INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

ISEL | DEETC

ÁREA DEPARTAMENTAMENTAL DE ENGENHARIA DE
ELECTRÓNICA E TELECOMUNICAÇÕES E DE COMPUTADORES

Mestrado em Engenharia
Informática e Computadores

Engenharia de Software

Relatório Final

Ano Lectivo:
2023/2024

Aluno:
45125 - Luís FONSECA

Docentes:
Eng. Luis MORGADO
Eng. Jorge BRANCO

14/01/2024

Contents

List of Figures

1 Introdução

O presente trabalho tem como intuito um maior entendimento sobre a área de engenharia de *software*. Para tal, foi desenvolvida uma aplicação *web*, através das práticas dadas nas aulas.

A aplicação desenvolvida consiste num *website* de música, de nome ***Tunespot*** com o objetivo de dar a conhecer e consultar diferentes tipos de música, assim como conhecer diferentes tipos de músicas, bandas e artistas.

De forma a seguir todas as boas práticas para a maior facilidade na construção da aplicação teve se em conta o seguinte processo de desenvolvimento:

- Especificação de Requisitos,
- Arquitetura Lógica,
- Arquitetura Detalhada,
- Implementação.

Devido aos requisitos se manterem constantes durante todo o processo desenvolvimento, foi adotada esta estrutura, possibilitando uma boa divisão das partes, diminuindo a complexidade em cada delas.

2 Enquadramento teórico

No que diz respeito à construção de uma aplicação, é necessário ter em conta alguns dos conhecimentos teóricos de *software*. Uns dos conceitos consiste na **arquitetura da aplicação**, onde consta no desenvolvimento, operação e manutenção de *software* de maneira sistemática e quantificável.

Algumas das caraterísticas e métricas para o desenvolvimento de *software* são:

Complexidade: é o grau de dificuldade de previsão das propriedades de um sistema dadas as propriedades das partes individuais. A mesma está relacionada com a informação que é necessária para a caracterização de um sistema.

Um sistema é tanto mais complexo quando mais informação seja necessária para a sua descrição.

Entropia: É a medida do grau de dispersão relativa que existe num sistema fechado num dado instante de tempo.

2.1 Métricas de Arquitetura

Quando falamos do desenvolvimento de uma arquitetura é necessário ter em conta algumas das suas métricas. Essas métricas são as seguintes:

Coesão: Nível coerência funcional de um subsistema/módulo (até que ponto esse módulo realiza uma única função):

- Um nível de coesão baixo leva a que, em caso de necessidade de alteração de um subsistema, o número de módulos afectados seja elevado.
- Se o nível de coesão for elevado, o número de módulos afectados será minimizado.
- Um módulo com um nível de coesão baixo é mais complexo, logo mais difícil de conceber e de testar.

Acoplamento: Grau de interdependência entre subsistemas. Com um nível de acoplamento baixo obtém-se:

- Maior facilidade de desenvolvimento, instalação, manutenção e expansão.
- Melhor escalabilidade, devido à possibilidade de distribuição e replicação de módulos.
- Maior tolerância de falhas, uma vez que a falha de um subsistema tem um impacto restrito.

Simplicidade: Nível de facilidade de compreensão e comunicação da arquitectura implementada

Adaptabilidade: Nível de facilidade de alteração da arquitectura para incorporação de novos requisitos ou de alterações nos requisitos previamente definidos

2.2 Princípios

Abstração: Processo de descrição de conhecimento a diferentes níveis de detalhe e tipos de representação. É uma ferramenta base para lidar com a complexidade

Modularidade: Composta pela **decomposição** do sistema em partes coesas, o **encapsulamento** de dos detalhes internos do sistema e, as **interfaces** de comunicação com o exterior.

Fatorização: Processo de redução de redundância através de **herança** e **delegação**.

2.3 Padrão *Model View Controller*

Para a arquitetura de *software* foi utilizada uma arquitetura de ***três camadas***, no entanto podem ser definidas várias camadas que reduzem o acoplamento entre níveis lógicos e físicos. No diagrama abaixo é possível de ver a representação usada para esta arquitetura.

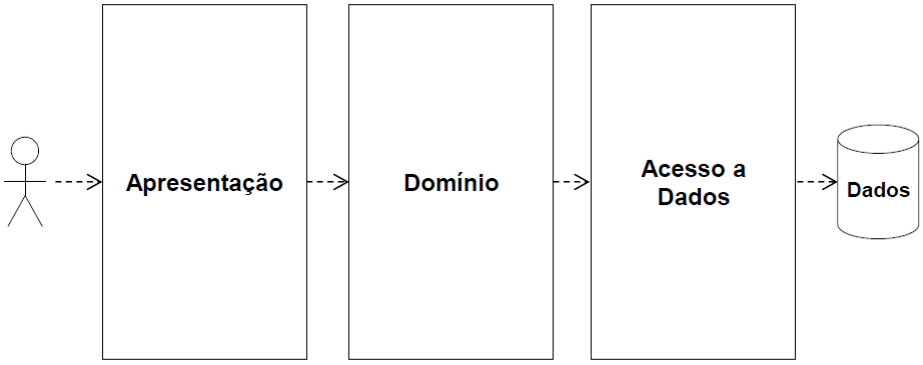


Figure 1: Arquitetura de 3 camadas

A camada de apresentação é responsável por gerir a interação do utilizador com o sistema. O domínio é responsável por manter a lógica da aplicação. Acesso a dados define a forma como os dados são acedidos

No entanto, surge o problema de ter de encontrar uma forma sucinta de organizar a relação entre as classes responsáveis pela interação com o utilizador e as classes para processamento interno. A solução é o padrão MVC (Model View Controller)

O padrão MVC (*Model View Controller*) procura separar o modelo de domínio do controlo de interação e da apresentação. Os papéis do Modelo, da Vista e do Controlador são podem ser descritos com ajuda da seguinte figura:

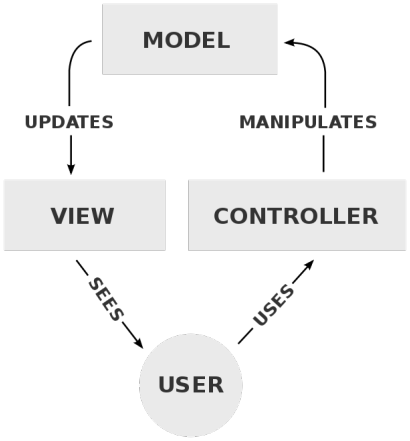


Figure 2: Padrão MVC

O **Modelo** está encarregue de gerir os dados do domínio da aplicação. As **Vistas** gerem a informação que é demonstrada ao utilizador e, o **Controlador** interpreta os *inputs* do utilizador.

A **vista** é responsável por apresentar os dados ao utilizador

O **Controlador** é responsável por gerir o fluxo dos dados entre o modelo e a vista. Contém as classes que gerem as ações dos utilizadores e que solicitam dados ao modelo. Existem também outras variantes inspiradas no padrão MVC, como por exemplo o padrão Model- View-Presenter (MVP) e Model-View-ViewModel (MVVM).

Sendo assim, do ponto de vista de um utilizador, o mesmo faz *requests* ao controlador, que por sua vez manipula o modelo, que afeta diretamente as vistas que o utilizador vê.

2.4 Redução de Acoplamento

Uma forma de reduzir o acomplamento em sistemas, é o uso de uma camada de fachada. Isto permite que todo o apenas haja uma interface de comunicação entre camadas e, não seja tudo ligado diretamente.

A fachada disponibiliza uma interface uniforme apartir de um conjunto de interfaces internas de um sistema.

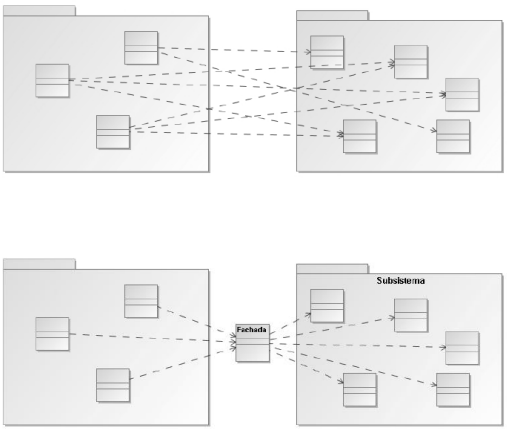


Figure 3: Uso de uma fachada

No contexto da aplicação, vai ser utilizada uma fachada para traduzir os dados vindos da apresentação para a camada de domínio. Como a fachada vair possuir dependências não irá inserir diretamente na camada de domínio.

3 Análise de requisitos

A análise de requisitos é responsável por recolher dados indispensáveis para o desenvolvimento de um sistema. A ideia é começar com uma visão geral do problema e, chegar a requisitos concretos do sistema.

A **visão** geral consiste no documento de visão, onde se descreve o sistema em termos gerais, o contexto operacional, as partes envolvidas, as características do sistema, etc. Este documento define, de forma geral, o problema assim como a solução.

Um **glossário** foi também desenvolvido com o propósito de prevenir falhas de comunicação. Habitualmente são utilizados glossários entre cliente-fornecedor para que ambos se consigam entender na especificação de requisitos ou até mesmo em falhas que possam eventualmente acontecer. Este documento permite estabelecer uma linguagem comum entre o cliente e o fornecedor.

Com estes pilares bem estabelecidos (visão e glossário), é possível especificar os requisitos. Esta especificação é realizada através de **casos de utilização** (outras metodologias tais como o *Agile* podem utilizar outros aspetos de especificação de requisitos tais como a *user stories*) e **especificação suplementar**.

Os **casos de utilização** representam os requisitos funcionais, que são elaborados na perspetiva dos utilizadores e, a **especificação suplementar** requisitos não funcionais, especificação suplementar de funcionalidades.

O presente projeto inclui os seguintes casos de utilização:

