havendo particular atenção à distribuição de cada parte para não haver sobrecarga de trabalho.

De uma forma geral e em forma de heteroavaliação, apurou-se um bom resultado na sua globalidade com a boa participação de todos os elementos do grupo.

Conclusão

Para o proposto neste projeto foi feita a implementação de uma API com acesso a uma base de dados para a gestão de informação de uma plataforma de *streaming* de áudio, conforme já indicado. A estrutura desenvolvida foi engenhosamente ponderada e feitos os testes finais verificamos que o sistema tem as suas funcionalidades aplicadas de uma forma correta. Como exemplo, um utilizador não registado (não tenha uma *token* associada) apenas poderá criar uma conta e não tem acesso a qualquer outra funcionalidade, por outro lado apenas se for um artista poderá criar um álbum e inserir músicas, bem como um utilizador final só poderá ser um cliente se tiver associados à sua conta cartões com valores válidos. Toda esta relação e entrelaçamento das diferentes tabelas garante em primeira instância a unicidade das informações, a segurança e a integridade dos dados.

Considerou-se por fim que há pouca margem para melhoria no presente modelo ER, havendo sempre a possibilidade de melhorar um pouco mais na questão das *queries* SQL usadas na API por forma a reduzir gradativamente o recurso ao *Python* como ferramenta para o controlo de fluxo.

Manual de instalação

Para o desenvolvimento do projeto é necessário um conjunto de requisitos essenciais ao sistema de desenvolvimento e produção. Como tal, foi usado um servidor web usando a *framework* web para *Python* – *Flask* com alguns requisitos obrigatórios indicados abaixo.

O sistema de DBMS escolhido foi o *PostgreSQL*, cuja instalação também estará incluída no manual.

Para a utilização da API e porque não existe uma interface web desenvolvida no âmbito do projeto, será utilizado o *Postman*, cuja instalação também estará incluída no manual.

Todo este manual refere-se à versão de Windows 11 e terá aspetos similares em *MacOS*. Para a instalação em sistemas Linux, por favor confirme as *guidelines* de instalação oficiais disponíveis nas páginas do *Python*, *PostgreSQL* e *Postman*.

