UniTunes

Integrantes:

Cristian Hippler, Diego dos Santos, Lucas Galhardo, Nícolas Nunes

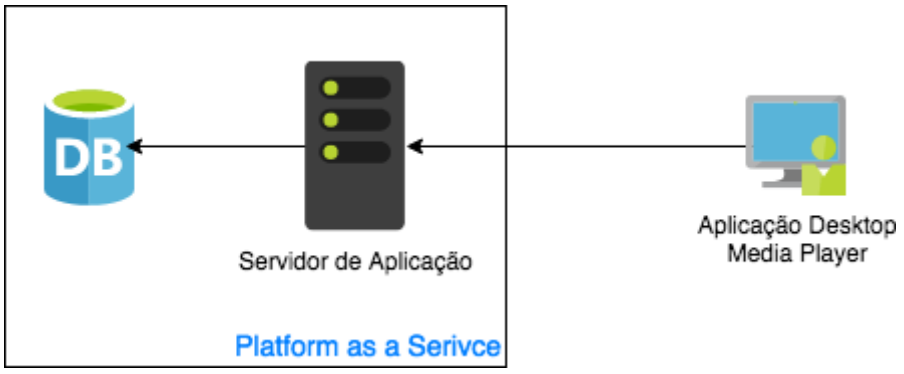
Introdução do projeto

UniTunes é um software de gerenciamento de mídias voltado para a comunidade acadêmica universitária.

Descrição da arquitetura

UniTunes será desenvolvido usando uma arquitetura cliente/servidor. O cliente é uma aplicação desktop composta pelo reprodutor e gerenciador de mídia. O servidor fica responsável pelo armazenamento e streaming dos conteúdos publicados, autenticação e autorização de usuários, e o gerenciamento de venda/compra de conteúdos.

O servidor ficará hospedado em um *Platform as a Service*, desta forma, é possível escalar aplicação em picos de alta demanda e providenciar alta disponibilidade. Para a persistência de dados, será utilizado um banco de dados relacional para informações de usuários, permissões e gerenciamento de conteúdo e armazenamento de mídias. O cliente sempre irá consumir a mídia a partir do servidor sem possibilidade de manter uma cópia, portanto o cliente apenas será funcional se estiver conectado ao servidor.



Padrões de projeto utilizados

REPRODUÇÃO DE MÍDIAS

Para implementar o componente de reprodução de mídias, decidiu-se usar dois padrões de projeto, Strategy e Singleton, que são descritos a seguir.

Strategy

Na aplicação UniTunes, pretende-se suportar a reprodução de mídias de áudio e vídeo. Para cada um dos tipos de mídia, temos diferentes formatos de arquivo, como MP3 e WMA para áudio e MP4, MOV e WMV para vídeo. Para cada um destes formatos, o algoritmo de reprodução muda. Portanto, para facilitar a

manutenção e adição de formatos de mídia suportados, e também para facilitar a escolha do algoritmo de reprodução apropriado durante a execução do programa, decidiu-se utilizar o padrão de projeto Strategy.

Temos então, uma classe MediaPlayer, que age como a classe de contexto, fornecendo o método *play(m : PlayableMedia)* para permitir que quaisquer clientes reproduzam uma mídia. *PlayableMedia* é uma interface que define os métodos *getMediaFile() : byte[]* e *getExtension() : char[]*, que busca as informações do servidor para executar uma mídia.

O método *play(m : PlayableMedia)* da classe MediaPlayer pode verificar a extensão da mídia que se deseja reproduzir e então utilizar o algoritmo adequado.

Para a definição do algoritmo de reprodução, temos a interface *MediaPlayerAlgorithm,* que define o método *play(m : PlayableMedia).* Esta classe representa a *Strategy* do padrão. As classes que vão implementar o algoritmo de reprodução de fato são *as classes MP3Player, MP4Player* e *WMVPlayer,* que realizam a interface *MediaPlayerAlgorithm* e representam as *ConcreteStrategy* do padrão.

Singleton

Como não queremos que mais de uma música seja reproduzida ao mesmo tempo, é interessante que tenhamos somente uma instância da classe *MediaPlayer.* Para atingir isso facilmente, decidimos usar o padrão *Singleton,* que provê um ponto central de acesso à instância de uma classe. Para implementar este padrão, a classe *MediaPlayer* armazena uma instância de si mesma e fornece o método *getInstance()*. Quando o método é chamado pela primeira vez, uma instância da classe é criada e armazenada em uma variável. Nas chamadas subsequentes, a instância é simplesmente retornada.

