Design Patterns e Domain-Driven Design (DDD) com Java [24E2_2]

ASSESSMENT

Frederico Flores

https://github.com/nagualcode/musicfun

INDICE

1. Introdução

O sistema MusicFu com RestAPI foi desenvolvido em Java, com utilização do framework Spring Boot. O objetivo principal do sistema é gerenciar usuários, suas playlists e assinaturas de maneira eficiente e segura. Este documento descreve detalhadamente as funcionalidades, endpoints e serviços, utilizando exemplos reais do código-fonte para ilustrar seu funcionamento.

2. Funcionalidades do Sistema

As principais funcionalidades de MusicFun são:

- **Gestão de Playlists**: Usuários podem criar, editar e deletar playlists, adicionando músicas conforme desejarem.
- **Transações e Pagamentos**: Implementação de um sistema robusto para gerenciar transações e assinaturas, garantindo segurança e eficiência.
- **Controle de Acesso**: Sistema de autenticação e autorização para gerenciamento seguro dos acessos aos diferentes tipos de usuários.
- **Prevenção a Fraudes**: Integração de um serviço de anti-fraude para validar e assegurar as transações realizadas.

3. Estrutura do Projeto

O projeto segue uma estrutura modular clara, facilitando o desenvolvimento e a manutenção:

- **domain**: Classes de entidades e repositórios.
- **service**: Lógica de negócios e serviços de aplicação.
- **controller**: Controladores que respondem às interações do usuário e redirecionam para as vistas apropriadas.
- **config**: Configurações globais da aplicação, incluindo configurações de segurança.
- **exception**: Tratamento de exceções específicas da aplicação.

4. Domínios

Os domínios no projeto MusicFun representam diferentes aspectos funcionais da aplicação, organizados em módulos para facilitar o gerenciamento e a manutenção. Aqui estão os domínios principais com suas respectivas funcionalidades e estruturas:

4.1. Usuário (User)

- DTOs: Contém UserDTO e RoleDTO, responsáveis pelo transporte de dados do usuário e suas permissões entre as camadas.
- Repositórios: UserRepository e RoleRepository para operações de banco de dados relacionadas aos usuários e suas roles.
- Controladores: UserController e RoleController gerenciam as requisições HTTP relacionadas aos usuários e roles.
- Modelos: AppUser, Role e AppUserDetails representam a estrutura de dados dos usuários, suas permissões e os detalhes usados para autenticação.
- Serviços: UserService e RoleService encapsulam a lógica de negócios associada aos usuários e suas roles.

4.2. Playlist

- DTOs: Inclui MusicDTO e PlaylistDTO para a transferência de dados de músicas e playlists.
- Repositórios: MusicRepository e PlaylistRepository para manipulação de dados de músicas e playlists.
- Controladores: MusicController e PlaylistController lidam com as requisições relacionadas à criação e gestão de playlists e músicas.
- Modelos: Music e Playlist são as entidades que representam as músicas individuais e as coleções de músicas.
- Serviços: MusicService e PlaylistService gerenciam as operações de negócios para músicas e playlists.

4.3. Pagamento (Payment)

- DTOs: Contém TransactionDTO e SubscriptionDTO para transporte de dados de transações e assinaturas.
- Repositórios: TransactionRepository e SubscriptionRepository gerenciam as interações com o banco de dados para transações e assinaturas.
- Controladores: TransactionController e SubscriptionController controlam as requisições HTTP para gerenciamento de transações e assinaturas.

Modelos: Transaction e Subscription representam as informações de transações financeiras e assinaturas de serviços, respectivamente.
Serviços: TransactionService, SubscriptionService e AntiFraudService tratam da lógica de negócios envolvendo pagamentos, assinaturas e a prevenção de fraudes.

4.4. Configuração (Config)

Contém classes como **SecurityConfig**, **WebConfig** e **AppConfig** que configuram aspectos centrais da aplicação, incluindo segurança, configurações web e configurações gerais da aplicação.

4.5 Organização dos arquivos

6. Endpoints

Os endpoints do sistema cobrem funcionalidades de CRUD (criar, ler, atualizar e deletar) para as entidades principais do sistema e oferecem funcionalidades específicas como a manipulação de playlists por **usuários autenticados** .