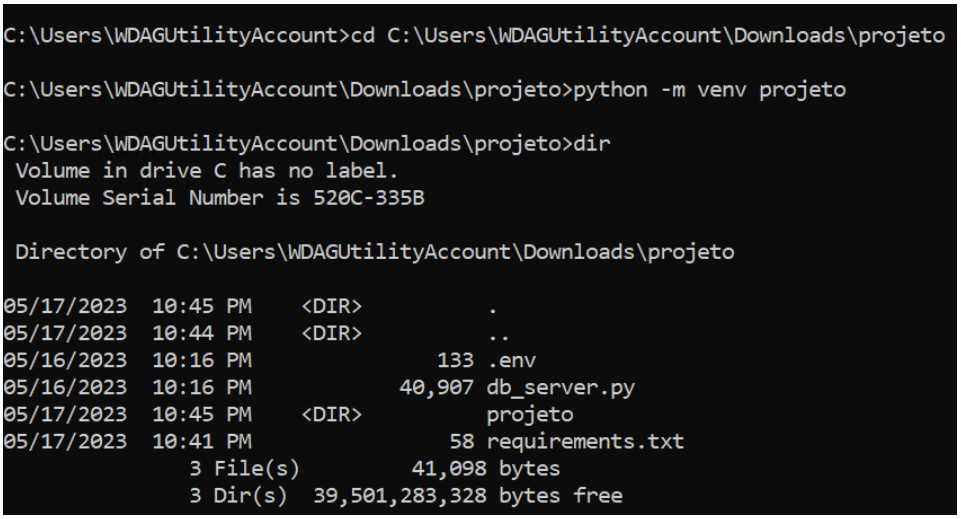
Instalação do Python

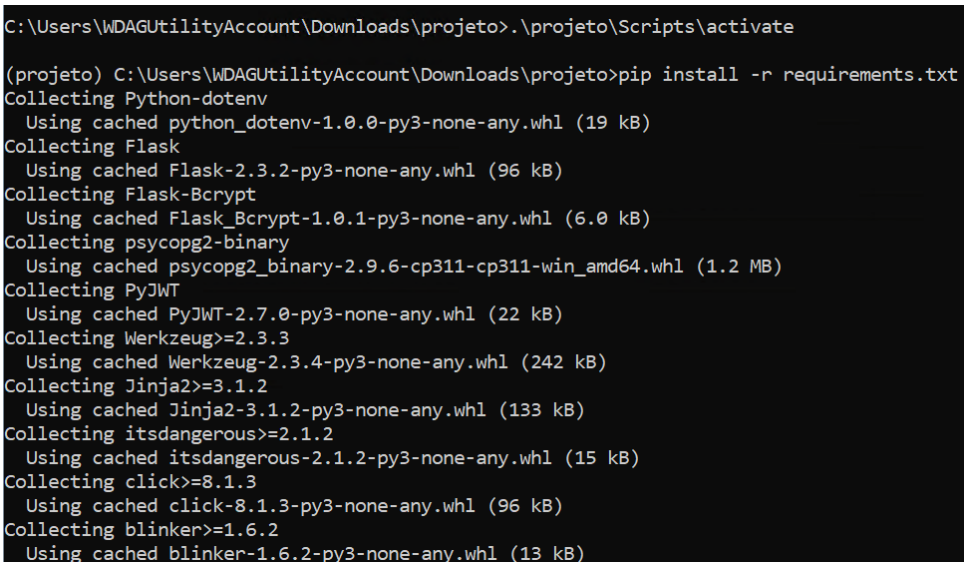
1. Aceda ao seguinte endereço: https://www.python.org/downloads/ e faça download da última versão do *Python*.
2. Na seguinte janela, não se esqueça de selecionar a opção '*Add python.exe to PATH*' para associar o *Python* como uma aplicação de sistema.

3. Clique '*Install Now*
4. Após a instalação, aceda ao menu iniciar e abra uma linha de comandos (pesquise por *cmd*)
5. Aceda à pasta do projeto através de cd `C:\Users\<username>\Downloads\Projeto` onde <username> deverá ser o nome de utilizador da conta Windows iniciada.
6. Em *Python*, verifique a existência do PIP através de `pip --version` devendo retornar a versão do *pip* bem como o PATH de instalação.
7. Proceda à instalação dos requisitos necessários num ambiente separado:
    a. Para a criação de um ambiente separado, vamos usar o *virtualenv*, incluído na versão base do *Python*, com o comando `python -m venv projeto` para a criação de um ambiente de nome projeto.

b. Ative o ambiente usando `.\projeto\Scripts\activate` (verá na consola que (projeto) aparecerá como prefixo)
c. Instale os requisitos com o seguinte comando `pip install -r requirements.txt`