PAGAMENTO

Para o componente de pagamentos, utilizamos o padrão Strategy para implementar os diferentes métodos de pagamento, para permitir a fácil manutenção e adição de variações dos métodos de pagamento.

Temos uma interface Strategy chamada *MetodoPagamento* que define o método *efetuarPagamento(c : PagamentoController) : double.* As classes *ConcreteStrategy* realizam esta interface, implementando o método *efetuarPagamento* de acordo com as especificações de cada modo de pagamento. Estas classes são: *PagamentoBoletoBancario, PagamentoCartaoCredito* e *PagamentoTransferencia.* Como classe de contexto, temos a classe *PagamentoController,* que permite que clientes definem o método de pagamento desejado com o método *setMetodoPagamento(mp : MetodoPagamento)* e chamem o comportamento *efetuarPagamento.*

A classe *PagamentoController* também permite armazenar atributos necessários para os diferentes métodos de pagamento e permite que os algoritmos de pagamento recuperem esses atributos para processar o pagamento, pois é passada como parâmetro para o método *efetuarPagamento,* implementado pelas classes *ConcreteStrategy.*

LISTA DE FAVORITOS

Para o componente da lista de favoritos, utilizamos o padrão comportamental Iterator para implementar a navegação na lista de mídias. Termos uma interface Aggregate que define o método *getIterator(): Iterator*, e uma interface Iterator responsável por acessar e navegar pelos elementos da lista. A

classe *MidiaList* realiza a interface Aggregate, implementando o método *getIterator(): Iterator*, assim fazendo o papel do ConcreteAggregate. A classe *MidiaIterator* realiza a interface Iterator, implementando os métodos responsáveis pela navegação na lista: *next(): Object, prev(): Object* e *current(): Object.*

PLAYER

O player será elaborado de forma que possa gerenciar todas as funcionalidades enquanto uma música é reproduzida. O reprodutor está separado em algumas instâncias, cada uma denominada e setada para cada tipo de mídia que está sendo reproduzida, como MP3, MP4 e WMV. Através da classe *MediaPlayer,* podemos encontrar os métodos responsáveis pelo gerenciamento das músicas que estão armazenadas na PlayList: *play(): Object, pause(): Object, next(): Object, previus(): Object* e *enqueue(): Object.*

Tecnologias Utilizadas

Springboot framework

O Spring Boot é um projeto da Spring que veio para facilitar o processo de configuração e publicação de nossas aplicações. A intenção é ter o seu projeto rodando o mais rápido possível e sem complicação. Ele consegue isso favorecendo a convenção sobre a configuração. Basta que você diga pra ele quais módulos deseja utilizar (WEB, Template, Persistência, Segurança, etc.) que ele vai reconhecer e configurar. Você escolhe os módulos que deseja através dos starters que inclui no pom.xml do seu projeto.

Apesar do Spring Boot, através da convenção, já deixar tudo configurado, nada impede que você crie as suas customizações caso sejam necessárias. O maior benefício do Spring Boot é que ele nos deixa mais livres para pensarmos nas regras de negócio da nossa aplicação.

Rest API

É frequentemente aplicado à web services fornecendo APIs para acesso a um serviço qualquer na web. Ele usa integralmente as mensagens HTTP para se comunicar através do que já é definido no protocolo sem precisar "inventar" novos protocolos específicos para aquela aplicação.

Você trabalha essencialmente com componentes, conectores e dados.

- Ele usa o protocolo HTTP (verbos, accept headers, códigos de estado HTTP, Content-Type) de forma explícita e representativa para se comunicar. URIs são usados para expor a estrutura do serviço. Utiliza uma notação comum para transferência de dados como XML ou JSON.
- Não possui estado entre essas comunicações, ou seja, cada comunicação é independente e uniforme (padronizada) precisando passar toda informação necessária.
- Ele deve facilitar o cache de conteúdo no cliente.
- Deve ter clara definição do que faz parte do cliente e do servidor. O cliente não precisa saber como o servidor armazena dados, por exemplo. Assim cada implementação não depende da outra e se torna mais escalável.
- Permite o uso em camadas também facilitando a escalabilidade, confiabilidade e segurança.

- Frequentemente é criado com alguma forma de extensibilidade.

Banco H2

O H2 é um banco de dados Open Source que funciona em memória com um console acessível pelo browser dentro do contexto da aplicação. Como ele funciona em memória, todo seu armazenamento é volátil, ou seja, a cada sobe e desce da aplicação ele será reconstruído. Seu intuito é ser um banco de configuração rápido e fácil, visando favorecer a produtividade. O H2 foi escolhido pela fácil configuração em projetos Spring Boot, que também utilizamos no projeto.