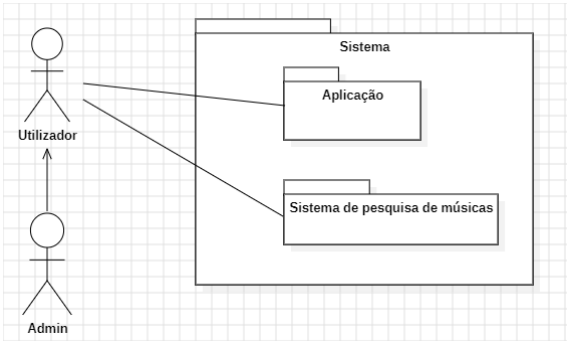


Figure 4: Casos de utilização: Funcionamento geral

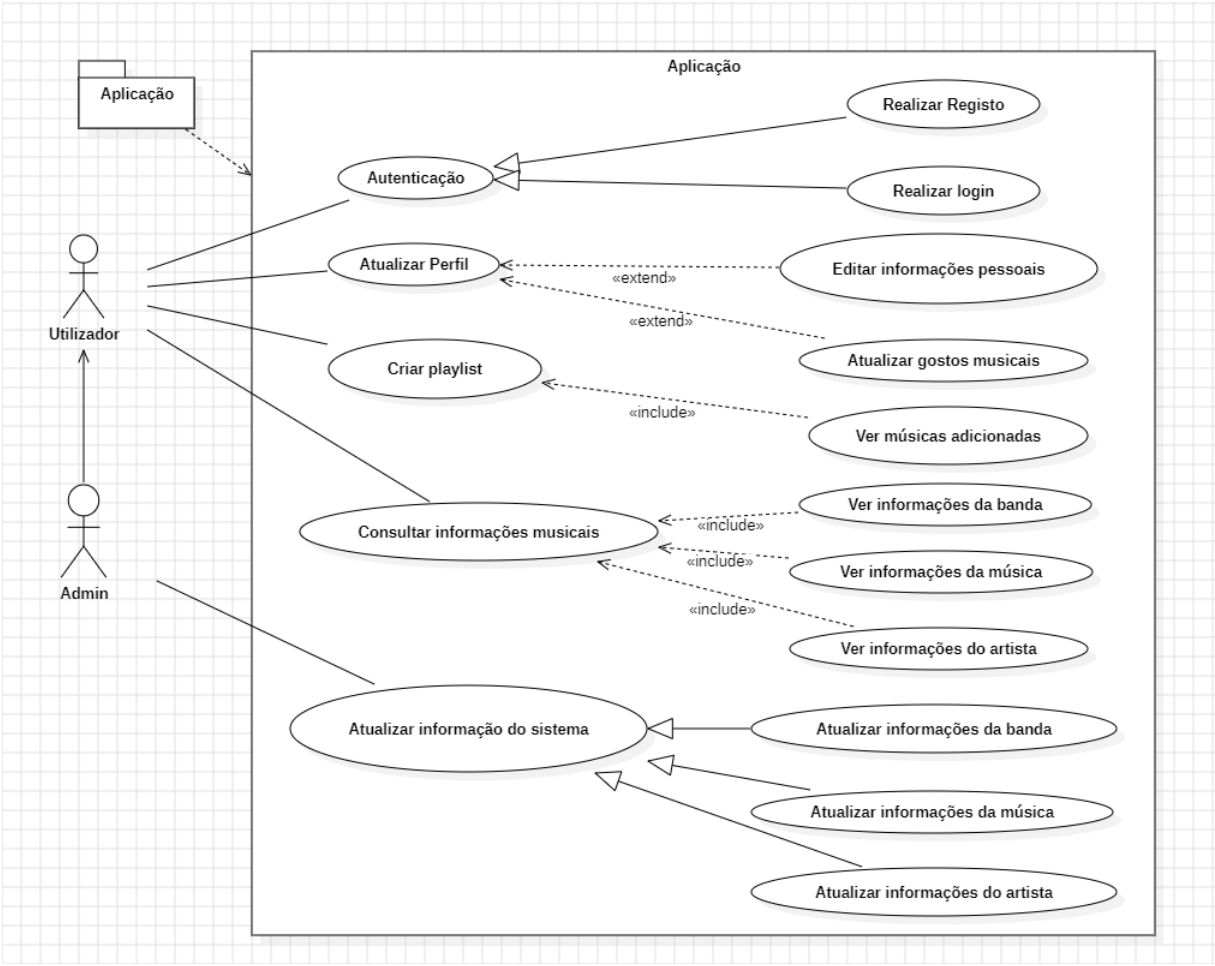


Figure 5: Casos de utilização: Aplicação

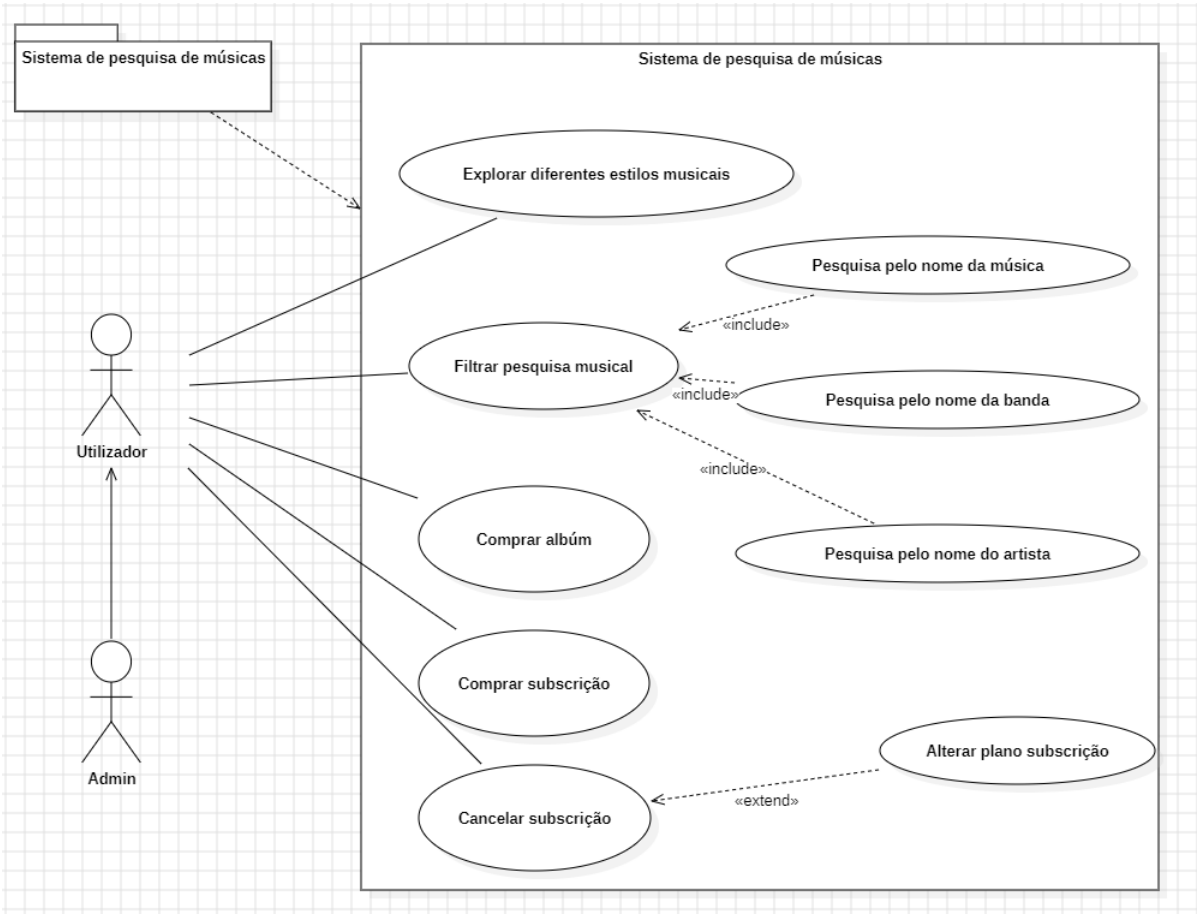


Figure 6: Casos de utilização: Sistema de pesquisa

De forma a manter coesão entre as partes foram utilizadas relações entre casos, tais como:

- Relação de extensão.
- Relação de inclusão.
- Relação de generalização.

A relação de generalização pretende especializar um caso de utilização abstrato.

A relação de extensão (*extend*) é opcional e, estende o comportamento do caso de utilização, adicionando mais passos.

Já a relação de inclusão (*include*) procura completar um caso de utilização, incluindo outro.

Por fim, analisando os casos de utilização detalhadamente, foi desenvolvida a específicação complementar. Nesta, constam os requisitos não funcionais tais como especificação de restrições em campos específicos do sistema para ter em conta durante o processo de construção.

4 Projeto de arquitetura da aplicação

4.1 Modelo de Domínio

O modelo de domínio, faz parte da arquitetura lógica da aplicação. Aqui, são descritos todos os objetos necessários para suportar o comportamento e os dados do sistema, num ponto de vista completamente lógico. Isto é, apesar de ser um diagrama UML de classes, não procura especificar detalhadamente a arquitetura, mas sim representar o conhecimento que o arquiteto tem sobre o domínio da aplicação.

O modelo de domínio da aplicação é o seguinte (fig. 7):

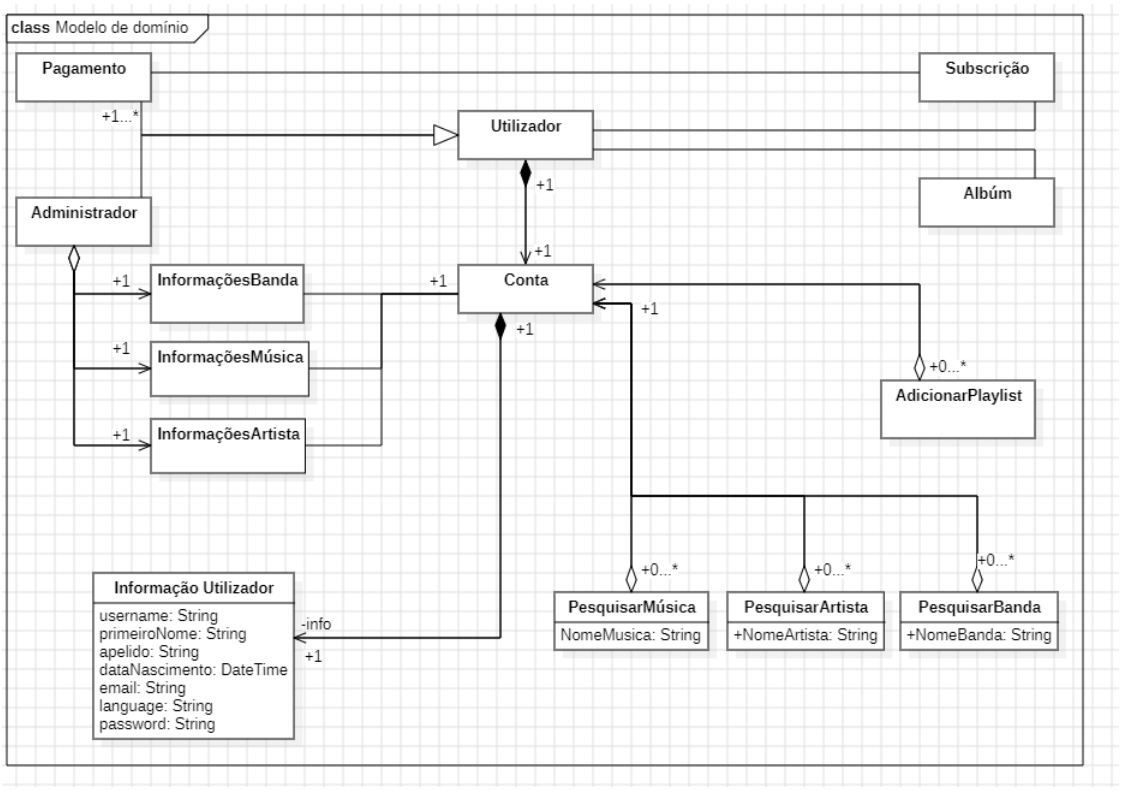


Figure 7: Modelo de Domínio

4.2 Realização dos casos de utilização

Com um modelo de domínio desenvolvido e, um padrão determinado, é necessário começar a detalhar os casos de utilização estabelecidos na Análise de Requisitos.

Para tal, são utilizados diagramas de interação e temporização que, conseguem descrever detalhadamente cada caso de utilização. Estes diagramas permitem ao arquiteto da aplicação prever o fluxo de controlo que acontece na aplicação. São estes diagramas que, devem fazer uso do conhecimento do modelo de domínio de forma a dar origem a diagramas de classes mais específicos.

Um exemplo de diagramas de interação construído é o seguinte(fig. 8):

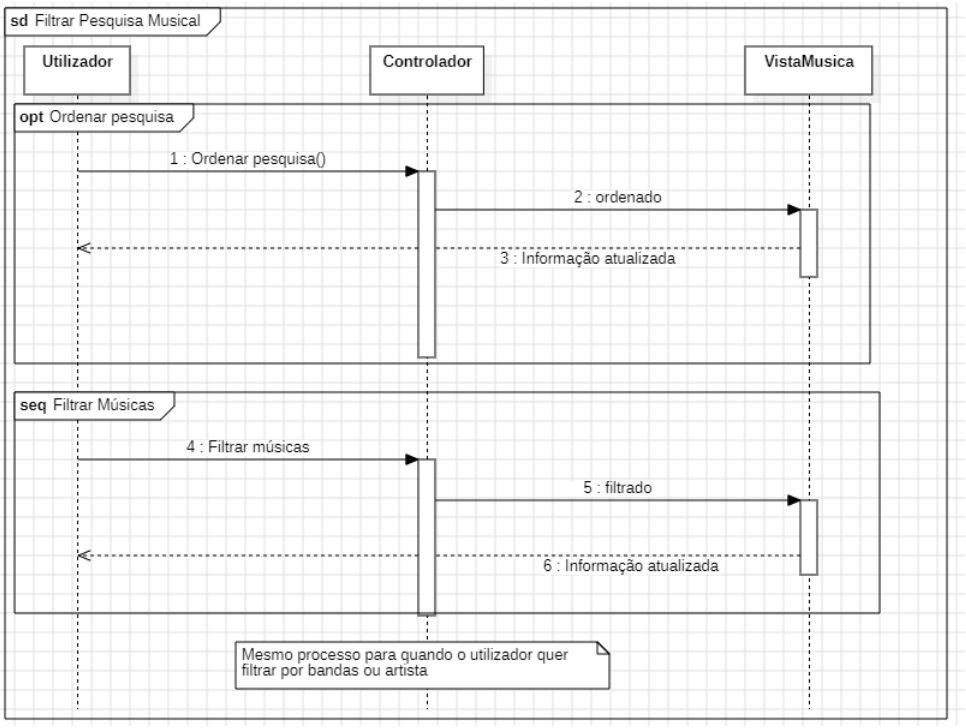


Figure 8: Diagrama de sequência para a realização de um dos casos de utilizaçãos

4.3 Arquitetura de mecanismos

A arquitetura de mecanismos, pretende especificar os mecanismos da aplicação definidos nos casos de utilização. É aqui que se deve olhar para os diagramas de interação criados anteriormente. Muitas das classes criadas podem ser retiradas diretamente da realização dos casos de utilização assim como alguns dos seus métodos. A ideia é detalhar os mecanismos necessários para tornar reais os casos de utilização.

Um exemplo do diagrama de mecanismos consiste no seguinte(fig. 9:

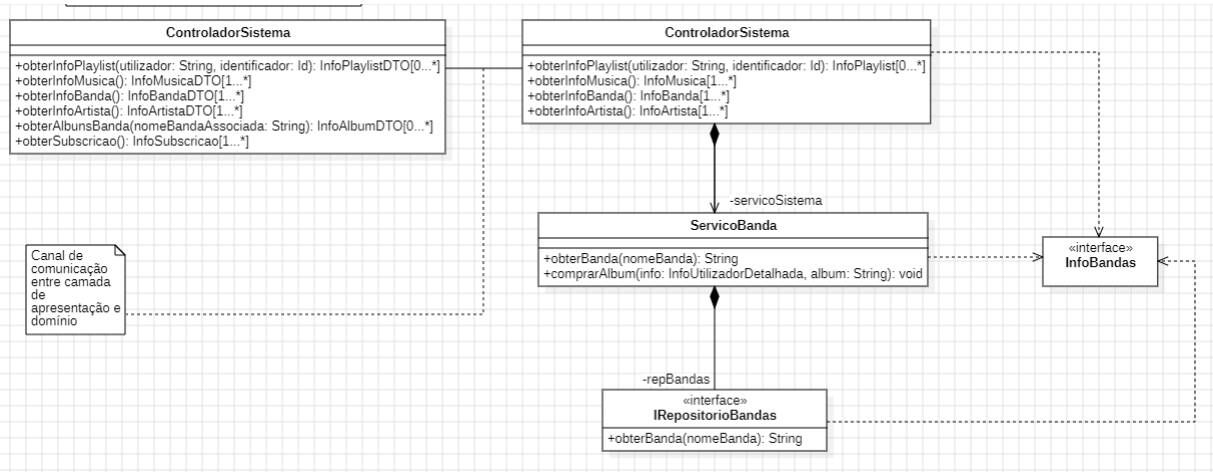


Figure 9: Diagrama de mecanismos

4.4 Arquitetura geral da solução

Com os mecanismos construídos, pretende-se abstrair um pouco mais, ficando assim com uma visão geral da solução proposta. A arquitetura geral da solução é constituída por subsistemas que, por sua vez, alojam mecanismos. De seguida pode-se observar a arquitetura geral da solução para a aplicação a desenvolver:

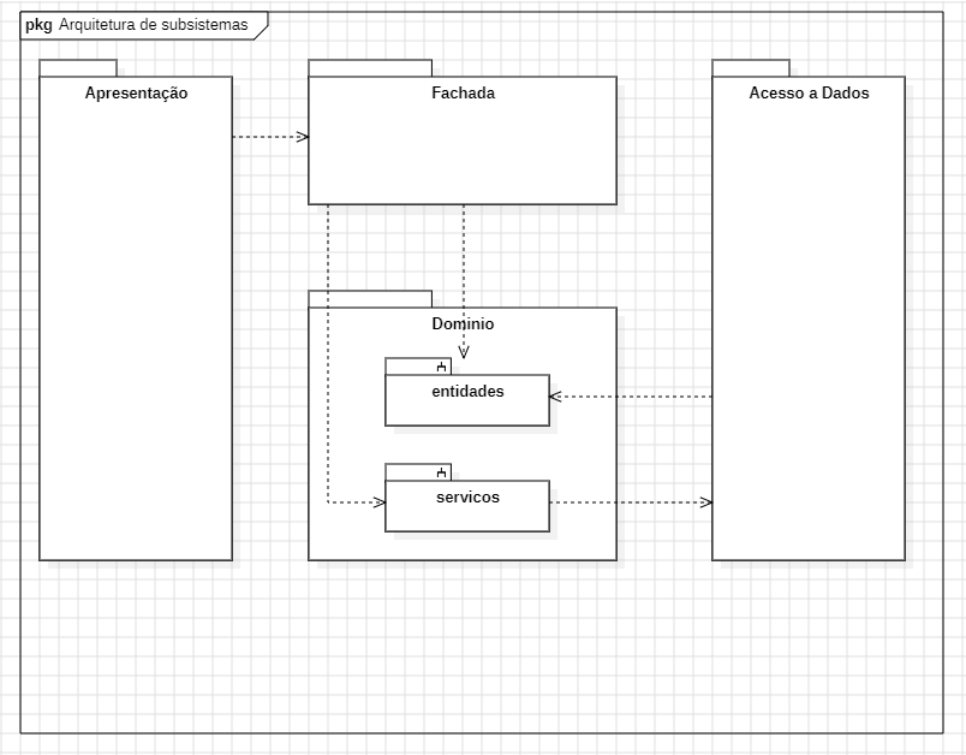


Figure 10: Diagrama de pacotes - Arquitetura geral da solução

4.5 Modelo de dinâmica

Representa a estrutura do sistema e como as diferentes partes interagem entre si. Denota também a forma de como as partes evoluem no tempo e como reagem a estímulos externos

Para esta aplicação teve se em contas os diferentes estados da aplicação. Muitos destes estados representam as diferentes atividades que o utilizador vai realizar quando navega na aplicação. É também possível de ilustrar como os diferentes estados conseguem estar relacionados entre si.