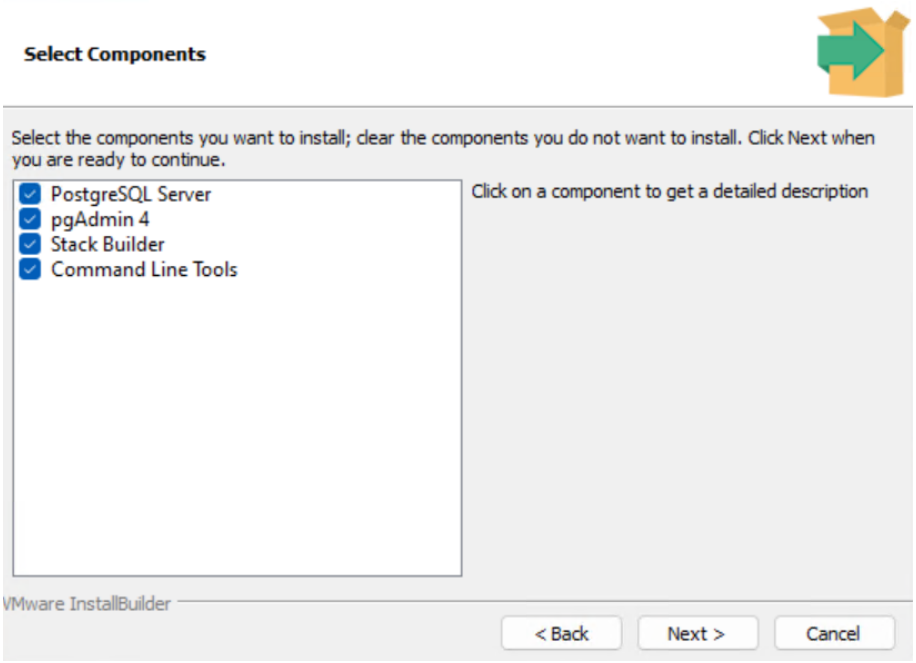
8. A instalação base do *Python* está feita, bem como a sua configuração base. Recorde-se que no IDE, caso utilize um, deverá configurar o interpretador de *Python* com o ambiente agora criado. Em alternativa, a execução do servidor web será feito com o comando `python db_server.py`



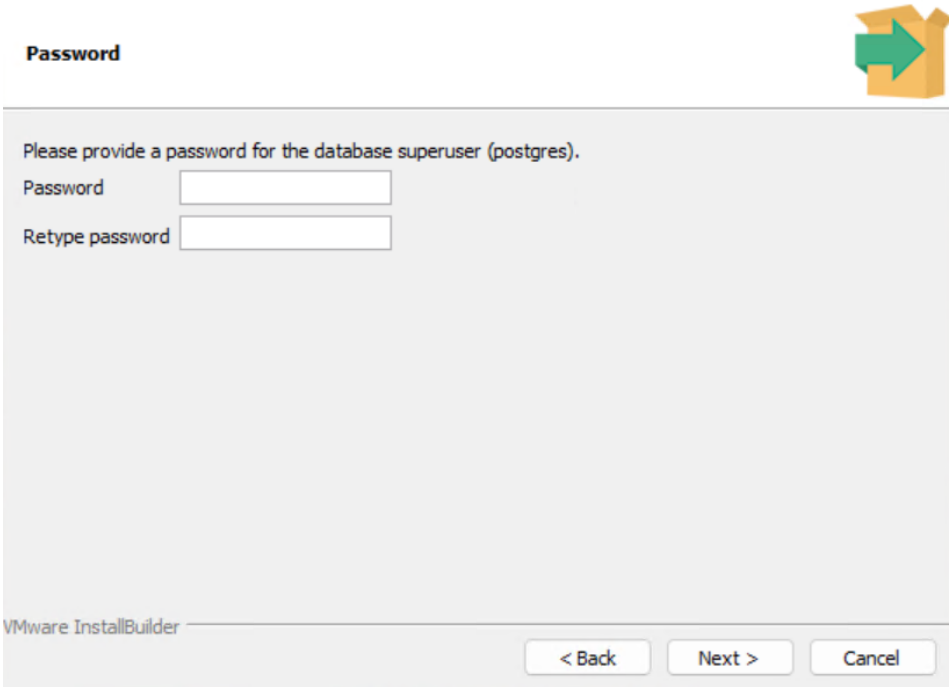


Instalação do DBMS (PostgreSQL)

