Relatório Técnico - LPS 3

Grupo: Pedro Negri Leão Lambert, Pedro Henrique Pires, Vinícius Rezende
Grupo do projeto analisado: Jhonatan Gutemberg Rosa Ferreira, Gabriel Henrique Miranda Rodrigues,André Cota Guimarães e Lívia Carolina de Souza Lima

Tecnologias e Arquitetura do Sistema

Tecnologias Utilizadas

O sistema foi desenvolvido utilizando uma stack composta por:

- **Java**
- **Spring Boot**
- **Swagger**
- **Maven**: Ferramenta de build utilizada para gerenciamento de dependências e automação de tarefas de construção do projeto.
- **PostgreSQL**
- **Spring Security**: Responsável pela implementação das camadas de segurança, incluindo autenticação e autorização.
- **Lombok**: Biblioteca que reduz a verbosidade do código Java ao gerar automaticamente métodos como getters, setters e construtores.
- **Serviço de e-mail com Spring**: Integração para envio de e-mails no sistema, utilizando os recursos nativos do framework.
- **JWT (JSON Web Tokens)**

Arquitetura do Sistema

O sistema foi projetado com uma **arquitetura em camadas**, adaptando o padrão **MVC (Model-View-Controller)** para as necessidades específicas do projeto. A arquitetura garante modularidade, separação de responsabilidades e facilidade de manutenção.

Camadas do Sistema

1. **Controller**
    - Responsável pela interface com o usuário ou consumidores da API.
    - Depende exclusivamente das **interfaces da camada de serviço** e dos **DTOs** (Data Transfer Objects).
    - Atua como intermediária, recebendo requisições, delegando a lógica para os serviços e retornando as respostas.
2. **Serviço**
    - Encapsula toda a lógica de negócio do sistema.
    - Depende das **interfaces de repositório**, dos **DTOs** e dos **Models**.
    - A lógica é implementada nesta camada, promovendo uma separação clara entre os controladores e a persistência.

3. **Repositório**

    - Gerenciado por interfaces implementadas pelo **Spring Data JPA**.
    - Permite a abstração e automação de operações com o banco de dados, aproveitando um framework amplamente testado para reduzir o esforço de desenvolvimento.
    - Suporta consultas dinâmicas e personalizadas, garantindo flexibilidade e eficiência.
4. **Modelos (Models)**

    - Representam o domínio do problema, mapeando entidades do mundo real para classes Java.
    - Incluem atributos, relacionamentos e definições de enumeradores (enums) que ajudam na consistência do sistema.

Papéis dos DTOs

Os **DTOs (Data Transfer Objects)** são utilizados para transportar dados entre as camadas do sistema. São autocontidos e dependem apenas de alguns dos enumeradores, garantindo que os dados estejam organizados de forma eficiente e desacoplados de detalhes específicos de implementação.

Inversão de Dependências

A arquitetura adota o princípio da inversão de dependências, onde:

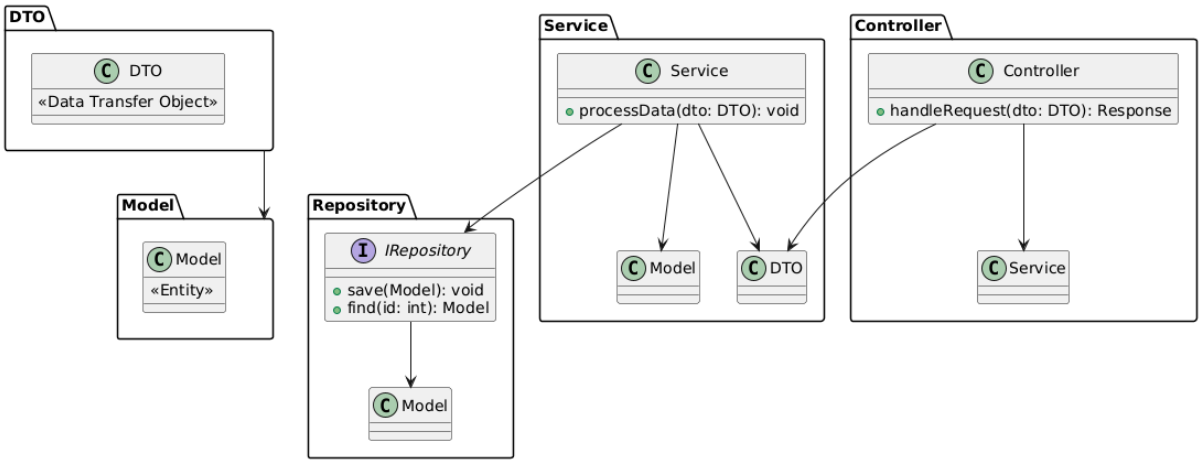
- As **interfaces de serviço e repositório** garantem que os componentes dependam de abstrações em vez de implementações concretas.
- Essa abordagem promove flexibilidade e facilita a troca ou extensão de componentes do sistema.

Classes Auxiliares e de Configuração

O sistema também conta com classes auxiliares e de configuração que dão suporte ao funcionamento geral, incluindo integração com serviços externos, envio de e-mails e configuração de segurança.