6.1 SubscriptionController (/subscriptions)

- GET /subscriptions: Lista todas as assinaturas. Retorna um array de SubscriptionDTO com todas as assinaturas disponíveis.
- GET /subscriptions/{id}: Recupera uma assinatura específica pelo ID. Retorna SubscriptionDTO da assinatura correspondente ou uma resposta de não encontrado se o ID não existir.
- POST /subscriptions: Cria uma nova assinatura. Recebe um SubscriptionDTO e retorna o SubscriptionDTO da assinatura criada com status 201.
- PUT /subscriptions/{id}: Atualiza uma assinatura existente com base no ID. Recebe um SubscriptionDTO e retorna o SubscriptionDTO atualizado.
- DELETE /subscriptions/{id}: Exclui uma assinatura com base no ID. Retorna uma resposta sem conteúdo se a exclusão for bem-sucedida.

6.2 TransactionController (/transactions)

- GET /transactions: Lista todas as transações. Retorna um array de TransactionDTO com todas as transações disponíveis.
- GET /transactions/{id}: Recupera uma transação específica pelo ID. Retorna TransactionDTO da transação correspondente.
- POST /transactions: Cria uma nova transação. Recebe um TransactionDTO e retorna o TransactionDTO da transação criada com status 201.
- PUT /transactions/{id}: Atualiza uma transação existente com base no ID. Recebe um TransactionDTO e retorna o TransactionDTO atualizado.
- DELETE /transactions/{id}: Exclui uma transação com base no ID. Retorna uma resposta sem conteúdo se a exclusão for bem-sucedida.

6.3 MusicController (/music)

- GET /music: Lista todas as músicas. Retorna um array de MusicDTO.
- GET /music/{id}: Recupera uma música específica pelo ID. Retorna MusicDTO da música correspondente.
- POST /music: Cria uma nova música. Recebe um MusicDTO e retorna o MusicDTO da música criada.
- PUT /music/{id}: Atualiza uma música existente com base no ID. Recebe um MusicDTO e retorna o MusicDTO atualizado.
- DELETE /music/{id}: Exclui uma música com base no ID.

6.4 PlaylistController (/playlists)

- GET /playlists: Lista todas as playlists. Retorna um array de PlaylistDTO.
- GET /playlists/{id}: Recupera uma playlist específica pelo ID. Retorna PlaylistDTO da playlist correspondente.
- GET /playlists/user: Lista playlists do usuário autenticado.
- GET /playlists/user/{username}: Lista playlists de um usuário específico pelo nome de usuário.
- GET /playlists/search: Busca playlists pelo nome.
- POST /playlists: Cria ou atualiza uma playlist. Recebe um PlaylistDTO e retorna o PlaylistDTO da playlist criada ou atualizada.
- PUT /playlists/{id}: Atualiza uma playlist existente com base no ID. Recebe um PlaylistDTO e retorna o PlaylistDTO atualizado.
- DELETE /playlists/{id}: Exclui uma playlist com base no ID.
- POST /playlists/user/favorites: Adiciona músicas à playlist de favoritos do usuário autenticado.

6.5 UserController (/users)

- GET /users: Lista todos os usuários. Retorna um array de UserDTO com todos os usuários cadastrados.
- GET /users/{id}: Recupera um usuário específico pelo ID. Retorna UserDTO do usuário correspondente ou lança uma exceção se o usuário não for encontrado.
- POST /users: Cria um novo usuário. Recebe um UserDTO e retorna o UserDTO do usuário criado com o ID atribuído.
- PUT /users/{id}: Atualiza um usuário existente com base no ID fornecido. Recebe um UserDTO e retorna o UserDTO atualizado após a modificação dos dados.
- DELETE /users/{id}: Exclui um usuário com base no ID. Executa a exclusão do usuário no banco de dados e não retorna conteúdo na resposta

7. Testes Unitários

Os testes unitários do projeto são implementados com chamadas de API simuladas usando o framework **MockMvc**, o que permite uma avaliação detalhada das respostas do sistema sob diversas condições, garantindo que as funcionalidades mais críticas do sistema sejam testadas de maneira segura e eficaz, assim atingindo uma cobertura robusta das funcionalidades essenciais do sistema, testando tanto os aspectos de segurança quanto as principais operações dos serviços:

7.1 Testes do AntiFraudService:

- Teste de transações de alta frequência: Verifica se o sistema pode identificar e rejeitar uma transação quando a frequência é excessivamente alta, simulando um possível cenário de fraude .
- Teste de transações duplicadas: Testa a capacidade do sistema de detectar e rejeitar transações duplicadas, um cenário comum em tentativas de fraude .
- Teste de transações de alto valor: Avalia se o sistema pode rejeitar transações que excedam um certo limite de valor, que poderia indicar uma tentativa de movimentação suspeita de fundos .