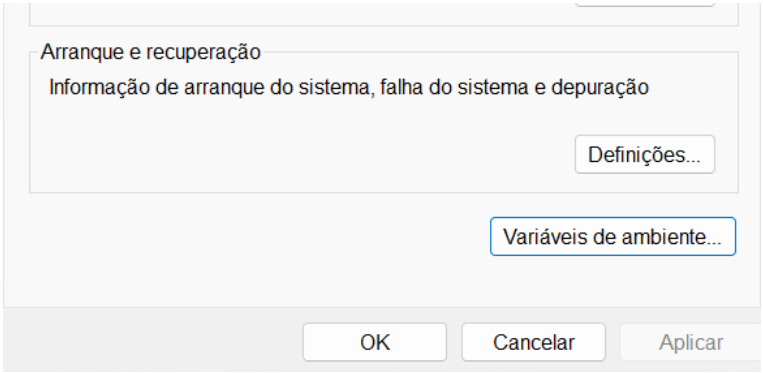
1. Aceda ao seguinte endereço: https://www.postgresql.org/download e faça download da versão mais adequada ao seu sistema. Como este manual se refere à versão de Windows 11, dirija-se ao seguinte endereço: https://www.postgresql.org/download/windows/ e faça download da última versão (versão 15.3 do *Postgres* à data da redação deste manual).
2. De acordo com a imagem seguinte, recomendamos que deixe todas as opções instaladas.

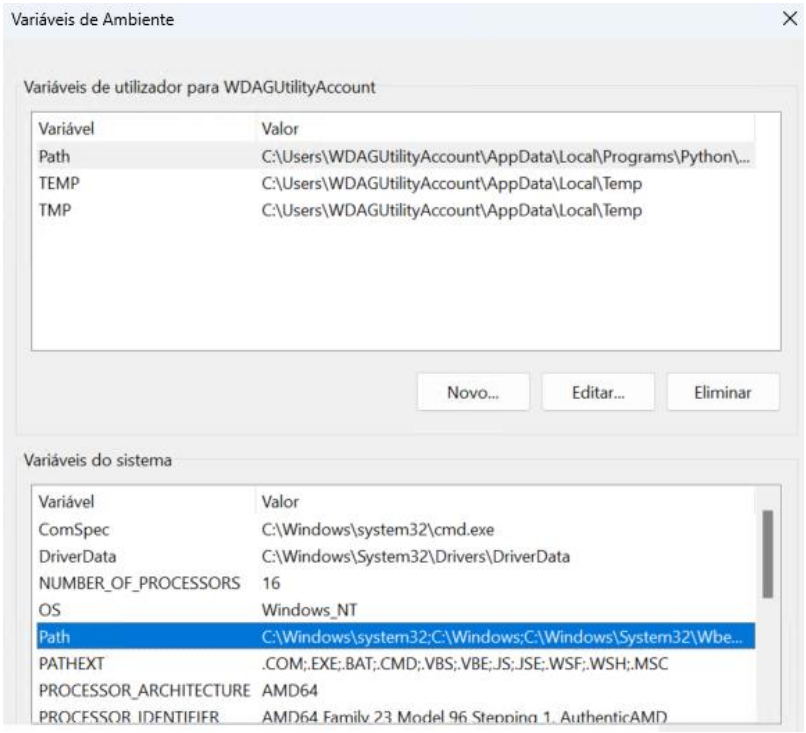


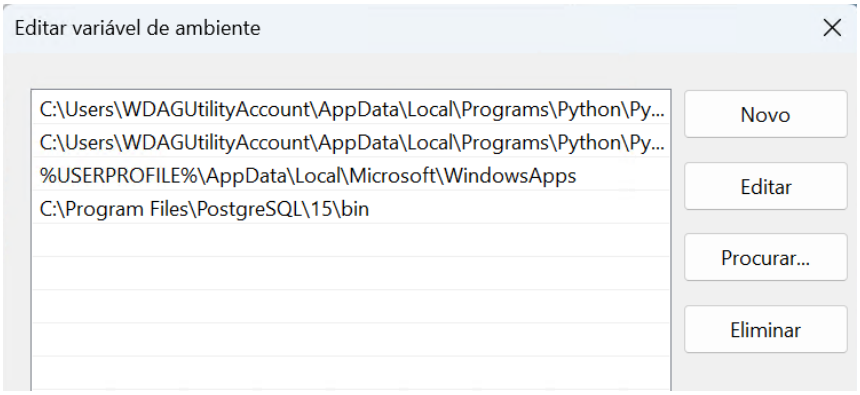
3. Defina uma password obrigatória conforme abaixo. Recomendamos, para efeito deste tutorial e de fácil memorização, que insira `postgres` como a senha. Em alternativa poderá colocar `admin`. Recomenda-se que em servidores de produção todas as senhas `default` sejam de uma complexidade superior a 16 caracteres.