Este diagrama é iniciado quando o utilizador acede à página principal, através do seu *browser*. Assim que seleciona uma das opções, o utilizador poderá efetuar uma **pesquisa musical** pela música que deseja procurar, ou então pelo nome do artista ou pelo nome da música.

O modelo de dinâmica construído é o seguinte: (fig. 11);

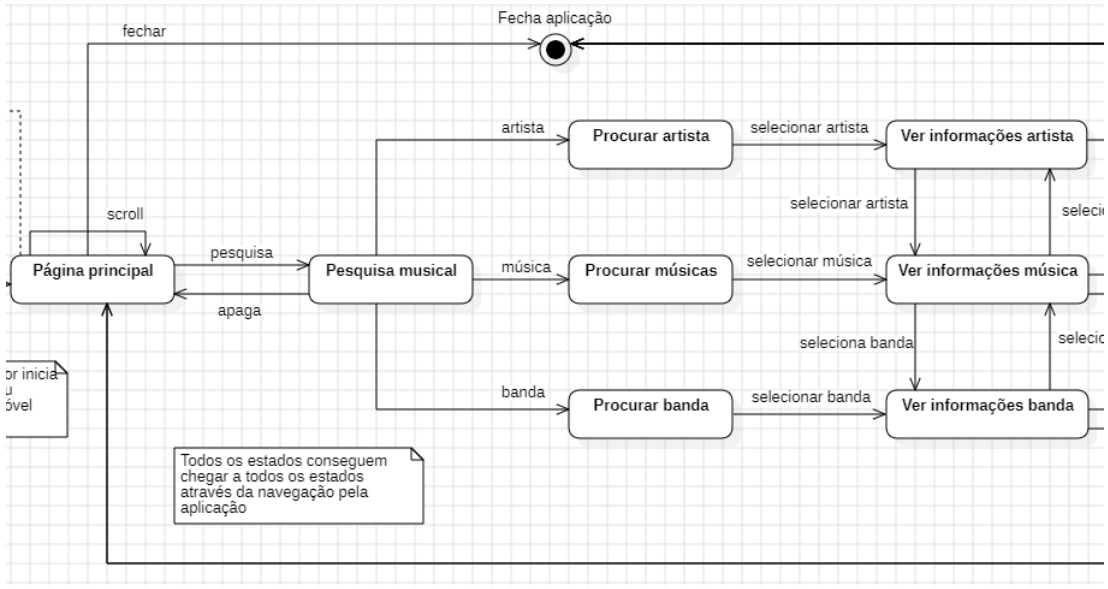


Figure 11: Diagrama do modelo de dinâmica

4.6 Arquitetura detalhada

Com os mecanismos definidos e a arquitetura geral também, resta testar as classes presentes no domínio. O protótipo de testes deve testar a camada de domínio. Esta é uma camada sem quaisquer dependências e, por isso pode ser testada utilizando apenas uma linguagem de programação pura, neste caso pode ser usado o JAVA.

Com ajuda das classes presentes nos mecanismos é possível traduzir os diagramas de classes da camada de domínio em código propriamente dito. Antes disso, é apenas necessário o desenho da aplicação de teste. Esta, deve conter uma classe por cada caso de utilização a testar de domínio. É também necessária uma classe base que, correrá os diferentes testes todos juntos.

O diagrama de classe da aplicação de teste encontra-se na seguinte figura (fig. 12):

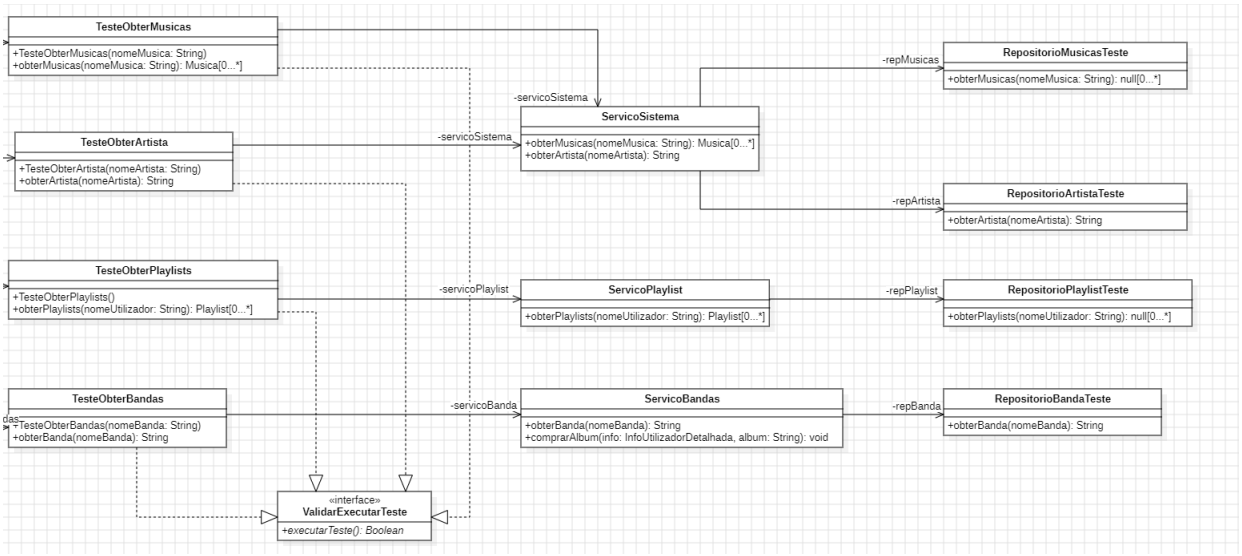


Figure 12: Diagrama UML - Protótipo de teste

É de notar que, devido à dimensão do desenho desta arquitetura, esta contém uma outra classe de nome ***TunespotApp*** onde é executado o método *main* onde permite correr a arquitetura desenhada.

No caso desta aplicação de teste, foram testadas quatro funcionalidades de domínio. É de notar que o os repositórios presentes não vão diretamente a uma base dados, retornam apenas objetos diretamente. Isto pois, a aplicação de teste apenas pretende testar o domínio e, o acesso a dados não pertence ao domínio.

5 Implementação da aplicação

Por fim, em termos de desenvolvimento de aplicação, é finalmente o criado o protótipo aplicacional. Foi definido do documento de visão, que a aplicação era na *Web*. No entanto, devido à arquitetura ter sido construída sem qualquer acoplamento com a linguagem de programação, caso fosse necessário fazer a aplicação para outro dispositivo, seria bastante possível e, sem muita complexidade em termos de arquitetura.

Para esta aplicação teste foi realizado o seguinte caso de utilização:

- Filtrar pesquisa musical.

A aplicação teste foi concebido com o seguinte *Tech Stack*:

- *ReactJS*.

Em alternativa, e caso fosse realizada uma aplicação um pouco mais complexa, iria ser usado o seguinte *tech stack*:

- *Framework* JAVA *Spring*.
- Base de dados relacional *Postgres*.

O diagrama de implantação que o sistema implementa é o seguinte (fig. 13)

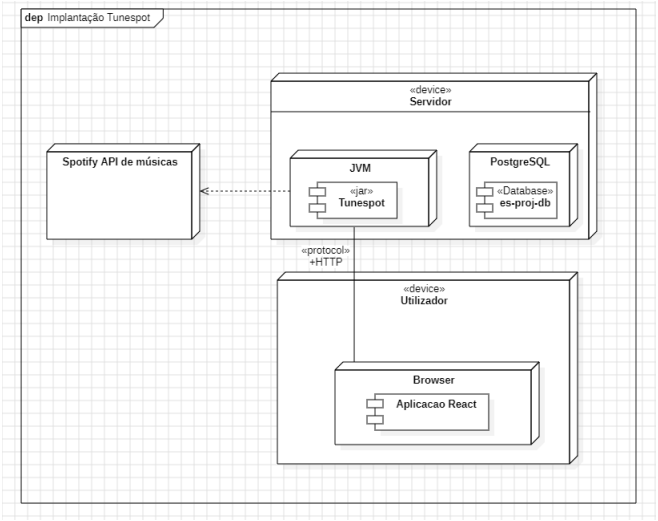


Figure 13: Diagrama de Implantação

Para a devida tradução dos diagramas de classes em código propriamente dito, foi utilizado o diagrama de detalhe de partes e mecanismos para o Domínio, Acesso a Dados e controladores de Apresentação, assim como o diagrama de mecanismo de apresentação para detalhar as vistas presentes na apresentação. Seguem-se as os dois diagramas, respetivamente:

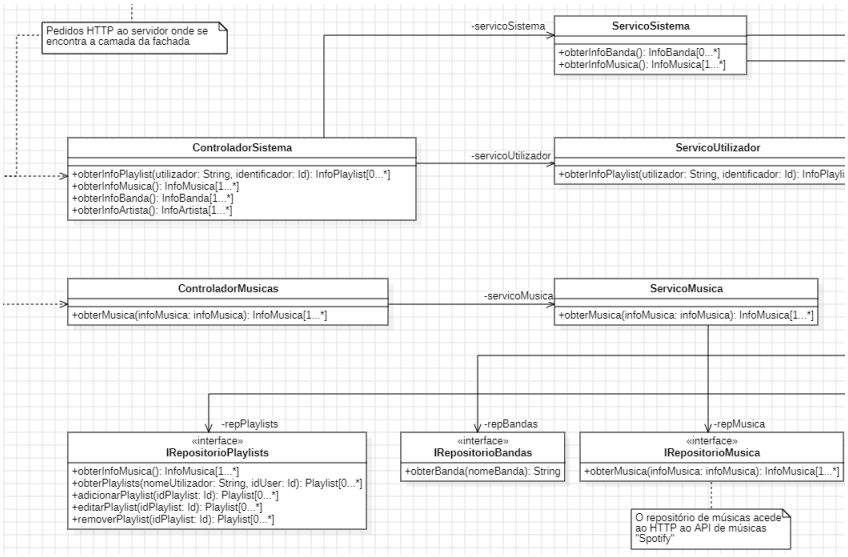


Figure 14: Diagrama UML - Detalhe de partes e mecanismos

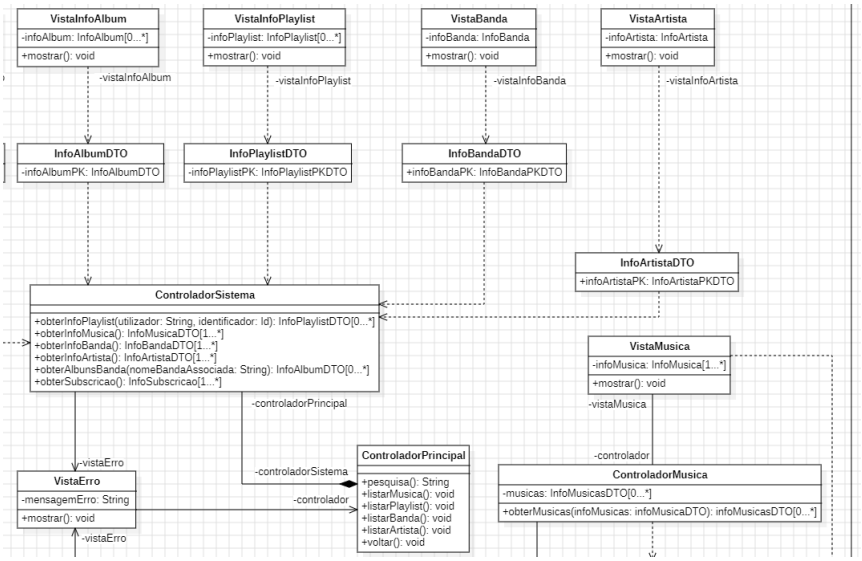


Figure 15: Diagrama UML - Mecanismo de apresentação

5.1 Domínio e Acesso a Dados

No que diz respeito ao desenvolvimento da camada de fachada, o domínio e, o acesso a dados, este pode estar presente na plataforma JAVA, podendo ser considerado o *backend* do sistema.

A razão pela qual não ter sido escolhido algo como o *NodeJS* para esta parte do trabalho, tem a ver com o facto de, não só de se estar mais habituado à linguagem de programação JAVA, mas também pois o JAVA, permite uma maior robustez devido a possuir tipos de variável, o que, claramente se traduz em menos ambiguidade e entropia no desenvolvimento.