7.2 Testes do UserController:

- Teste de busca de todos os usuários: Verifica se todos os usuários são listados corretamente, assegurando que o acesso aos dados do usuário está funcionando como esperado .
- Teste de criação de usuário: Testa a funcionalidade de adicionar um novo usuário ao sistema, crucial para a expansão e gerenciamento do usuário .
- Teste de exclusão de usuário: Garante que os usuários possam ser removidos de forma segura, uma operação importante para a manutenção e gestão de contas .

7.3 Testes do SubscriptionController:

- Teste de criação de assinatura: Confirma que o sistema pode processar novas assinaturas corretamente, uma parte vital do gerenciamento de assinaturas do usuário .
- Teste de listagem de todas as assinaturas: Assegura que todas as assinaturas sejam listadas corretamente, garantindo que a visibilidade dos dados da assinatura esteja correta .

7.4 Testes do PlaylistController:

- Teste de adição de música às favoritas do usuário: Verifica se a funcionalidade de adicionar músicas à playlist de favoritos do usuário funciona corretamente, um aspecto essencial da personalização da experiência do usuário .
- Teste de atualização de playlist: Testa a atualização das playlists, uma funcionalidade importante para manter a relevância e a precisão do conteúdo disponível para os usuários .

8. Exemplos práticos

8.1 Criação de Conta

Para criar uma nova conta de usuário, o sistema utiliza o método POST no controlador UserController. Aqui está um exemplo do código necessário para criar um novo usuário:

```
@PostMapping("/users")
public ResponseEntity<UserDTO> createUser(@RequestBody UserDTO userDTO) {
    UserDTO createdUser = userService.save(userDTO);
    return new ResponseEntity<>(createdUser, HttpStatus.CREATED);
}
```

8.2. Autorização de Transação

A autorização de transações é gerenciada pelo TransactionService, que valida transações usando o AntiFraudService. Aqui está um exemplo de como uma transação é criada e validada:

```
@PostMapping("/transactions")
public ResponseEntity<TransactionDTO> createTransaction(@RequestBody
TransactionDTO transactionDTO) {
    Transaction transaction = convertToEntity(transactionDTO);
    if (!antiFraudService.validateTransaction(transaction)) {
        throw new IllegalArgumentException("Transação suspeita de
fraude.");
    }
    TransactionDTO createdTransaction =
convertToDTO(transactionService.save(transaction));
    return new ResponseEntity<>(createdTransaction, HttpStatus.CREATED);
}
```

8.3. Favoritar Músicas

Para favoritar músicas, o sistema utiliza um endpoint específico no PlaylistController que permite ao usuário adicionar músicas a sua playlist de favoritos:

```
@PostMapping("/playlists/user/favorites")
public ResponseEntity<PlaylistDTO> addMusicToFavorites(@RequestBody
List<MusicDTO> musicDTOs, @AuthenticationPrincipal User user) {
    Playlist favorites = playlistService.addFavorites(user.getId(),
musicDTOs);
    return new ResponseEntity<>(convertToPlaylistDTO(favorites),
HttpStatus.OK);
}
```

8.4. Gerenciamento de Playlists

O gerenciamento de playlists envolve a criação, atualização e exclusão de playlists. Aqui está como uma playlist é criada:

```
@PostMapping("/playlists")
public ResponseEntity<PlaylistDTO> createPlaylist(@RequestBody PlaylistDTO
playlistDTO) {
    Playlist playlist = convertToEntity(playlistDTO);
    Playlist savedPlaylist = playlistService.saveOrUpdate(playlist);
    return new ResponseEntity<>(convertToPlaylistDTO(savedPlaylist),
HttpStatus.CREATED);
}
```