4. Deixe a porta *default* com o valor `5432`
5. No passo seguinte, deixe a opção como `[Default locale]`
6. O restante da instalação deverá ser automatizado
7. Recomendamos que adicione o *psql* (Command Line tools do postgres) à path, caso ainda não tenham sido inseridas. Para isso, aceda às suas variáveis de ambiente indo a iniciar e pesquisando por `Variáveis de ambiente`. Insira os parâmetros conforme abaixo, com PATH `C:\Program Files\PostgreSQL\15\bin`, de acordo com a sua versão (botão Novo para adicionar nova entrada).

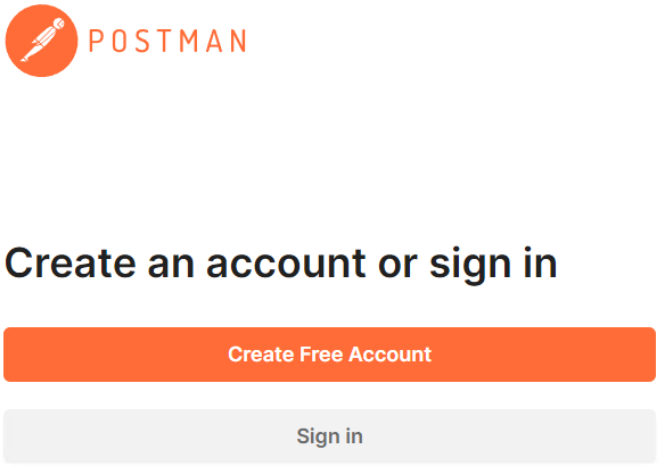






Instalação do Postman

1. Aceda ao seguinte endereço: https://www.postman.com/downloads/ e faça download da versão adequada ao seu sistema.
2. Após o download, instale o *Postman*. Por este aplicativo ter um instalador totalmente automatizado não é configurar nenhuma opção (nem tampouco é possível) durante o processo de instalação.
3. Após a instalação, abra o *Postman*, onde verá que necessita de iniciar sessão. Para tal, se ainda não tiver uma conta de utilizador, por favor crie uma no website através da ligação a laranja *Create Free Acount*.