A *Framework Spring*, disponibiliza uma forma fácil de expor *endpoints* para camada de apresentação. Estes *endpoints* são disponibilizados pela camada de fachada que, acede diretamente aos serviços presentes no domínio que, por sua vez comunicam com a base de dados ou APIs externas.

A *Framework* disponibiliza também uma forma mais fácil de conexão com bases de dados, não tendo de estar a transferir o *drivers* para o programa comunicar com a base de dados específica.

Assim, o *JAVA Spring*, facilitou em grande escala o trabalho que se teria a criar esta parte da aplicação.

5.2 API de Música - *Spotify*

Esta foi uma API utilizada para a recolha de músicas na escrita da palavra. De todas as APIs encontradas, o *Spotify*, foi aquela que é a mais completa e fácil de utilizar, tendo em conta a linguagem de programação Javascript.

Todos os repositórios (músicas, bandas e artistas) podem comunicar diretamente com APIs externas. Através desta API é possível, fazer a pesquisa pelo nome da banda, do artista e da música, com elevado detalhe na informação recebida.

5.3 Apresentação

O *ReactJS* é onde se situa a camada de apresentação e, possuí todos os controladores e vistas. O uso da biblioteca *Javascript*, desenvolvida pela empresa *Meta* (antigo *Facebook*), deve-se à facilidade que o *ReactJS* dá na construção de aplicações na *web*. Para criação de interfaces de utilização complexas, aconselha-se a utilização de bibliotecas que facilitam este processo.

O *ReactJS*, não só ajuda no desenvolvimento da **interface de utilizador**, como tem uma boa **performance**.

Os controladores e vistas foram desenvolvidos como sendo componentes *React*. Isto para que, os controladores consigam um rápido acesso ao *input* dos utilizadores e, as vistas tenham acesso direto à informação que é mostrada ao utilizador.

A aplicação ficou com o seguinte aspeto (fig. 16):

Figure 16: Implemetação da aplicação

6 Análise crítica do projeto realizado

Tendo em conta o desenvolvimento do projeto, salienta-se a existência de alguns pontos fortes, assim como pontos fracos. Demarca-se o facto dos casos de utilização não terem sido detalhados o melhor possível inicialmente, o que deu origem a um resultado diferente do final apresentado. Apenas foi detetado o erro nos casos de utilização quando se tinha começado a implementação dos mesmos. No entanto, como ainda se estaria numa parte prematura do projeto, o erro foi retificado e, adicionalmente foi realizada uma revisão à visão do problema e, todos os casos de utilização de forma a que, não fosse possível outro erro relacionado.

O problema deste acontecimento tem a ver com, não estando bem estruturados os casos de utilização, todas as etapas a seguir não iram estar corretas. Nesta sequência, revela-se o facto de quanto mais preliminar o problema, uma maior taxa de erros constará no projeto.

Para prevenir situações como esta, pode-se adotar uma metodologia de desenvolvimento com um dos princípios ágeis, o uso de iterações. Realizando o projeto de forma iterativa, uma equipa é de certa forma obrigada a rever os casos de utilização por cada iteração.

Houve também pontos mais fortes que, permitiram um desenvolvimento do sistema de forma simples e eficaz. A arquitetura não ficou complexa, podendo ser reproduzida em qualquer sistema independentemente das linguagens utilizadas. Foram impedidas improvisações de ultima hora, apenas se seguiu o diagrama e desenvolveu-se o código como o descrito.

7 Conclusões

Em boa verdade foi um projeto imensamente denso. No inicio, foi deveras difícil desviar o olhar das tecnologias a utilizar. Como se constrói um sistema, sem deliberar sobre a tecnologia. A verdade é que as linguagens de programação apenas ajudam a traduzir instruções para os computadores perceberem. Esta unidade curricular deu a entender que, com uma boa arquitetura, as falhas são drasticamente menores. Não estando "'preso" a uma linguagem de programação, é desenhado um sistema muito fácil de entender, com menor complexidade e, por sua vez, menor entropia.

As linguagens são ferramentas certamente importantes nos dias de hoje, mas sem uma arquitetura bem desenhada é sempre impossível a criação de sistemas reais, de grande escala, que garantem confiabilidade aos clientes.

Uma pergunta inicial que pareceu estranho no início mas, rapidamente se entranhou foi, como se cria uma arquitetura para um programa escrito funcionalmente? A forma como se realizou no presente trabalho (na parte de *ReactJS*) foi, apenas seguir a arquitetura. Se calhar não seriam desenhadas classes, mas sim componentes que, cumpriam tudo o que estava presente na arquitetura, todavia fora de um paradigma orientado a objetos.

Como principais conceitos tem-se que, a arquitetura deve ser sempre pensada inicialmente como um problema não dependente de uma tecnologia ou plataforma, pois dependências como essas podem vir a causar um desvios na forma de resolver problemas que poderiam ser resolvidos com mais simplicidade. Tem-se também que, a análise de requisitos não deve nunca ser descartada, pelo que é daí que se determina o domínio do problema e tudo o resto. Descartar um pilar como estes leva logo a caminhos de grande entropia que, muitas vezes não levam a lado nenhum.

8 Anexo A - Implementação de um caso de utilização

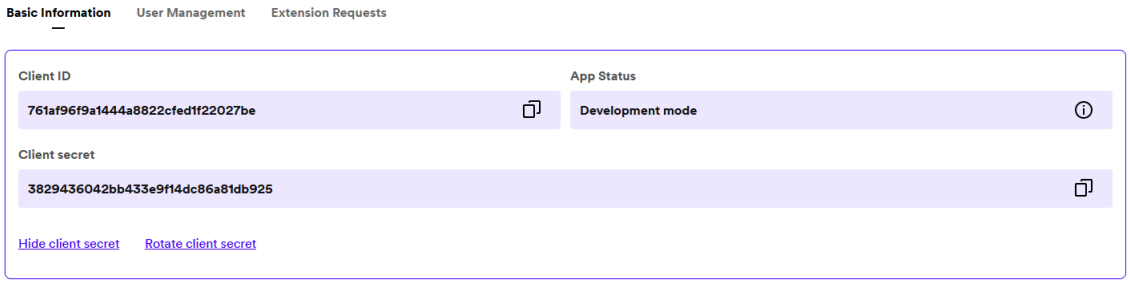


Figure 17: Spotify Dashboard

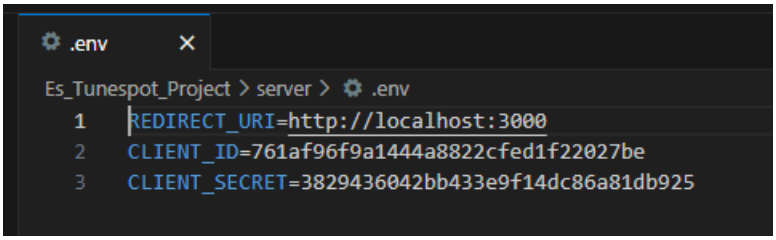


Figure 18: Ficheiro ENV com dados do *Spotify Dashboard*

```
app.post("/refresh", (req, res) => {
  const refreshToken = req.body.refreshToken
  const spotifyApi = new SpotifyWebApi({
    redirectUri: process.env.REDIRECT_URI,
    clientId: process.env.CLIENT_ID,
    clientSecret: process.env.CLIENT_SECRET,
    refreshToken,
  })

  spotifyApi
    .refreshAccessToken()
    .then(data => {
      res.json({
        accessToken: data.body.accessToken,
        expiresIn: data.body.expiresIn,
      })
    })
    .catch(err => {
      console.log(err)
      res.sendStatus(400)
    })
})
```

Figure 19: Ligação servidor

```jsx
export default function TrackSearchResult({ track, chooseTrack }) {
  function handlePlay() {
    chooseTrack(track)
  }

  return (
    <div
      className="d-flex m-2 align-items-center"
      style={{ cursor: "pointer"}}
      onClick={handlePlay}
    >
      <img src={track.albumUrl} style={{ height: "64px", width: "64px" }} />
      <div className="ml-3">
        <div>{track.title}</div>
        <div className="text-muted">{track.artist}</div>
      </div>
    </div>
  )
}
```

Figure 20: Desenvolvimento do ”filtrar por pesquisa musical”

9 Anexo B - Arquitetura da solução

9.1 Modelo de domínio

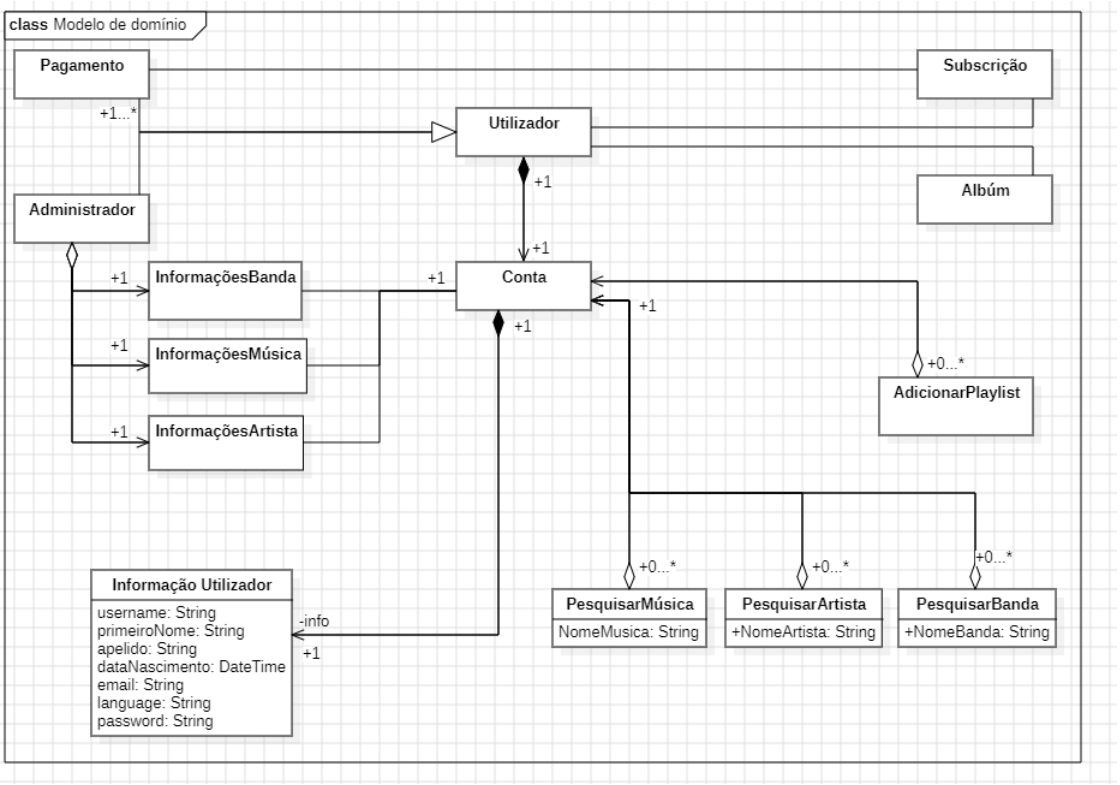
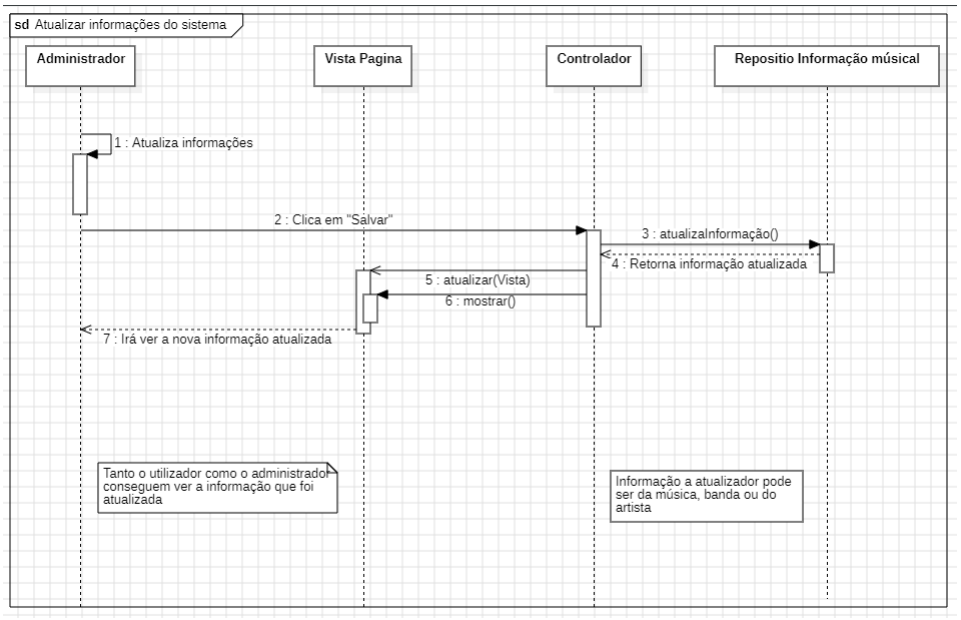
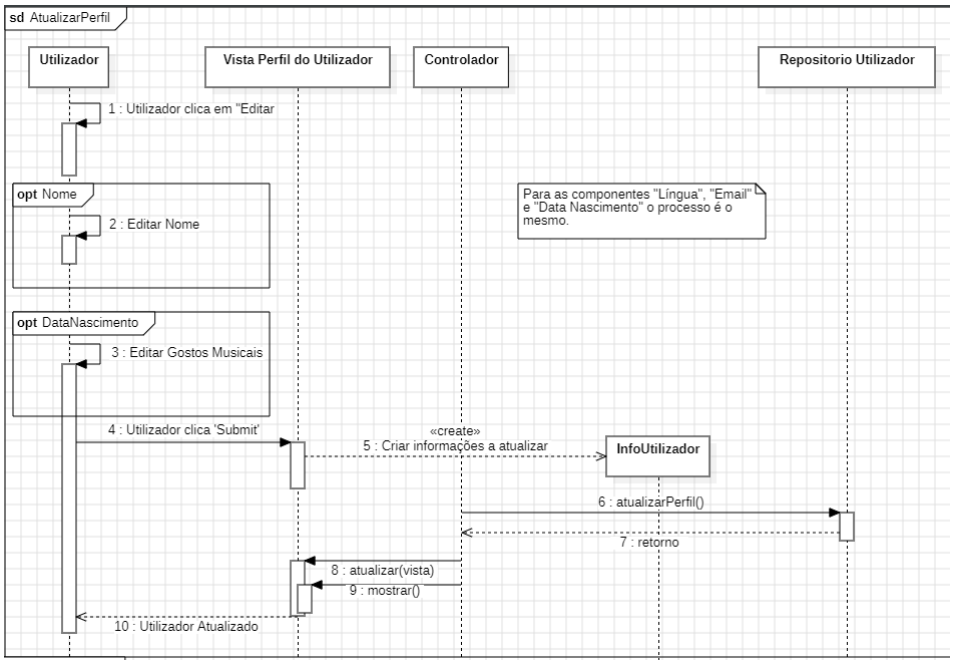


