Documentação do projeto SGEscolar

Tecnologias

Lado Servidor - Backend

Para o lado servidor, foi escolhida a linguagem **Java 11** e framework **Spring Boot 2.6.0**, bem como, banco de dados **PostgreSQL 13**.

O servidor de aplicação java escolhido foi o **Apache Tomcat 9.0.54**

O backend disponibiliza uma **API** (Serviços Web oferecidos) que tem sua especificação documentada e acessível através do **swagger 3.0**.

Lado Cliente - Frontend

Como trata-se de um sistema Web, o frontend envolve: **HTML**, **CSS**, **Javascript**, **Typescript**, **AJAX** e **bootstrap**.

O repositório de código

O sistema de gerenciamento de código utilizado será o **GIT** e, para isso, serão criados dois repositórios no sistema Github. Isto é, um projeto para o código fonte de todo o sistema e outro projeto para o frontend.

Estrutura do projeto

Backend

No backend java, o projeto está estruturado conforme o padrão MVC para Spring Boot. Veja a estrutura abaixo:

/src – Pasta do código fonte

/src/main/java – Pasta dos **pacotes** e **classes** Java, isto é, onde ficam os arquivos com extensão.java

/src/main/resources – Pasta onde fica a configuração do spring. Nesta pasta pode ser encontrado o arquivo **application.yml**.

/src/main/webapp – Pasta onde ficam os arquivos **html, css** e **javascript**. O que significa que, nesta pasta, podem estar os arquivos componentes do frontend da aplicação. O uso desta pasta é útil quando se deseja empacotar toda a aplicação (backend e frontend) em um único arquivo de extensão **.war** ou numa pasta que pode ser colocada em um servidor de aplicação java.

Estrutura de pacotes

Os pacotes java ficam na pasta "/src/main/java", conforme já foi explicado acima. Abaixo a descrição dos principais pacotes e classes do spring e do sistema:

sisescolar – pacote base do projeto. Assuma que os próximos pacotes descritos têm base neste

model – pacote onde ficam as entidades do banco de dados mapeadas pelo spring data. Objetos dessas classes são utilizadas nos pacotes: **repository, service, dao e builder**.

> model.request – pacote onde ficam as classes **DTOs** que mapeiam o corpo das requisições que estão no formato **JSON.** Estas classes são utilizadas nos pacotes: **service, controller, builder**
>
> model.response – pacote onde ficam as classes **DTOs** que mapeiam o corpo das respostas que estão no formato **JSON.** Estas classes são utilizadas nos pacotes: **service, controller, builder**

repository – pacote onde ficam as interfaces que estendem a interface **JpaRepository**. É nas interfaces desses pacotes que são herdados os métodos básicos de acesso aos dados da interface **JpaRepository** para entidade configurada em cada interface. As consultas e outras instruções **SQL** nativo ou **HQL** personalizadas podem ser colocadas em métodos destas interfaces para serem acessadas pelos serviços presentes no pacote service.

dao – pacote onde ficam objetos que agrupam conjuntos de instruções que podem ser acessadas quantas vezes necessário pelas classes do pacote service. **Obs:** Alguns métodos dessas classes lançam exceções que passam pelos services que as utilizam até chegar aos controllers para, então, serem tratadas.

builder – pacote onde ficam os componentes que oferecem métodos para transferência de dados de **DTOs** do pacote model.request e entidades do pacote model, bem como, também, transferência de dados entre entidades do pacote model e **DTOs** do pacote model.response. Essas classes são utilizadas nos serviços que ficam no pacote service. **Obs:** Alguns métodos dessas

classes lançam exceções que passam pelos services que as utilizam até chegar aos controllers para, então, serem tratadas.

service – pacote onde ficam as classes de serviços (não confundir com webservices) que implementam as regras de negócio. São objetos com funções de **BOs** (Business Object). Objetos dessas classes podem utilizar em seus atributos e métodos vários repositories, daos e builders para fornecer os métodos para uso nos controllers. Muitos métodos dessas classes lançam exceções que são capturadas e tratadas nos controllers que as utilizam.

controller – pacote onde ficam os controllers que vinculam seus métodos a endpoints, conforme o protocolo **HTTP** e o formato, quase sempre, como **JSONs**. Essas classes chamam os services e retornam respostas de erro **400** que significa **Bad Request** e um **JSON** com o código e mensagem de erro**,** isso em caso de exceção capturada ou falha em alguma validação feita sobre objetos de requisição. Em caso de falta de permissões para acesso ao endpoint solicitado, é retornado o erro **401** ou **403**.