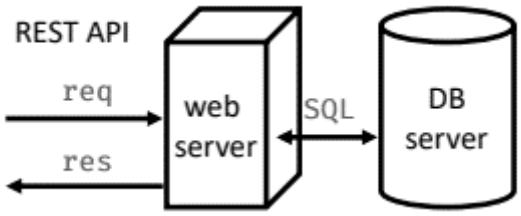


Instalação do projeto no DBMS

1. Para instalar o projeto propriamente dito, abra uma nova linha de comandos (*cmd*) e coloque o comando para mudar o diretório até à pasta específica do projeto como por exemplo `C:\Users\<username>\Downloads\Projeto`
2. Ligue-se através do terminal ao *Postgres* usando o seguinte comando `psql -h localhost -U postgres` e coloque a senha configurada anteriormente (recomendamos que seja *postgres* para maior simplicidade).
3. Após entrar, crie uma base de dados usando `create database projeto;`
4. Saia com o comando `quit`
5. Novamente, na mesma posição do ponto 1, coloque o seguinte comando para a criação de todos os parâmetros da base de dados dentro da base de dados recentemente criada, com nome projeto `psql -h localhost -U postgres -d projeto -f db_create.sql`. É importante que esteja no diretório do ponto 1, uma vez que de outra forma o arquivo *db_create.sql* não será encontrado.
6. A base de dados encontra-se agora configurada podendo ser desde já utilizada pelo *Postman*. Em alternativa, importe os dados de teste pré-criados para que possa ter dados iniciais para teste, usando o comando `psq -h localhost -U postgres -d projeto -f db_import.sql.`
7. O projeto encontra-se agora devidamente importado e pronto a usar.

Manual de utilização

Após a configuração necessária ao funcionamento da aplicação, conforme descrito no manual de instalação, é necessário proceder à utilização do ambiente criado. Para tal, o presente manual de utilização pretende dar a conhecer os *endpoints* onde é possível submeter *requests* através do *Postman* e obter os *results*, de acordo com a seguinte figura.



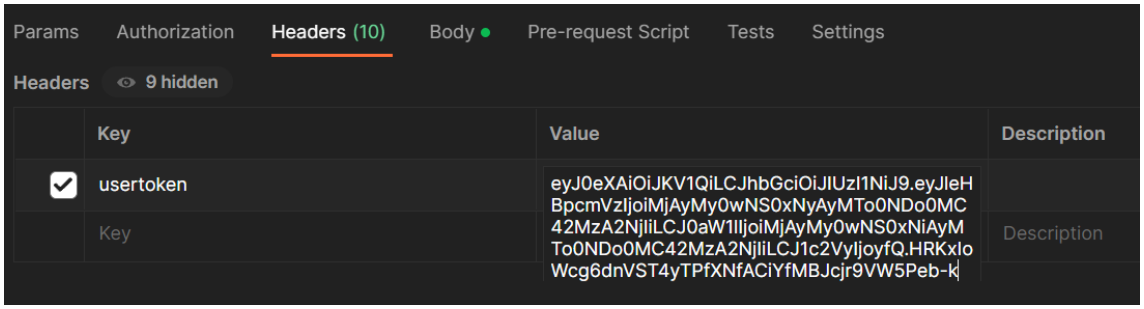
Assim, passamos à explicação dos *endpoints*;

Nota: Nos casos em que estiver indicado *Errors*, o utilizador poderá desde já assumir que irá receber um status de erro com uma descrição pormenorizada do problema em questão, não sendo por isso incluído o erro no manual e sim toda a descrição de uma utilização correta para que não sejam originados erros.

{{baseURL}} conforme indicado no *Postman* diz respeito a https://localhost/dbproj

Colocação de token [Transversal]

Em caso de necessidade de colocação de uma *token* para a execução de uma *request*, deverá ser feito o login primeiramente, conforme descrito abaixo no *endpoint* Login. Após a extração da *token* gerada, cole-a na opção *Headers* conforme indicado abaixo, em qualquer uma das *requests* que pretender enviar.



Nota: Lembre-se que deverá fazer esta colocação da *token* apenas uma vez após o login, em cada uma das *requests* que pretender enviar ao servidor. A *token* é válida por 24 horas. É igualmente necessário, caso pretenda trocar de conta, que faça login com a conta desejada antes de enviar a *request*.

Registo de novos utilizadores

Cria um utilizador na plataforma. Em caso de inexistência de uma *token*, a conta será do tipo *consumer*. Caso tenha um *token* associada em conta do tipo *admin* a conta criada será do tipo artista, uma vez que os administradores são os únicos utilizadores do sistema com capacidade para criar conta desse tipo. Mais se indica que se um administrador pretender criar uma conta de *consumer* para algum cliente que tenha dificuldade, deverá desassociar a *token* da sessão ativa.