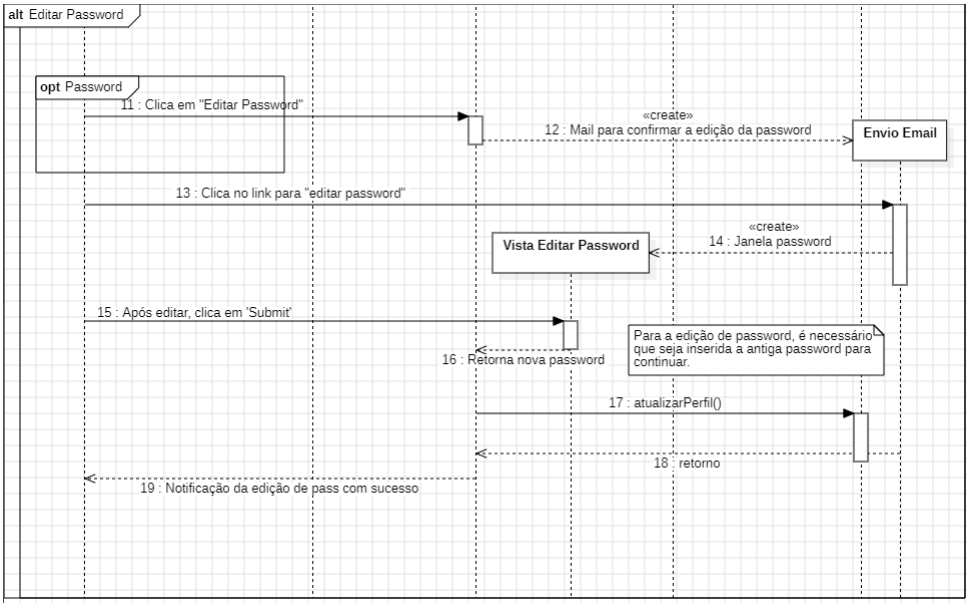
Figure 21: Modelo de Domínio

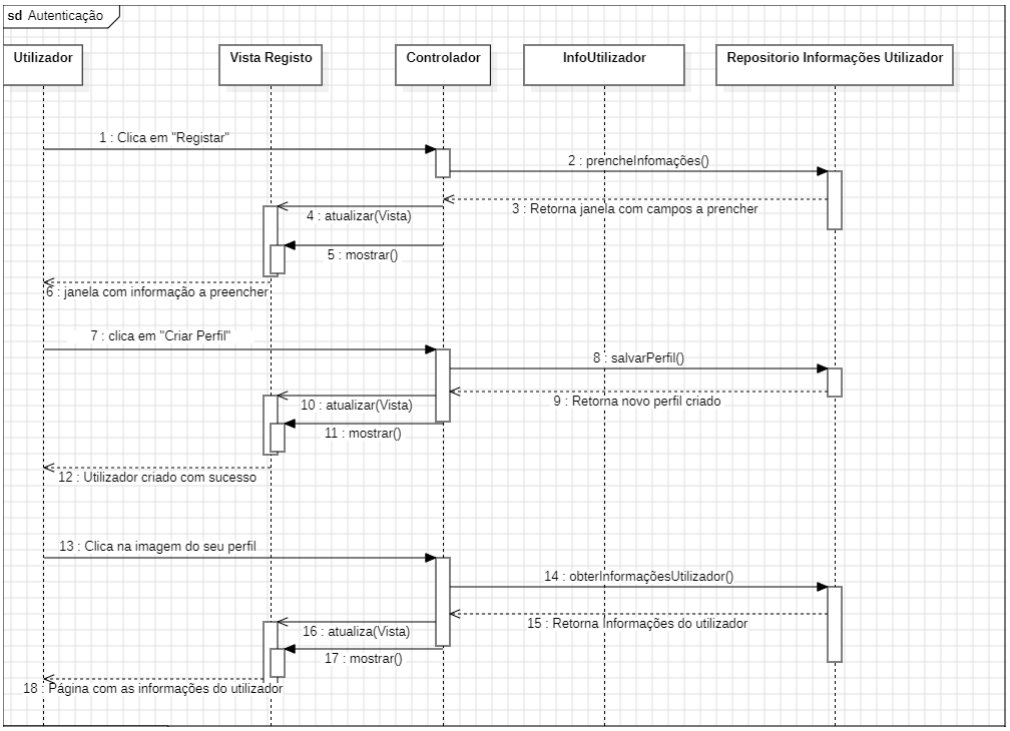
9.2 Arquitetura Lógica

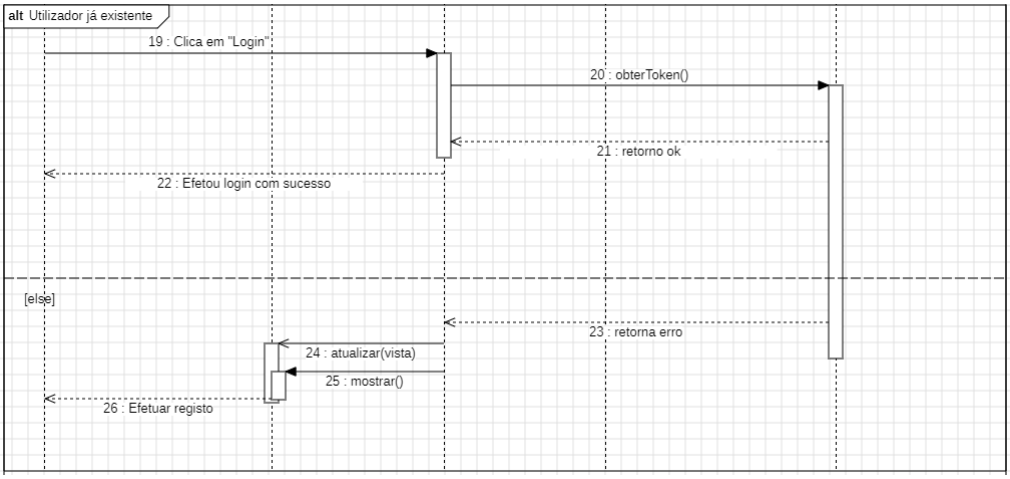
9.2.1 Modelo de interação

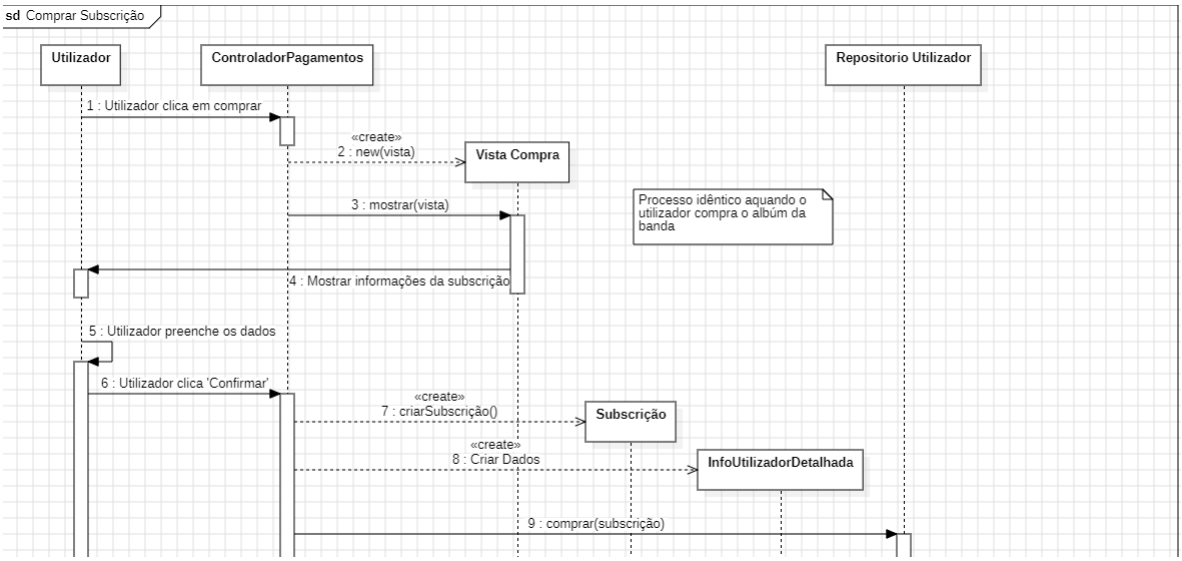


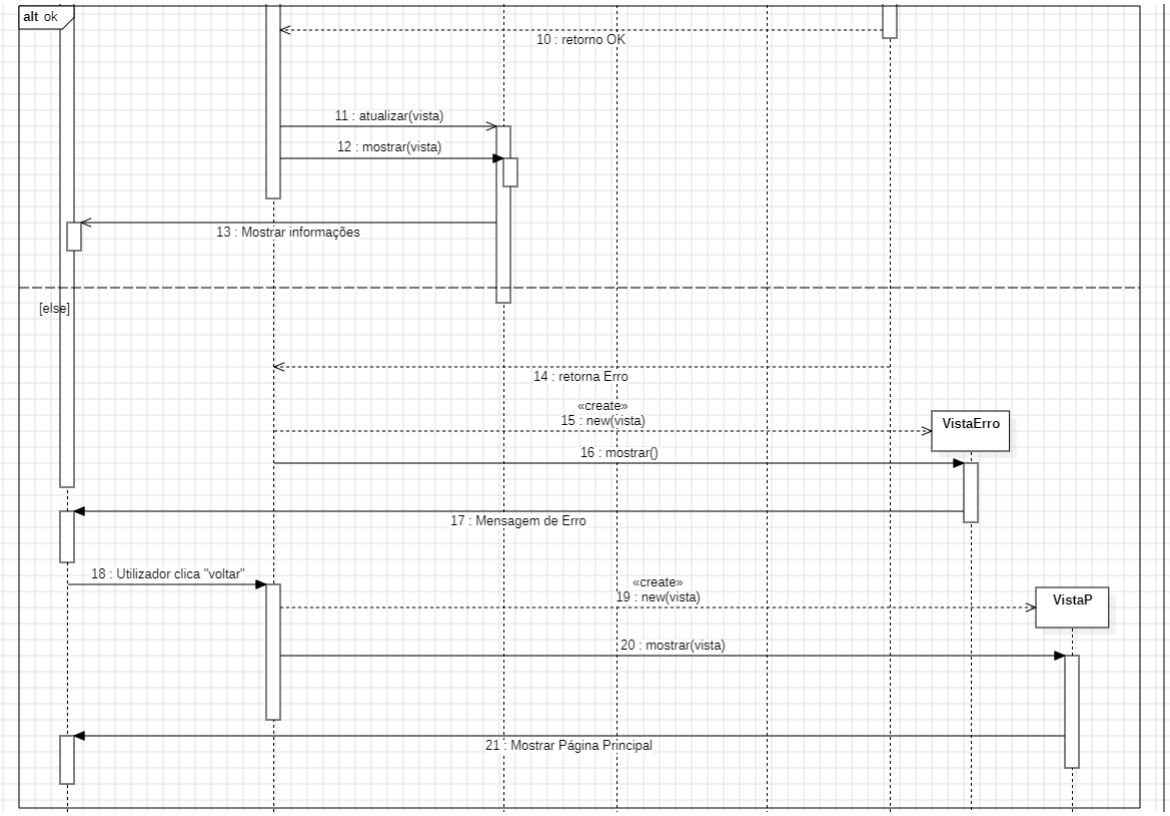


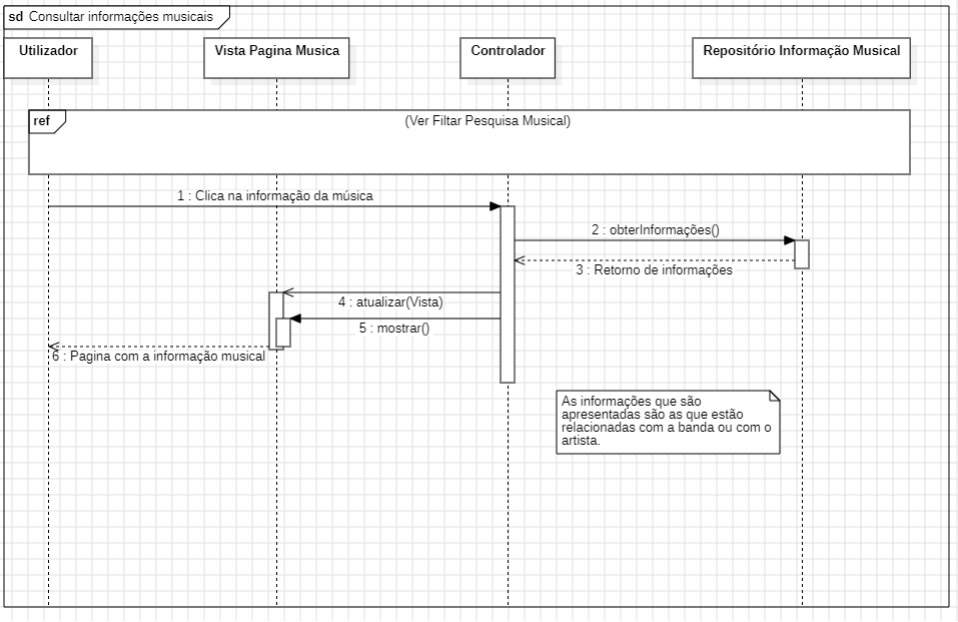

alt Editar Password
opt Password
11 : Clica em "Editar Password"
«create»
12 : Mail para confirmar a edição da password
Envio Email
13 : Clica no link para "editar password"
«create»
14 : Janela password
Vista Editar Password
15 : Após editar, clica em 'Submit'
16 : Retorna nova password
Para a edição de password, é necessário que seja inserida a antiga password para continuar.
17 : atualizarPerfil()
18 : retorno
19 : Notificação da edição de pass com sucesso