security – Onde ficam as configurações de segurança utilizando o **spring security** e, também, o filtro que intercepta as requisições, extraindo o **token Jwt** quando necessário e carregando os papéis (authorities) do usuário que requisitou o recurso filtrado para permitir a verificação se há permissões suficientes para acesso ao recurso requisitado.

exception – pacote onde ficam todas as exceções capturadas e tratadas nos controllers. Há também um **DTO** de resposta para envio do erro ao frontend. Esse **DTO** é a classe **ErroResponse** do pacote model.response. Nesta classe de resposta fica os códigos de erro como constantes inteiras e o mapeamento desses códigos que os vincula as mensagens de erro correspondentes a cada código. **Obs:** há na classe ErroResponse, um mapeamento para cada tipo de exceção capturada e tratada nos controllers.

util – pacote onde ficam as classes utilitárias para formatação de números de ponto flutuante, datas e horas, manipulação de **token JWT** e conversão sobre tipos **enum** definidas no pacote model.

Frontend

O frontend utiliza as tecnologias HTML, CSS, Javascript e AJAX para permitir a interação do usuário com as telas do sistema.

bootstrap 5.1.2

O framework **bootstrap** foi integrado ao sistema para facilitar a configuração de estilos das páginas HTML e utilização de componentes prontos para criação de menus, alertas, painéis, tabelas, etc. Para integrar o bootstrap ao projeto, foi necessário:

1. Adicionar os arquivos: **bootstrap.min.css** e **bootstrap.min.css.map** na pasta css/ do projeto e fazer referência ao arquivo CSS na página index.html
2. Adicionar os arquivos: **bootstrap.bundle.min.js** e **bootstrap.bundle.min.js.map** na pasta js/lib do projeto e fazer referência ao arquivo JS no final da tag body da página index.html.

Organização do frontend

O frontend está organizado da seguinte forma:

index.html – Arquivo da página padrão. Entenda que nesse arquivo que fica na raiz da pasta do frontend do projeto, serve para chamar todos os recursos CSS e JS do sistema e, também, pode servir pra definir o layout do projeto.

Lembre-se, sempre que criar um script JS ou um arquivo CSS, faça uma chamada a ele no arquivo **index.html**.

Pastas principais

Os arquivos do frontend estão organizado em três pastas principais, são elas:

pages/ – Onde devem ser colocados os conteúdos das páginas HTML que serão carregadas dinamicamente via **AJAX**. Obs: o conteúdo desses arquivos é conteúdo HTML, mas, não tem tags: <HTML>, <HEAD> ou <BODY>.

Obs: a cada vez que criar uma página, crie também um arquivo de script JS para tratar os eventos de clique em botão ou outros componentes de interface gráfica da página HTML.

css/ – Onde estão os arquivos de estilo em formato CSS e onde devem ser inseridos os novos arquivos em formato CSS criados.

Obs: A cada vez que criar um arquivo css, você deve inserir a referência a ele no arquivo index.html ou configurar um arquivo de grupo de estilos que tem formato CSS e importa outros arquivos CSS. Nesse caso, pelo menos o arquivo que importa os outros CSSs deve estar referenciado na página index.html.

js/ – Onde estão os scripts em formato javascript (JS) do sistema e onde devem ser inseridos os novos arquivos com código javascript criados.

Bibliotecas Javascript

As bibliotecas utilizadas pelos script vinculados as páginas, estão nos arquivos:

- js/lib/
  - ajax.js: biblioteca de código para simplificar o uso do AJAX. Pode ser integrada com outros projetos
- js/logica/
  - ajax2.js: biblioteca que simplifica ainda mais o uso do ajax. Isso porque foi feita para este projeto. Na função ajax2, é feito o tratamento dos códigos de erros vindos do servidor e configurada a mensagem correspondente. Logo, esta biblioteca não pode ser utilizada sem adaptações a outro projeto.
  - Sistema.js: classe do sistema que oferece diversos métodos para exibição de mensagens e carregamento de páginas. Conforme o decorrer do projeto, provavelmente, novos métodos serão implementados nesta classe.
- principal.js: script que define instâncias dos objetos utilizados na configuração de eventos das páginas HTML e nos demais scripts JS. Neste arquivo foi definido também a função que é executada após o carregamento da página index.html. Logo, de início, esta função está com um código de carregamento de uma página.