REQUEST -> [POST] {{baseURL}}/user

```
{
    "username" : "Username",                        # Obrigatório
    "password" : "Password",                        # Obrigatório
    "email" : "email@utilizador.com",               # Obrigatório
    "name" : "Nome do utilizador",                  # Obrigatório
    "address" : "Endereço do utilizador",           # Obrigatório
    "artistic_name" : "Nome artístico",             # Opcional (Artistas)
    "label" : "Nome da editora"                     # Opcional (Artistas)
}
```

RESULT -> User ID em caso de sucesso || Erros

A aplicação fará a verificação se uma conta com os mesmos dados de *username* e email já existem, caso sobre os quais não irá criar novo utilizador.

Login de utilizador

Faz login em conta de utilizador. Não é, portanto, necessário neste *request* inserir o *token* em qualquer momento. Deverá apenas introduzir o *username* e password, momento em que será autenticado e atribuída uma *token* para a sua sessão.

REQUEST -> [PUT] {{baseURL}}/user

```
{
    "username" : "Username",                        # Obrigatório
    "password" : "Password",                        # Obrigatório
}
```

RESULT -> Token de sessão em caso de sucesso || Erros

Adicionar uma música

Insere uma música na plataforma. Só é possível inserir uma música na plataforma se a conta cuja sessão foi iniciada com login é de um utilizador do tipo Artista. De outra forma não será possível adicionar uma música. É por isso relevante que a conta em que inicia sessão, seja de um artista.

REQUEST -> [POST] {{baseURL}}/song

```
{
    "name" : "Nome da música",                      # Obrigatório
    "release_date" : "DD-MM-AAAA",                  # Obrigatório
    "genre" : "Género musical",                     # Obrigatório
    "duration" : 5, # em minutos                   # Obrigatório
    "album" : "Título do álbum",                    # Opcional (se tiver)
    "label" : "Nome da editora",                    # Obrigatório
```

```
    "other_artists" : ["Artista 1", "Artista 2", ...]
      # Opcionais (manter [])
}
```

RESULT -> Song ID em caso de sucesso || Erros

Adicionar um álbum

Insere um álbum na plataforma. À semelhança da funcionalidade anterior, a capacidade de adicionar um álbum à plataforma está reservada apenas a utilizadores do tipo Artista. É ainda necessário que esse utilizador tenha músicas publicadas na plataforma para que as possa associar ao álbum (recorra ao ponto anterior para adicionar músicas).

REQUEST -> [POST] {{baseURL}}/album

```
{
    "name" : "Nome do álbum",                # Obrigatório
    "release_date" : "DD-MM-AAAA",           # Obrigatório
    "label" : "Nome da editora",             # Obrigatório
    "songs" : ["Música 1", "Música 2", ...]  # No mín. uma música obrigatória
}
```

RESULT -> Album ID em caso de sucesso || Erros

Pesquisar uma música

Pesquisa uma música na plataforma através de uma *keyword*. A formação de pesquisa irá então procurar a existência daquela *keyword* no meio dos nomes das músicas para averiguar se alguma música com *keyword* igual existe. Retorna uma lista de zero ou mais elementos encontrados para a respetiva pesquisa.

REQUEST -> [GET] {{baseURL}}/song/{keyword}

```
Não possui payload nesta request. O pedido é feito via formação de URL.
```

RESULT -> Retorna uma lista com o formato {('Título', '[Artistas]', '[Albuns]'), (), …} em caso de sucesso || Erros

Pesquisar um artista

Pesquisa os detalhes de um artista da plataforma, através de um ID de artista fornecido através da URL do *endpoint*. Devolve então todas as informações relativas ao artista na plataforma.