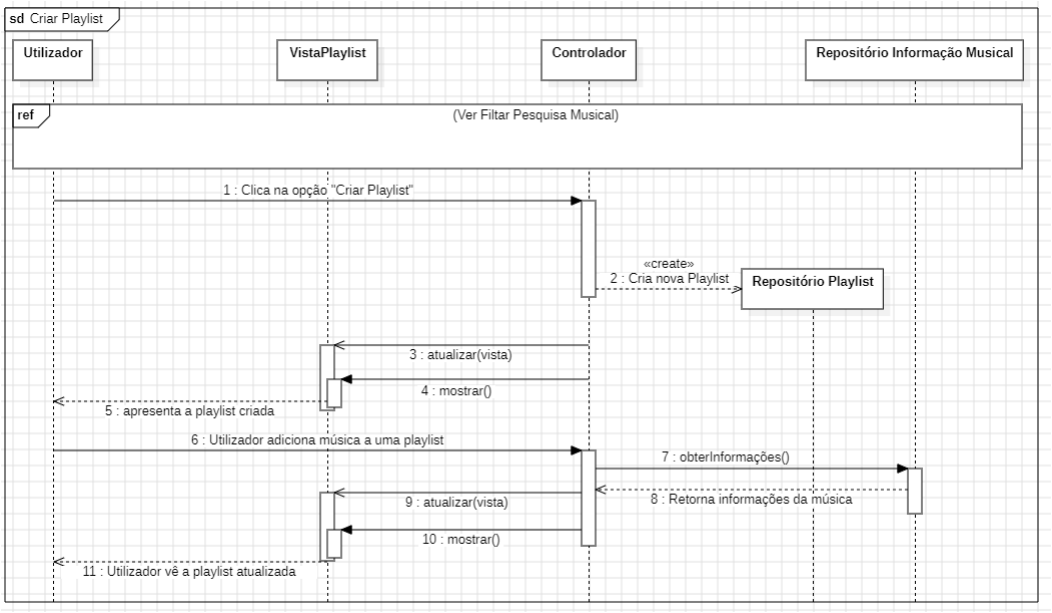

sd Autenticação
Utilizador
Vista Registo
Controlador
InfoUtilizador
Repositorio Informações Utilizador
1 : Clica em "Registar"
2 : prencheInfomações()
3 : Retorna janela com campos a prencher
4 : atualizar(Vista)
5 : mostrar()
6 : janela com informação a preencher
7 : clica em "Criar Perfil"
8 : salvarPerfil()
9 : Retorna novo perfil criado
10 : atualizar(Vista)
11 : mostrar()
12 : Utilizador criado com sucesso
13 : Clica na imagem do seu perfil
14 : obterInformaçõesUtilizador()
15 : Retorna Informações do utilizador
16 : atualiza(Vista)
17 : mostrar()
18 : Página com as informações do utilizador

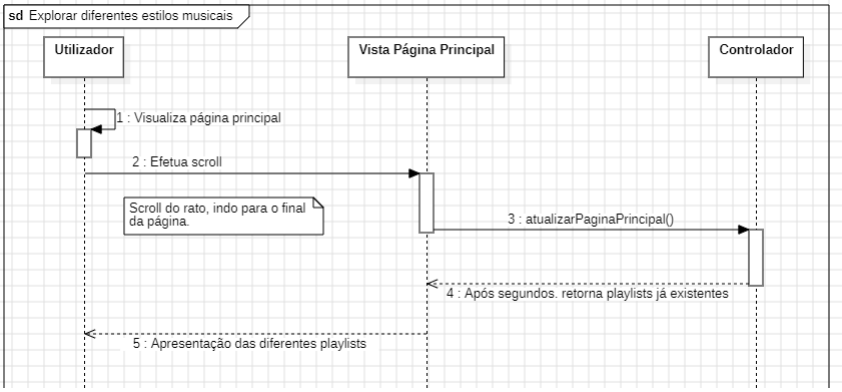

alt Utilizador já existente
19 : Clica em "Login"
20 : obterToken()
21 : retorno ok
22 : Efetou login com sucesso
[else]
23 : retorna erro
24 : atualizar(vista)
25 : mostrar()
26 : Efetuar registo

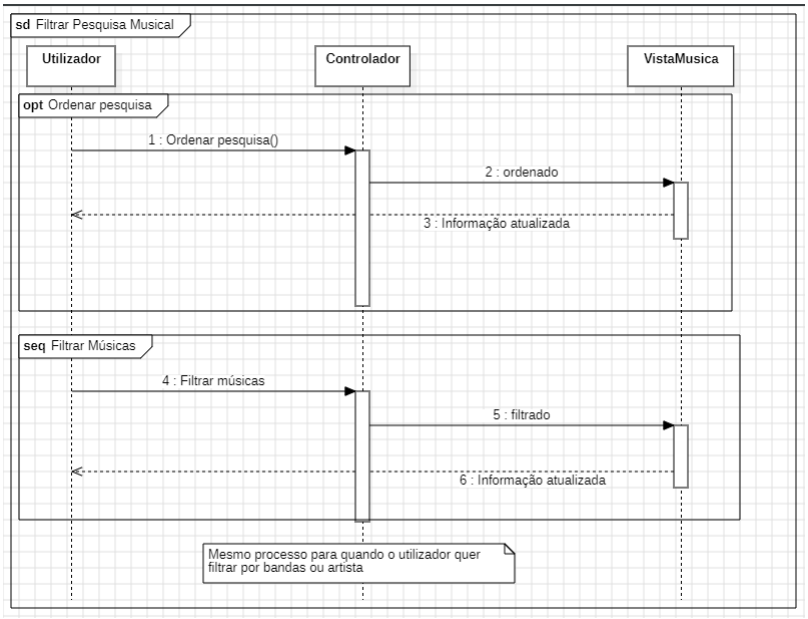

sd Comprar Subscrição
Utilizador
ControladorPagamentos
Repositorio Utilizador
1 : Utilizador clica em comprar
«create»
2 : new(vista)
Vista Compra
3 : mostrar(vista)
Processo idêntico aquando o utilizador compra o albúm da banda
4 : Mostrar informações da subscrição
5 : Utilizador preenche os dados
6 : Utilizador clica 'Confirmar'
«create»
7 : criarSubscrição()
Subscrição
«create»
8 : Criar Dados
InfoUtilizadorDetalhada
9 : comprar(subscrição)


alt ok
10 : retorno OK
11 : atualizar(vista)
12 : mostrar(vista)
13 : Mostrar informações
[else]
14 : retorna Erro
«create»
15 : new(vista)
VistaErro
16 : mostrar()
17 : Mensagem de Erro
18 : Utilizador clica "voltar"
«create»
19 : new(vista)
VistaP
20 : mostrar(vista)
21 : Mostrar Página Principal

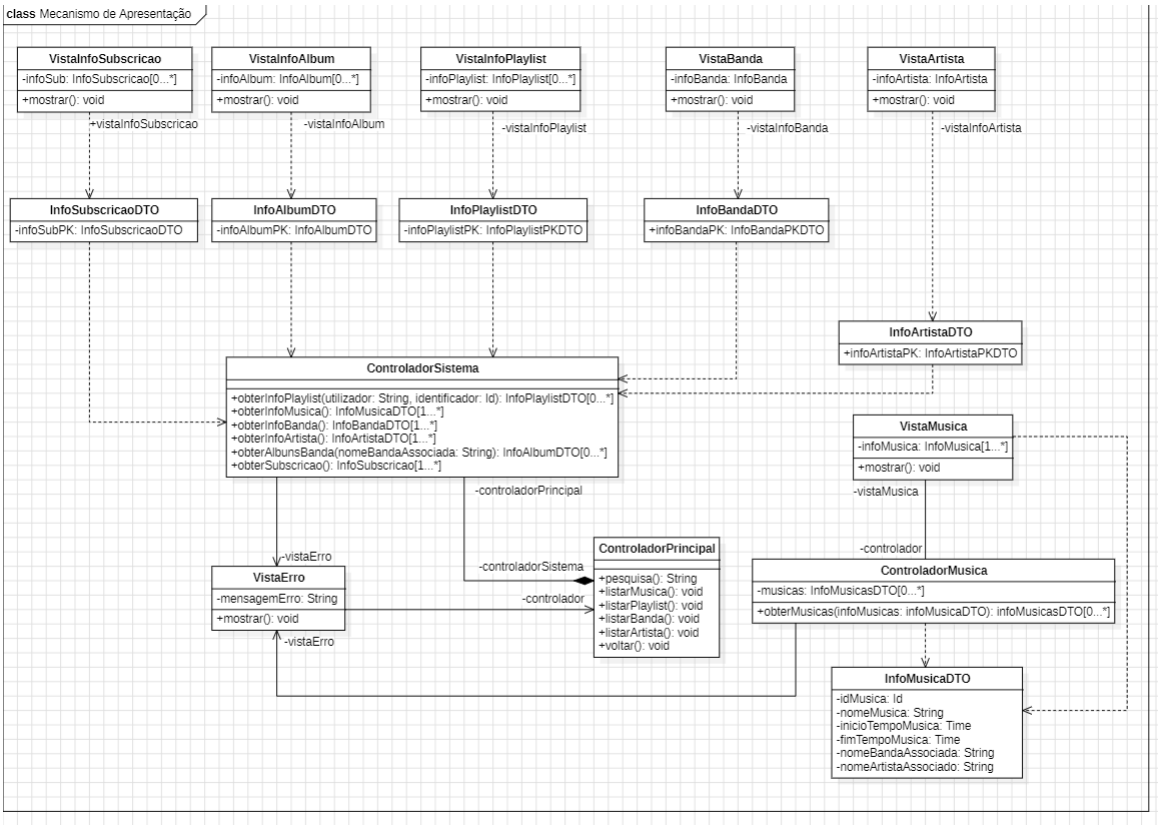

sd Consultar informações musicais
Utilizador
Vista Pagina Musica
Controlador
Repositório Informação Musical
ref
(Ver Filtar Pesquisa Musical)
1 : Clica na informação da música
2 : obterInformações()
3 : Retorno de informações
4 : atualizar(Vista)
5 : mostrar()
6 : Pagina com a informação musical
As informações que são apresentadas são as que estão relacionadas com a banda ou com o artista.

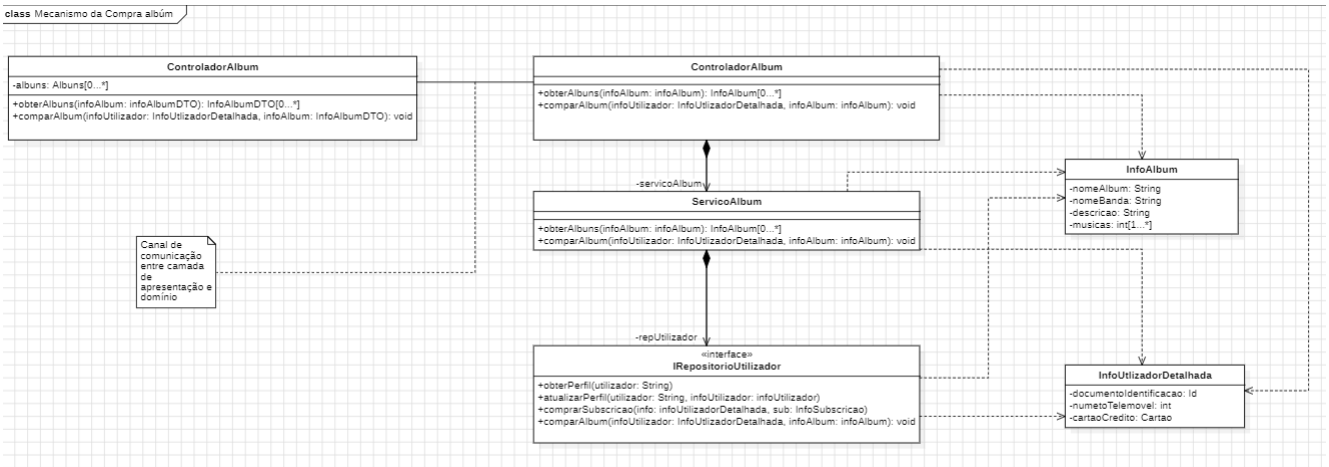

sd Criar Playlist
Utilizador
VistaPlaylist
Controlador
Repositório Informação Musical
ref
(Ver Filtar Pesquisa Musical)
1 : Clica na opção "Criar Playlist"
«create»
2 : Cria nova Playlist
Repositório Playlist
3 : atualizar(vista)
4 : mostrar()
5 : apresenta a playlist criada
6 : Utilizador adiciona música a uma playlist
7 : obterInformações()
8 : Retorna informações da música
9 : atualizar(vista)
10 : mostrar()
11 : Utilizador vê a playlist atualizada