Inclusive, foi utilizado um padrão que deve ser sempre seguido no projeto para melhor organização. Isto é, cada arquivo JS que tem uma classe Javascript como conteúdo deve conter as letras das palavras em maiúsculo e cada arquivo JS que contém funções não definidas em classes, deve iniciar com letra minúscula.

Ex: ajax.js, contém apenas funções e Sistema.js, contém a classe Sistema.

Inclusão de novas entidades

Ao criar uma entidade no frontend, você deve seguir o seguinte padrão:

pages/{entidade}/{Entidade}Tela.html
pages/{entidade}/{Entidade}Form.html
pages/{entidade}/{Entidade}Detalhes.html

js/telas/{entidade}/{Entidade}Tela.js
js/tela/{entidade}/{Entidade}Form.js
js/tela/{entidade}/{Entidade}Detalhes.js

Em que:

- {entidade} corresponde ao nome da entidade com letras minúsculas e
- {Entidade} corresponde ao nome da entidade com a primeira letra em maiúsculo.

Exemplo:

Para criar a entidade de nome aluno no frontend, basta criar os arquivos com os caminhos conforme abaixo:

pages/aluno/aluno-tela.html
pages/aluno/aluno-form.html
pages/aluno/aluno-detalhes.html

js/telas/aluno/AlunoTela.js
js/telas/aluno/AlunoForm.js
js/telas/aluno/AlunoDetalhes.js

Os arquivos HTML devem conter o texto em formato HTML da página e nos arquivos JS deve conter uma classe com o nome do arquivo sem a extensão.

Exemplo:

O arquivo AlunoTela.js, pode ter, inicialmente, o seguinte conteúdo:

```
class AlunoTela {
    // atributos
    // metodos
}
```

O arquivo aluno-tela.html, pode ter, inicialmente o seguinte conteúdo:

```
<div>
        <h1>Tela de alunos</h1>
        <br />
        <div id=”mensagem-el”></div>
</div>
```

Inclusão do componente aluno:

Para simplificar o exemplo, vamos exemplificar apenas um componente. Por exemplo, o arquivo aluno-tela.html deve ser vinculado ao arquivo de script JS correspondente, que é: AlunoTela.js, logo, o seguinte arquivo deve ser alterado: js/principal.js. Neste arquivo, adicione conforme destacado em cor azul abaixo:

```
let pessoaForm = new PessoaForm();
let pessoaTela = new PessoaTela();
let alunoTela = new AlunoTela();

let componentes = {
        { ‘pessoa-form’ : { pagina : ‘pages/pessoa/PessoaForm.html’, jsObj : pessoaForm },
        { ‘pessoa-tela’ : { pagina : ‘pages/pessoa/PessoaTela.html’, jsObj : pessoaTela },
        { ‘aluno-tela’ : { pagina : ‘pages/aluno/AlunoTela.html’, jsObj : alunoTela },
}
```

Após os arquivos aluno-tela.html e AlunoTela.js criados e feitas as configurações acima, você já pode chamar em algum script ou tela o seguinte código javascript:

```
sistema.carregaPagina( ‘aluno-tela’ );
```

ou

```
sistema.carregaPagina( ‘aluno-tela’, { id : 1, nome : ‘Italo’ } );
```

No segundo caso, os parâmetros como objeto javascript JSON são utilizados pelo método que, caso exista, é executado por padrão após o carregamento da página como componente HTML.

Entenda que ‘aluno-tela’, corresponde ao valor ‘aluno-tela’ definido no objeto JSON de **componentes.**

Então, AlunoTela.js, pode ter o seguinte conteúdo, indicando uma mensagem para ser mostrada após a página ser carregada:

```
class AlunoTela {
    onCarregado( jsObj, params ) {
        sistema.mostraMensagem( 'mensagem-id', 'info', 'Página carregada com sucesso!' );
    }
}
```

Após criados os arquivos HTML e JS e a vinculação desses arquivos, basta incluir a referência ao script JS criado, na página index.html, conforme o destaque azul logo abaixo:

```
<head>
    .
    .
    .
    <script src="js/telas/pessoa/PessoaTela.js"></script>
    <script src="js/telas/aluno/AlunoTela.js></script>

    <script src="js/principal.js"></script>
<head>
```

Agora seu componente já está configurado. Basta mexer nos arquivos pessoa-tela.html e PessoaTela.js para desenvolver a devida função que o acesso a esta página deve ter!