Considerações Finais

A arquitetura em camadas, aliada às tecnologias escolhidas, proporciona uma solução bem estruturada, escalável e alinhada às boas práticas de desenvolvimento. A utilização de frameworks amplamente adotados e testados, como o Spring Boot e o JPA, aumenta a confiabilidade do sistema e reduz o tempo de desenvolvimento, enquanto a separação clara de responsabilidades facilita a manutenção e evolução da aplicação.



Organização do Repositório GitHub

A organização do repositório GitHub reflete uma preocupação com a clareza e estruturação do projeto, mas há pontos que podem ser melhorados para fornecer uma visão mais completa e funcional do sistema.

Pontos Positivos

- **Diagramas bem elaborados**: O repositório apresenta diagramas claros, bem feitos e explicativos, que ajudam na compreensão da arquitetura e do funcionamento do sistema.
- **Organização de arquivos**: A disposição dos arquivos e pastas é bem estruturada, seguindo boas práticas de organização de projetos. Isso facilita a navegação e a localização de componentes importantes do sistema.

Dúvidas e Pontos Possivelmente Negativos

- **Ausência de código do front-end**: Não há, no repositório, código referente ao front-end da aplicação. Além disso, o README não menciona a existência de outro repositório dedicado ao front-end. Caso o front-end esteja em um repositório separado, seria importante incluir uma referência clara a ele no README, permitindo que os usuários compreendam o sistema como um todo.

Pontos a Melhorar

1. **Formato dos Casos de Uso no README**:
    - Os casos de uso atualmente listados poderiam ser reformulados em formato de tabela Markdown. Essa estrutura facilitaria a leitura e a compreensão, apresentando os **atores**, **descrições** e **detalhes dos casos de uso** de maneira mais organizada e acessível.
2. **Descrição textual no README**:

- Embora os diagramas sejam bem explicativos, falta uma descrição textual que contextualize o trabalho. Uma introdução ao sistema, com informações sobre o objetivo do projeto, seria uma adição valiosa.
- Além disso, seria interessante incluir menções ao professor responsável e à disciplina relacionada, contextualizando o projeto no âmbito acadêmico.

3. **Informações sobre as tecnologias e setup do projeto**:

    - O README poderia incluir uma seção detalhada sobre as tecnologias utilizadas no desenvolvimento do sistema, destacando a stack tecnológica e seus principais benefícios.
    - Adicionar informações sobre como configurar e executar o projeto localmente (incluindo dependências e instruções de inicialização) seria essencial para novos colaboradores.

4. **Gifs de funcionamento**:

    - Uma seção com GIFs demonstrando o funcionamento da aplicação seria um excelente complemento. Esse recurso oferece uma visualização prática do sistema, tornando-o mais atrativo e intuitivo para potenciais usuários e colaboradores.

Considerações Finais

Apesar de algumas lacunas, o repositório apresenta uma ótima organização e documentações visuais de qualidade. Com pequenos ajustes no README, incluindo referências ao front-end (caso exista), melhorias na formatação e adição de informações contextuais e práticas, o projeto poderia alcançar um padrão ainda mais profissional e funcional.

Dificuldade da Configuração de Ambiente

A configuração do ambiente apresentou um nível de dificuldade moderado. Não foi encontrado um repositório para o front-end, portanto, o foco ficou exclusivamente no back-end. Apesar da ausência de instruções específicas para testes, localizar onde ajustar o usuário para acesso ao banco de dados foi relativamente simples. Além disso, a presença do Swagger facilitou a realização dos testes de funcionamento da API, tornando o processo mais direto e eficiente.

Pull Request e Alterações Sugeridas

As refatorações propostas para o sistema foram realizadas com o objetivo de melhorar a estrutura e consistência do código, alinhando-se aos princípios de boas práticas de desenvolvimento. A premissa foi realizar ajustes no design do software sem alterar seu comportamento externo, mantendo a lógica já implementada intacta. Todas as alterações estão detalhadas no seguinte pull request.

As principais melhorias sugeridas foram:

1. **Adição do `InstitutionEducationResponseDTO`**

   Refatoração dos métodos do `InstitutionEducationController` para utilizar o DTO, padronizando as respostas da API. Com isso, garantiu-se que os controllers não conheçam ou dependam diretamente dos Models. Além disso, foram criadas classes auxiliares de mapeamento para promover a reusabilidade de código e facilitar manutenções futuras.

```
+ package com.coinsystem.system.DTO;
+
+ import com.coinsystem.system.enums.UsersType;
+
+ public record InstitutionEducationResponseDTO(
+     Long id,
+     String name,
+     String email,
+     UsersType type,
+     String phoneNumber,
+     String address,
+     String cnpj
+ ) {
+
+ }
```