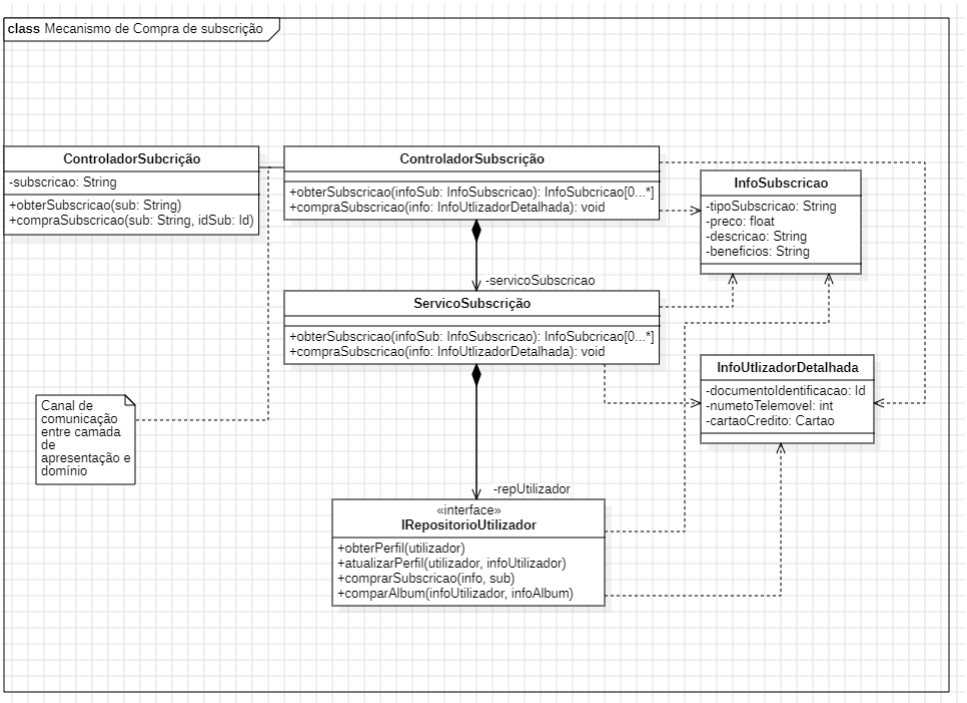

sd Explorar diferentes estilos musicais
Utilizador
Vista Página Principal
Controlador
1 : Visualiza página principal
2 : Efetua scroll
Scroll do rato, indo para o final da página.
3 : atualizarPaginaPrincipal()
4 : Após segundos. retorna playlists já existentes
5 : Apresentação das diferentes playlists

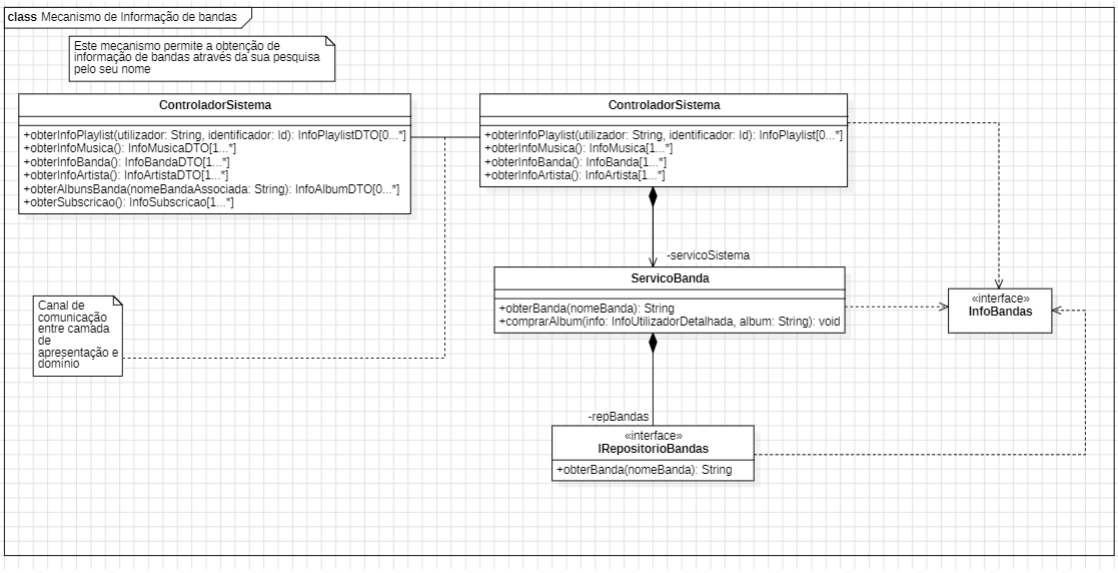

sd Filtrar Pesquisa Musical
Utilizador
Controlador
VistaMusica
opt Ordenar pesquisa
1 : Ordenar pesquisa()
2 : ordenado
3 : Informação atualizada
seq Filtrar Músicas
4 : Filtrar músicas
5 : filtrado
6 : Informação atualizada
Mesmo processo para quando o utilizador quer filtrar por bandas ou artista

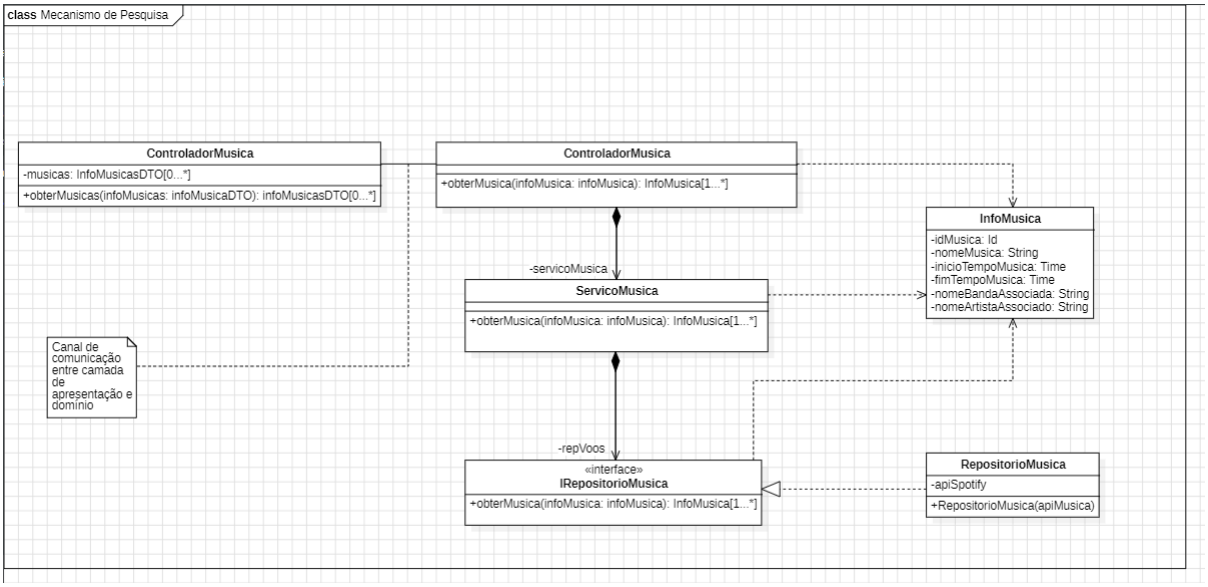
9.2.2 Arquitetura de mecanimos

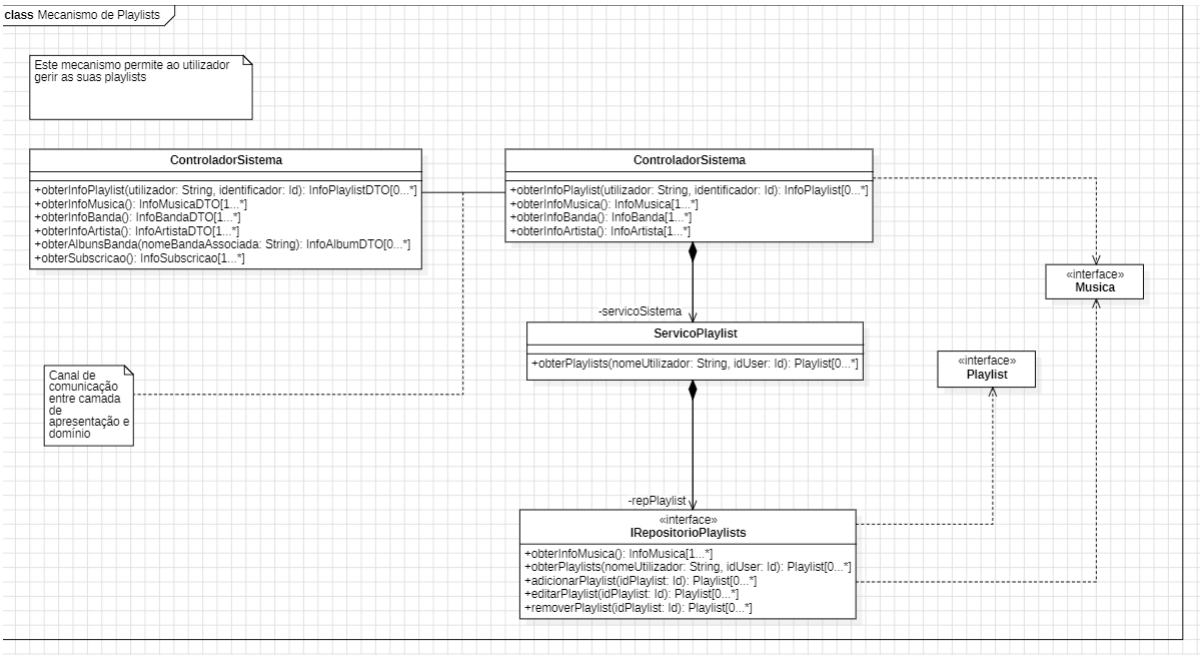




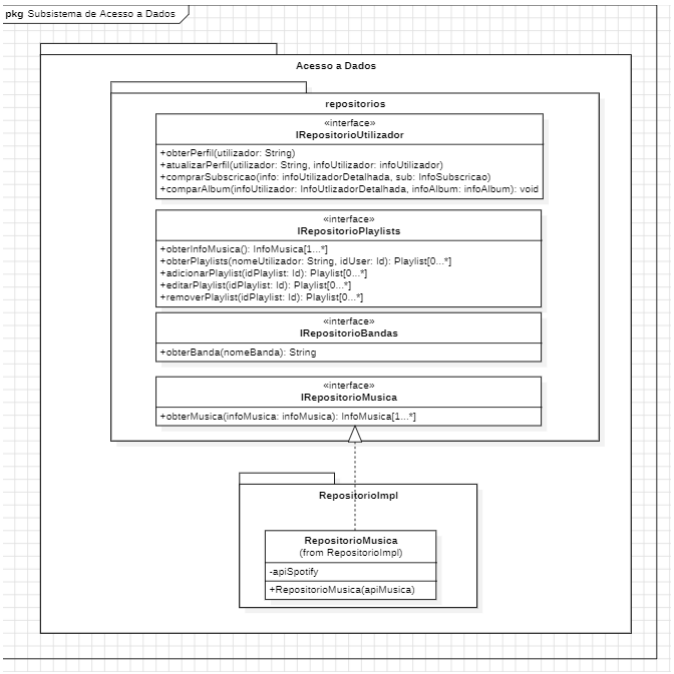

class Mecanismo de Compra de subscrição
ControladorSubcrição
-subscricao: String
+obterSubscricao(sub: String)
+compraSubscricao(sub: String, idSub: Id)
ControladorSubscrição
+obterSubscricao(infoSub: InfoSubscricao): InfoSubcricao[0...*]
+compraSubscricao(info: InfoUtlizadorDetalhada): void
InfoSubscricao
-tipoSubscricao: String
-preco: float
-descricao: String
-beneficios: String
-servicoSubscricao
ServicoSubscrição
+obterSubscricao(infoSub: InfoSubscricao): InfoSubcricao[0...*]
+compraSubscricao(info: InfoUtlizadorDetalhada): void
InfoUtlizadorDetalhada
-documentoIdentificacao: Id
-numetoTelemovel: int
-cartaoCredito: Cartao
Canal de comunicação entre camada de apresentação e domínio
-repUtilizador
«interface»
IRepositorioUtilizador
+obterPerfil(utilizador)
+atualizarPerfil(utilizador, infoUtilizador)
+comprarSubscricao(info, sub)
+comparAlbum(infoUtilizador, infoAlbum)

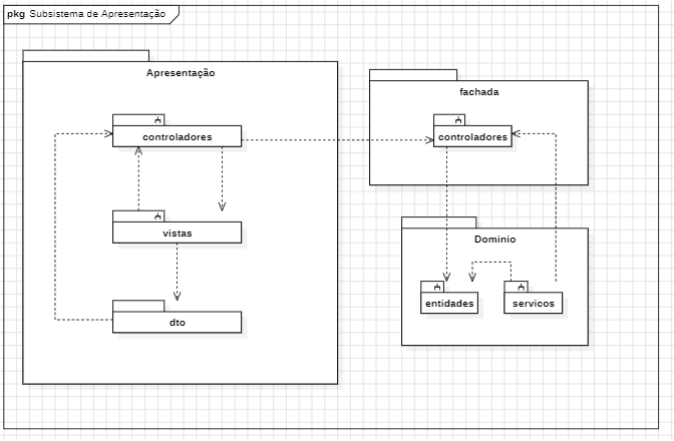

class Mecanismo de Informação de bandas
Este mecanismo permite a obtenção de informação de bandas através da sua pesquisa pelo seu nome
ControladorSistema
+obterInfoPlaylist(utilizador: String, identificador: Id): InfoPlaylistDTO[0...*]
+obterInfoMusica(): InfoMusicaDTO[1...*]
+obterInfoBanda(): InfoBandaDTO[1...*]
+obterInfoArtista(): InfoArtistaDTO[1...*]
+obterAlbunsBanda(nomeBandaAssociada: String): InfoAlbumDTO[0...*]
+obterSubscricao(): InfoSubscricao[1...*]
ControladorSistema
+obterInfoPlaylist(utilizador: String, identificador: Id): InfoPlaylist[0...*]
+obterInfoMusica(): InfoMusica[1...*]
+obterInfoBanda(): InfoBanda[1...*]
+obterInfoArtista(): InfoArtista[1...*]
-servicoSistema
ServicoBanda
+obterBanda(nomeBanda): String
+comprarAlbum(info: InfoUtilizadorDetalhada, album: String): void
«interface»
InfoBandas
Canal de comunicação entre camada de apresentação e domínio
-repBandas
«interface»
IRepositorioBandas
+obterBanda(nomeBanda): String