REQUEST -> [GET] {{baseURL}}/artist_info/{artist_id}

```
Não possui payload nesta request. O pedido é feito via formação de URL.
```

RESULT -> Retorna uma lista com o formato {('Artistic name', '[Songs]', '[Albuns]', '[Playlists]'), (), …} em caso de sucesso || Erros

Subscrever ao Premium

Subscreve ao serviço premium da plataforma, o qual irá permitir que o utilizador tenha acesso às funcionalidades de criação de playlist. Estas playlists poderão ser privadas ou públicas de acordo com as preferências do utilizador que as cria. Após o término do período de subscrição, um

utilizador que tenha playlists criadas terá acesso às mesmas apenas se a visibilidade for pública. Se a visibilidade for marcada como privada, depois da perda do estatuto de premium o utilizador perde também acesso a essas mesmas playlists. As playlists privadas são de acesso exclusivo do utilizador que as cria, não podendo um outro utilizador premium aceder.

REQUEST -> [POST] {{baseURL}}/subscription

```
{
    "period" : "month",                 # Obrigatório
    "cards" : [1,2]                     # Opcional (manter []) no entanto não
      conseguirá fazer a subscrição por falta de cartões associados
}
```

RESULT -> Subscription ID em caso de sucesso || Erros

Criar uma playlist

Cria uma playlist apenas sob as condições de que o utilizador autenticado seja um consumidor premium com subscrição ativa. De outra forma, não terá acesso à funcionalidade.

REQUEST -> [POST] {{baseURL}}/playlist

```
{
    "name" : "Nome da playlist",              # Obrigatório
    "visibility" : "public ou private",       # Obrigatório
    "songs" : ["Música 1","Música 2", ...]    # No mín. uma música obrigatória
}
```

RESULT -> Playlist ID em caso de sucesso || Erros

Reproduzir uma música

Reproduz uma música caso um utilizador do tipo *consumer* esteja autenticado. Após a reprodução será executado um *trigger* que irá atualizar a lista de TOP 10 músicas mais tocadas/favoritas do utilizador em questão. Pelo tipo de *request* que é, não foi implementada a funcionalidade de reprodução real da música, apenas se assume como implementado.

REQUEST -> [PUT] {{baseURL}}/{song_id}

```
    Não possui payload nesta request. O pedido é feito via formação de URL.
```

RESULT -> Apenas status code em caso de sucesso || Erros

Gerar um cartão pré-pago

Permite a criação de um ou vários cartões pré-pagos e com um determinado valor. Este valor repercute-se na quantidade de cartões que se pretende criar, resultando numa determinada quantidade de cartões de igual valor de saldo.

REQUEST -> [PUT] {{baseURL}}/card

```
{
    "number_cards" : 2,                       # Obrigatório
    "card_price" : 25                         # Obrigatório
}
```

RESULT -> Retorna uma lista com o formato {(‘Card ID’, ‘Card ID’, …} em caso de sucesso || Erros

Comentar uma música ou responder a comentário

Permite deixar um comentário numa música ou responder a um comentário já existe. O utilizador deverá estar autenticado para conseguir comentar. O comentário, em caso de ser *top-level*, não deverá incluir o *parent_id*. Por outro lado, se for uma resposta a um comentário já existente, deverá ser preenchido. O sistema é automático sabendo distinguir a existência ou não de um *parent_id* nas suas rotas em *Flask*.

REQUEST -> [PUT] {{baseURL}}/comments/{song_id}

REQUEST -> [PUT] {{baseURL}}/comments/{song_id}/{parent_id}

```
{
    "content": "Comentário a ser publicado”              # Obrigatório
}
```

RESULT -> Comment ID em caso de sucesso || Erros

Criar um relatório mensal

Cria um relatório mensal para o top de músicas por género e quantidade de reproduções durante um mês, no limite até aos últimos 12 meses. Este relatório só pode ser gerado com uma sessão iniciada.

REQUEST -> [GET] {{baseURL}}/report/{year}/{month}

```
Não possui payload nesta request. O pedido é feito via formação de URL.
```

RESULT -> Retorna uma lista com o formato {(‘Month’, ‘Genre’, ‘Playback amount’), (), …} em caso de sucesso || Erros