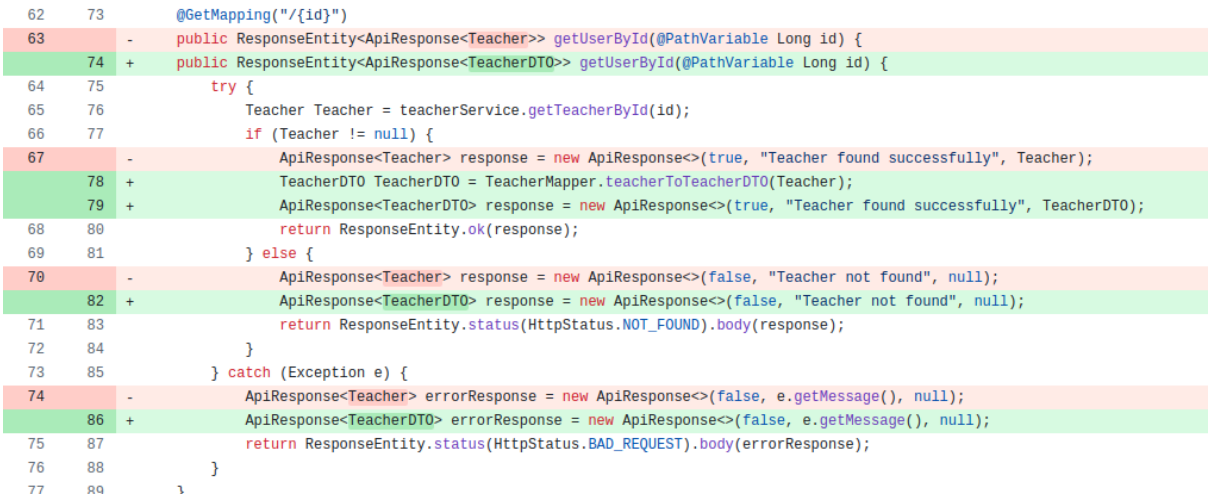
2. **Adição do `PartnerCompanyDTO`**

   Refatoração dos métodos do `PartnerCompanyController` para seguir o mesmo padrão, utilizando o `PartnerCompanyDTO` nas respostas. Assim como no primeiro caso, isso contribui para a consistência do design, reduzindo o acoplamento entre controllers e Models. Também foram criadas classes de mapeamento para gerenciar a transformação entre entidades e DTOs de maneira centralizada.

```
+ package com.coinsystem.system.mappers;
+
+ import com.coinsystem.system.DTO.PartnerCompanyDTO;
+ import com.coinsystem.system.model.PartnerCompany;
+
+ public class PartnerCompanyMapper {
+     public static PartnerCompanyDTO partnerCompanyToPartnerCompanyDTO(PartnerCompany partnerCompany) {
+         return new PartnerCompanyDTO(
+                 partnerCompany.getName(),
+                 partnerCompany.getEmail(),
+                 partnerCompany.getType(),
+                 partnerCompany.getPhoneNumber(),
+                 partnerCompany.getCnpj(),
+                 partnerCompany.getPassword(),
+                 partnerCompany.getAddress()
+         );
+     }
+ }
```

3. **Adição do `TeacherDTO`**

   Refatoração dos métodos do `TeacherController` para utilização do `TeacherDTO`, assegurando a padronização das respostas da API e seguindo os mesmos princípios de desacoplamento e reusabilidade estabelecidos nas refatorações anteriores.



```
62  73      @GetMapping("/{id}")
63      -   public ResponseEntity<ApiResponse<Teacher>> getUserById(@PathVariable Long id) {
    74  +   public ResponseEntity<ApiResponse<TeacherDTO>> getUserById(@PathVariable Long id) {
64  75          try {
65  76              Teacher Teacher = teacherService.getTeacherById(id);
66  77              if (Teacher != null) {
67      -               ApiResponse<Teacher> response = new ApiResponse<>(true, "Teacher found successfully", Teacher);
    78  +               TeacherDTO TeacherDTO = TeacherMapper.teacherToTeacherDTO(Teacher);
    79  +               ApiResponse<TeacherDTO> response = new ApiResponse<>(true, "Teacher found successfully", TeacherDTO);
68  80                  return ResponseEntity.ok(response);
69  81              } else {
70      -               ApiResponse<Teacher> response = new ApiResponse<>(false, "Teacher not found", null);
    82  +               ApiResponse<TeacherDTO> response = new ApiResponse<>(false, "Teacher not found", null);
71  83                  return ResponseEntity.status(HttpStatus.NOT_FOUND).body(response);
72  84              }
73  85          } catch (Exception e) {
74      -           ApiResponse<Teacher> errorResponse = new ApiResponse<>(false, e.getMessage(), null);
    86  +           ApiResponse<TeacherDTO> errorResponse = new ApiResponse<>(false, e.getMessage(), null);
75  87              return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(errorResponse);
76  88          }
77  89      }
```

Notas finais:

As refatorações foram focadas, principalmente, nos controllers devido à natureza do escopo, que não previa alterações na lógica ou funcionalidades do sistema. As camadas internas já apresentavam uma modularidade consistente e bem estruturada, e mudanças nelas poderiam implicar em modificações mais profundas no funcionamento do sistema, o que foge do objetivo deste trabalho.