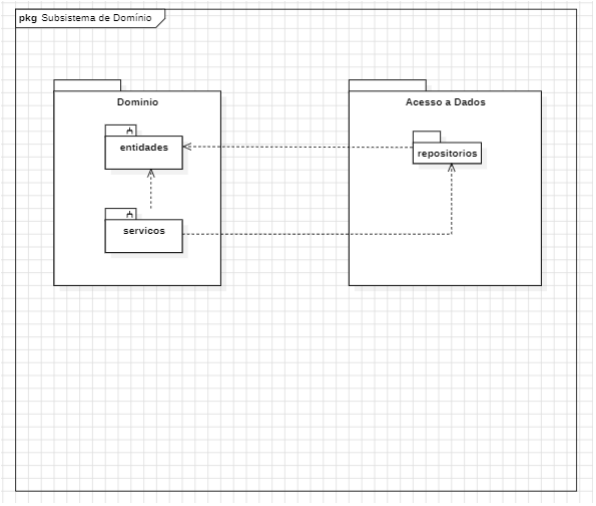

class Mecanismo de Pesquisa
ControladorMusica
-musicas: InfoMusicasDTO[0...*]
+obterMusicas(infoMusicas: infoMusicaDTO): infoMusicasDTO[0...*]
ControladorMusica
+obterMusica(infoMusica: infoMusica): InfoMusica[1...*]
InfoMusica
-idMusica: Id
-nomeMusica: String
-inicioTempoMusica: Time
-fimTempoMusica: Time
-nomeBandaAssociada: String
-nomeArtistaAssociado: String
-servicoMusica
ServicoMusica
+obterMusica(infoMusica: infoMusica): InfoMusica[1...*]
Canal de comunicação entre camada de apresentação e domínio
-repVoos
«interface»
IRepositorioMusica
+obterMusica(infoMusica: infoMusica): InfoMusica[1...*]
RepositorioMusica
-apiSpotify
+RepositorioMusica(apiMusica)

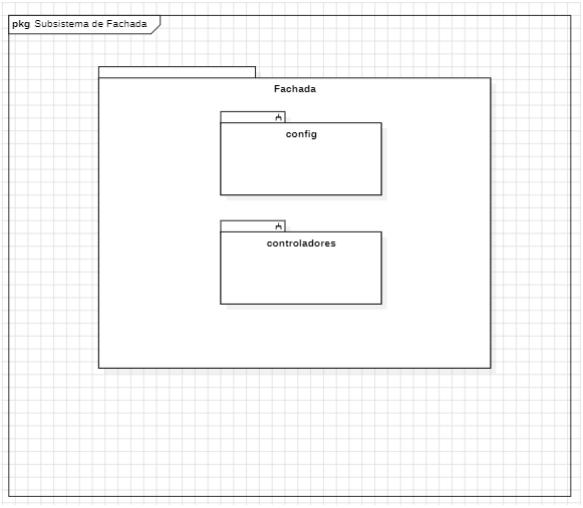

class Mecanismo de Playlists
Este mecanismo permite ao utilizador gerir as suas playlists
ControladorSistema
+obterInfoPlaylist(utilizador: String, identificador: Id): InfoPlaylistDTO[0...*]
+obterInfoMusica(): InfoMusicaDTO[1...*]
+obterInfoBanda(): InfoBandaDTO[1...*]
+obterInfoArtista(): InfoArtistaDTO[1...*]
+obterAlbunsBanda(nomeBandaAssociada: String): InfoAlbumDTO[0...*]
+obterSubscricao(): InfoSubscricao[1...*]
ControladorSistema
+obterInfoPlaylist(utilizador: String, identificador: Id): InfoPlaylist[0...*]
+obterInfoMusica(): InfoMusica[1...*]
+obterInfoBanda(): InfoBanda[1...*]
+obterInfoArtista(): InfoArtista[1...*]
«interface»
Musica
-servicoSistema
ServicoPlaylist
+obterPlaylists(nomeUtilizador: String, idUser: Id): Playlist[0...*]
«interface»
Playlist
Canal de comunicação entre camada de apresentação e domínio
-repPlaylist
«interface»
IRepositorioPlaylists
+obterInfoMusica(): InfoMusica[1...*]
+obterPlaylists(nomeUtilizador: String, idUser: Id): Playlist[0...*]
+adicionarPlaylist(idPlaylist: Id): Playlist[0...*]
+editarPlaylist(idPlaylist: Id): Playlist[0...*]
+removerPlaylist(idPlaylist: Id): Playlist[0...*]

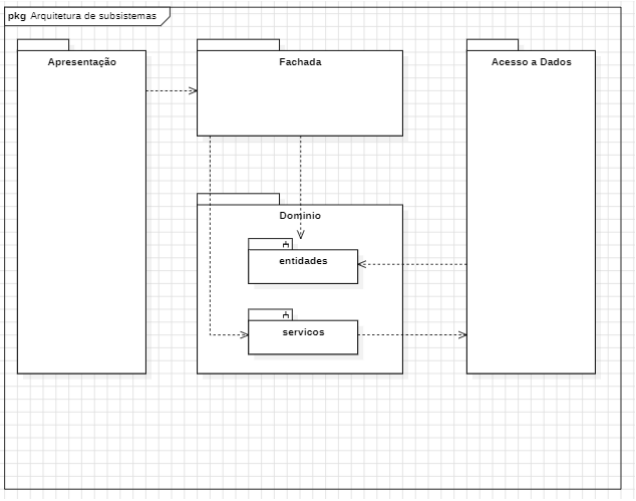
9.2.3 Arquitetura geral da solução






pkg Subsistema de Domínio
Dominio
entidades
servicos
Acesso a Dados
repositorios


pkg Subsistema de Fachada
Fachada
config
controladores



9.2.4 Modelo de dinâmica

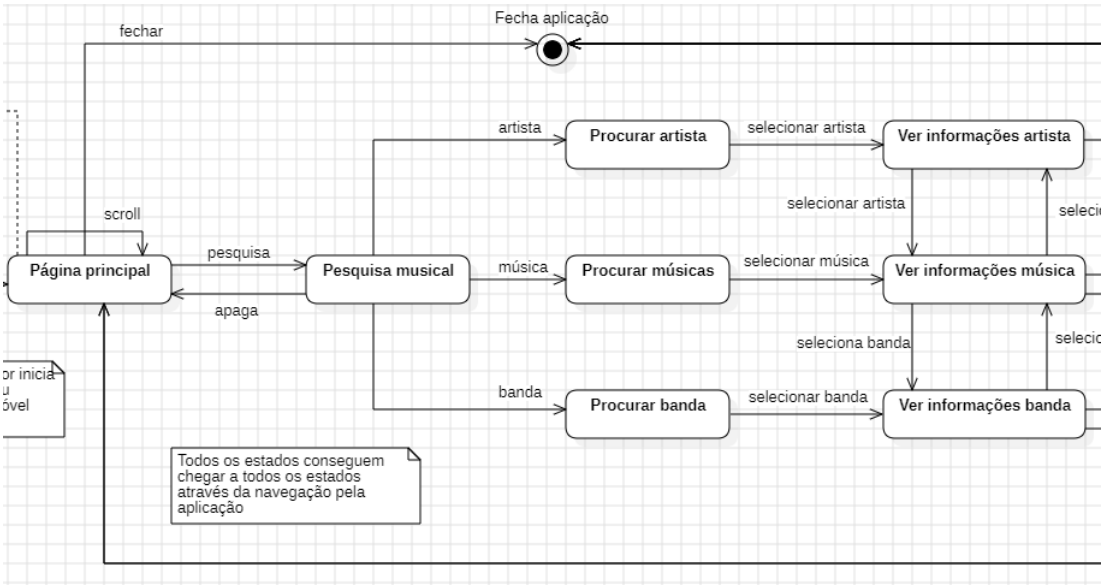
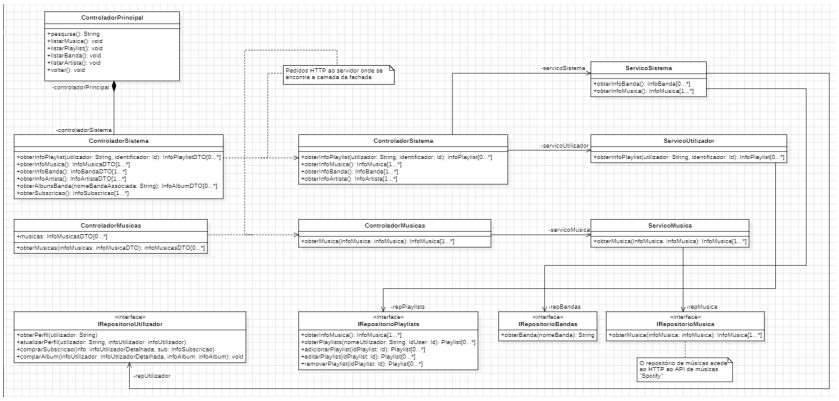


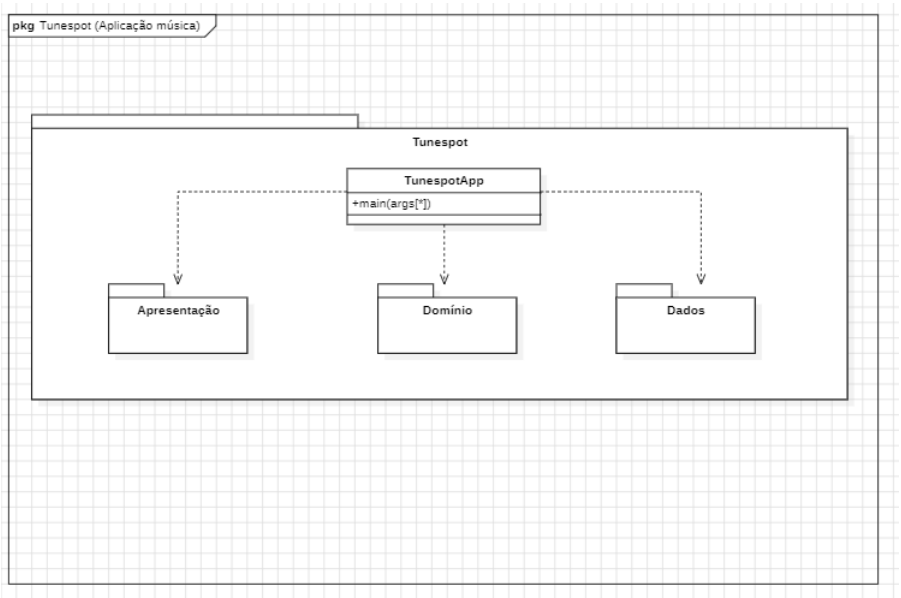
Figure 22: Diagrama do modelo de dinâmica

9.3 Arquitetura detalhada

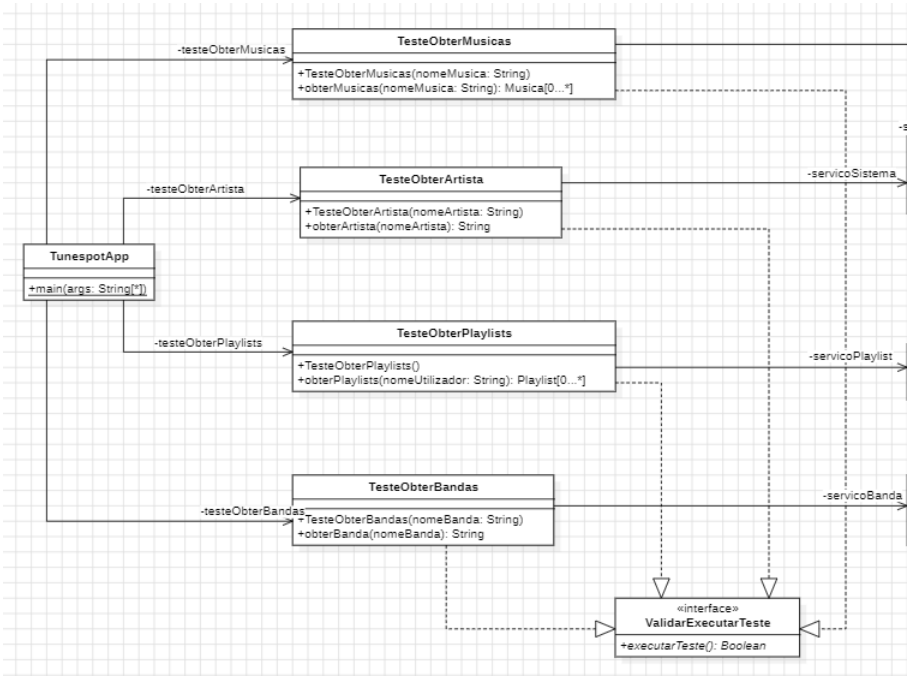
9.3.1 Detalhe de partes e mecanismos

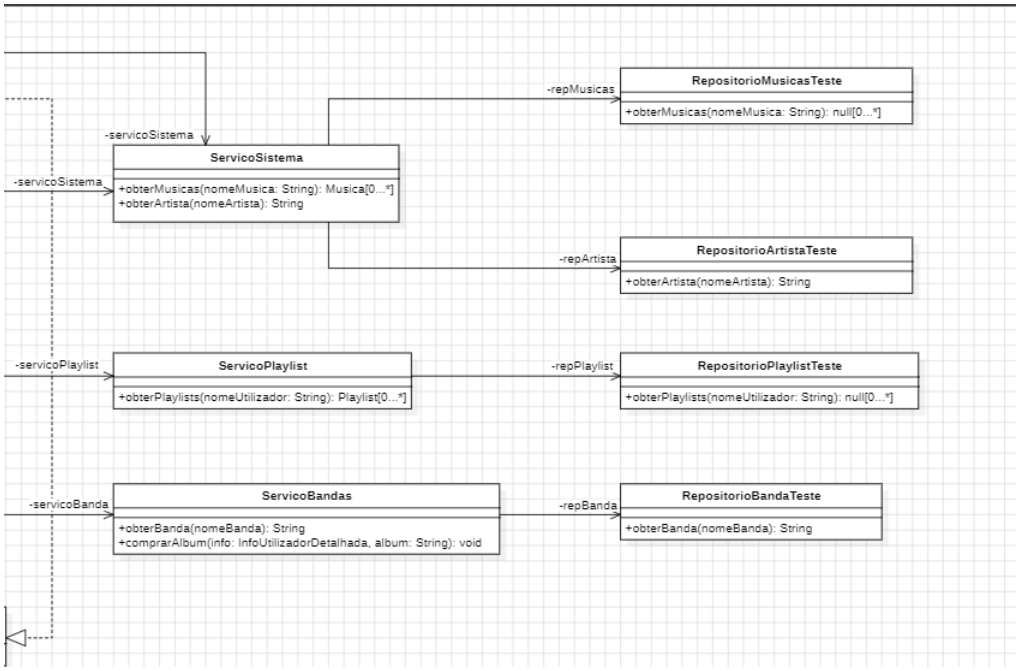


9.3.2 Modelo de implementação



9.3.3 Arquitetura de Teste





9.3.4 Modelo de implantação