8.5. Assinatura

A criação de uma assinatura é feita por meio do endpoint POST no SubscriptionController, que salva uma nova assinatura no sistema:

```
@PostMapping("/subscriptions")
public ResponseEntity<SubscriptionDTO> createSubscription(@RequestBody
SubscriptionDTO subscriptionDTO) {
    Subscription subscription = convertToEntity(subscriptionDTO);
    Subscription savedSubscription =
subscriptionService.save(subscription);
    return new ResponseEntity<>(convertToDTO(savedSubscription),
HttpStatus.CREATED);
}
```

9. Princípios de Design

9.1. Interfaces e Assinaturas de Métodos Reveladoras de Intenções

O projeto adota interfaces bem definidas que revelam as intenções de seus métodos, conforme proposto por Eric Evans em "Domain-Driven Design". Por exemplo, a interface **SubscriptionRepository** estende **BaseRepository**, e os métodos específicos como **save** e **findById** deixam claro o que realizam .

9.2. Princípios S.O.L.I.D.

O código demonstra a adesão aos princípios S.O.L.I.D:

- Single Responsibility Principle: Cada classe no projeto tem uma única responsabilidade. Por exemplo, UserService lida apenas com operações relacionadas a usuários .
- Open/Closed Principle: Classes como SubscriptionService são exemplos de módulos que estão abertos para extensão, mas fechados para modificação. Novas funcionalidades podem ser adicionadas sem alterar o código existente.
- Liskov Substitution Principle: As interfaces são utilizadas de maneira que permite que objetos de classes derivadas possam substituir objetos de classes base sem comprometer a funcionalidade, como é visto com a implementação de JpaRepository em SubscriptionRepository .
- Interface Segregation Principle: O projeto utiliza interfaces pequenas e específicas para os repositórios, que são implementadas pelos serviços de forma a não sobrecarregar as classes implementadoras com métodos desnecessários.
- Dependency Inversion Principle: Há uma clara separação e inversão de dependências, como visto no uso de injeção de dependências (@Autowired) nos serviços, como SubscriptionService, que dependem de abstrações (interfaces de repositórios) e não de detalhes concretos .

9.3. Utilização do Domain-Driven Design (DDD)

- Modelagem de Domínios Ricos: As entidades como Subscription e Transaction são ricas em comportamento, contendo lógicas de negócios significativas, como validação de fraudes e manipulação de estados de transações .
- Domain Services: Os serviços como TransactionService e SubscriptionService encapsulam lógicas de negócios complexas que transcendem uma única entidade, demonstrando um bom exemplo de serviços de domínio .
- Aggregates: A entidade Subscription pode ser vista como um Aggregate que mantém a consistência das transações relacionadas às assinaturas .
- Context Mapping: Embora o código específico para context mapping não seja explicitamente mencionado, a organização do código sugere a separação contextual, com diferentes partes do domínio como user, payment, e playlist, operando de maneira semi-independente mas integrada .

10. Conclusão

Este projeto exemplifica uma aplicação eficaz de princípios de engenharia de software moderna, integrando metodologias de Domain-Driven Design (DDD) e princípios S.O.L.I.D. para criar uma base robusta e escalável para um sistema de gerenciamento de música.

As entidades do domínio são modeladas com comportamentos ricos, encapsulando lógicas de negócio vitais, o que demonstra uma compreensão profunda dos requisitos do domínio. Os serviços de domínio, juntamente com os repositórios bem definidos, asseguram que as operações críticas sejam gerenciadas de forma coesa e desacoplada, promovendo uma arquitetura limpa e modular.

O projeto implementa interfaces claras e intencionais que suportam a extensibilidade e a substituição de componentes, alinhadas com os princípios de substituição de Liskov e segregação de interface. Isso é particularmente evidente na forma como as dependências são gerenciadas e injetadas, refletindo uma aplicação cuidadosa do princípio de inversão de dependência.

Em termos de práticas de desenvolvimento, o uso extensivo de testes automatizados demonstra um comprometimento com a qualidade e a robustez do software. Os testes cobrem funcionalidades chave, desde a gestão de usuários até operações transacionais complexas, assegurando que o sistema não apenas atenda às expectativas iniciais, mas também mantenha sua integridade frente a modificações e expansões.

https://github.com/nagualcode/musicfun

22/Junho/2024.