Claro, se a página não for carregada em nenhum lugar, você não terá como visualizar como ela está.

Alterando a página inicial

Para alterar a página inicial a ser carregada com o carregamento do arquivo index.html, é só alterar a função atribuída a variável: window.onload, alterando a chamada ao método carregaPagina('pessoa-tela') do objeto sistema.

Exemplo:

Para a tela criada de nome aluno-tela.html, altere o arquivo js/principal.js, conforme o seguinte:

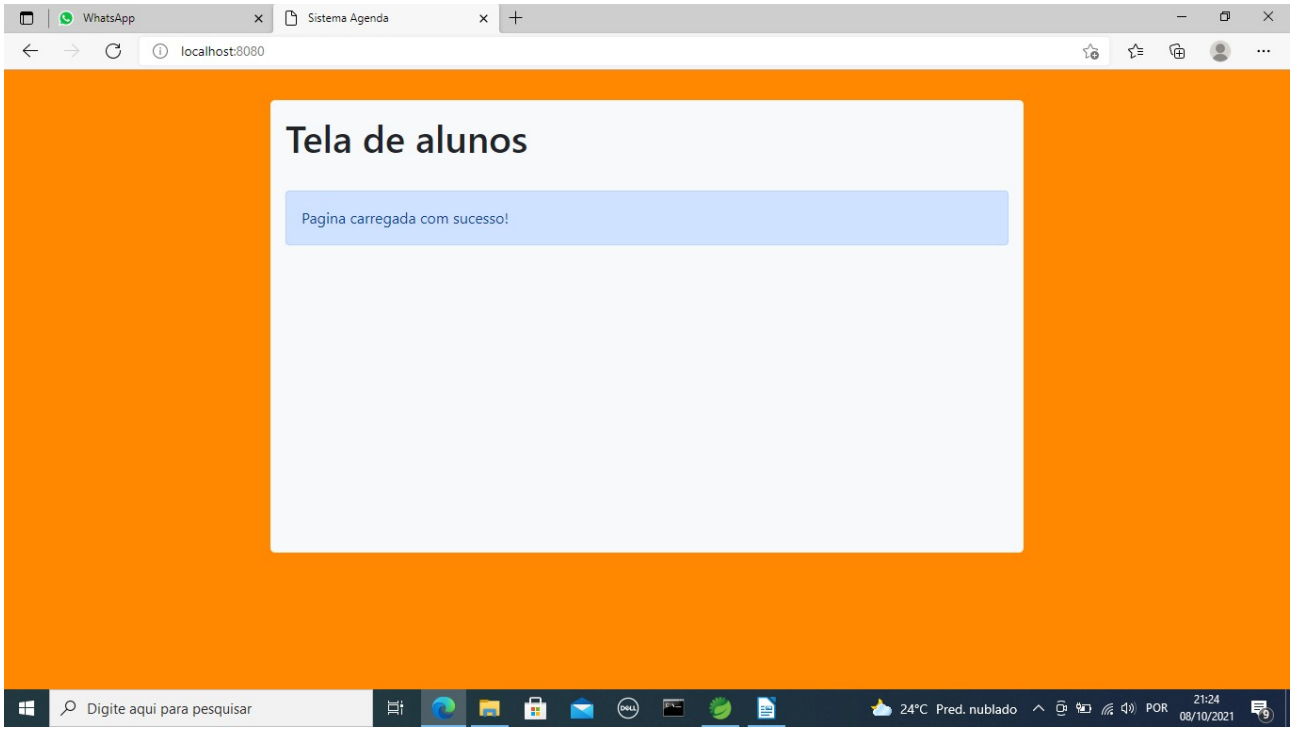
Procure pela função atribuída a window.onload e faça conforme abaixo:

```
window.onload = function() {
    // código de inicialização
    sistema.carregaPagina( 'aluno-tela' );
}
```

Pronto, agora é só testar, para tanto, faça:

1. Rode o servidor tomcat pelo MS-DOS com o comando:
   %CATALINA_HOME%\bin\startup
2. Acesse a página principal pelo navegador web:
   localhost:8080/
   ou
   localhost:8080/index.html

Deve aparecer a tela abaixo:



Agora é só mexer e alterar a página e o script!