Documentação do projeto SGEscolar

Sumário

1 Tecnologias

1 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**.

A biblioteca **lombok** foi utilizada para agilizar o escrita de métodos Gets e Sets e Construtores.

2 Frontend

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

3 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.

2 Backend

1 Java com Spring Boot

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.

2 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:

sgescolar – 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.

3 Frontend

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

1 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 lib/boostrap/ 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 lib/bootstrap do projeto e fazer referência ao arquivo JS no final da tag body da página index.html.

2 Moment 2.29.1

O framework moment foi integrado ao sistema para facilitar a manipulação de objetos ou dados de data e/ou hora. Para integrar ao projeto foi necessário

3 Organização do frontend

O frontend está organizado conforme detalhado a seguir:

1. **O arquivo: 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 faça uma referência a ele no arquivo **index.html**.

2. **O arquivo: index.js**

Arquivo onde ficam as instâncias de objetos, a configuração dos componentes em uma variável de nome “componentes” que deve ser mantida em formato **JSON** para **Javascript**.

Os componentes configurados nesse arquivo são passados como parâmetro para o construtor da classe Sistema que tem a função de fornecer meios de carregar esses componentes pelo nome dado a cada um deles.

Neste arquivo está também o método window.onload que carrega por padrão o layout e formulário de login como página principal do sistema.

3. **A pasta: componentes/**

Onde devem ser colocados os componentes. Onde, cada componente, tem uma página HTML e um Script JS vinculados a ele. Embora, seja possível criar componentes sem scripts JS associados a ele.

Recomendo criar classes nesses arquivos JS seguindo o padrão de quando se está programando em Java. Isto é, as primeiras letras de cada palavra componente da classe, devem ser maiúsculas.

Ex: para a classe pessoatela, o script deve ter o nome **PessoaTela.js** e a classe definida no script deve ter o nome **PessoaTela**.

componentes/login – Onde estão os componentes utilizados para compor a tela de login. Incluindo, também, o componente layout da página de login.

componentes/app – Onde estão os componentes, incluindo o componente layout, que compões a página da aplicação.

componentes/ext – Onde ficam os componentes extras utilizados para criação de telas prontas e configuráveis. Um exemplo é a tela de Confirmação que pode ser utilizada para pedir ao usuário que confirme uma operação de remoção.

4. **A pasta: css/**

Onde estão os arquivos de estilo em formato **CSS** e onde devem ser inseridos os novos arquivos em formato **CSS** criados. Nesta pasta os estilos estão divididos em três arquivos: **login.css, app.css** e **global.css**. O arquivo **style-list.css** apenas agrupa os outros três arquivos para que apenas ele seja referenciado na página **index.html**.

- login.css – Nesse arquivo ficam os estilos utilizados pelo layout e formulário de login.
- app.css – Neste arquivo ficam os estilos utilizados pelo layout e todas as telas da aplicação.
- global.css – Neste arquivo ficam os estilos que podem ser utilizados por qualquer arquivo html, seja da página de login ou das páginas da aplicação.

5. **A pasta: lib/**

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

- ajax.js: biblioteca de código para simplificar o uso do AJAX. Pode ser integrada com outros projetos.
- bootstrap: pasta onde ficam os arquivos componentes do framework **bootstrap**.
- moment: pasta onde ficam os arquivos da biblioteca moment para manipulação de dadas.
- menu.css: arquivo onde ficam os estilos do menu do layout utilizado como layout da aplicação do sistema.

6. **A pasta: services/**

Os scripts JS desta página são utilizados como serviços pelos scripts dos componentes. Por enquanto, há apenas um arquivo de classe JS nesta página, o arquivo Sistema.js.

- Sistema.js: classe do sistema que oferece diversos métodos para exibição de mensagens e carregamento de páginas e layouts, bem como, de componentes específicos como parte de outros componentes. Esta classe contém também os métodos para uso do **ajax** e conversão e formatação de dados em ponto flutuante e datas.

4 Biblioteca de ícones bootstrap

Os ícones bootstrap dependem do conteúdo da pasta: **fontes/** e dos arquivos **bootstrap-icons.css** e **bootstrap-icons.svg**. Há uma referência ao arquivo css desta biblioteca no index.html.

5 Padrão de nomes dos arquivos JS

Está sendo 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.

4 Inclusão de novos componentes no frontend

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

O componente tipo Tela

componentes/app/{entidade}/tela/{componente}-tela.html

componentes/app/{entidade}/tela/{Componente}Tela.html

O componente tipo Form

componentes/app/{entidade}/form/{componente}-form.html

componentes/app/{entidade}/form/{Componente}Form.html

O componente tipo Detalhes

componentes/app/{entidade}/detalhes/{componente}-detalhes.html

componentes/app/{entidade}/detalhes/{Componente}Detalhes.html

Onde:

- {entidade} corresponde ao nome da entidade com letras minúsculas e
- {componente} corresponde ao nome do componente em minúsculas. **Ex:** aluno
- {Componente} corresponde ao nome do componente em maiúsculas. **Ex:** Aluno

Exemplo:

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

componentes/app/aluno/tela/aluno-tela.html
componentes/app/aluno/tela/AlunoTela.js

componentes/app/aluno/form/aluno-form.html
componentes/app/aluno/form/AlunoForm.js

componentes/app/aluno/detalhes/aluno-detalhes.html
componentes/app/aluno/detalhes/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>
```

1 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: index.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 : 'componentes/app/pessoa/form/pessoa-form.html', jsObj : pessoaForm },
    { 'pessoa-tela' : { pagina : 'componentes/app/pessoa/tela/pessoa-tela.html', jsObj : pessoaTela },
    { 'aluno-tela' : { pagina : 'componentes/app/aluno/tela/aluno-tela.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() {
        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=”componentes/app/pessoa/tela/PessoaTela.js”></script>
    <script src=”componentes/app/aluno/tela/AlunoTela.js></script>
    ...
    <script src=”index.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.

2 Alterando a página inicial

Para alterar a página inicial a ser carregada com o carregamento do arquivo index.html, é só acessar: “componentes/app/layout/AppLayout.js” e, no final do método onCarregado alterar de carregaPagina(‘pessoa-tela’) para carregaPagina(‘aluno-tela’). Isso significa, alterar a pagina carregada com a inicialização do frontend de ‘pessoa-tela’ para ‘aluno-tela’.

Exemplo:

Para a tela criada de nome aluno-tela.html, altere o arquivo AppLayout.js, conforme o destaque em azul no método onCarregado:

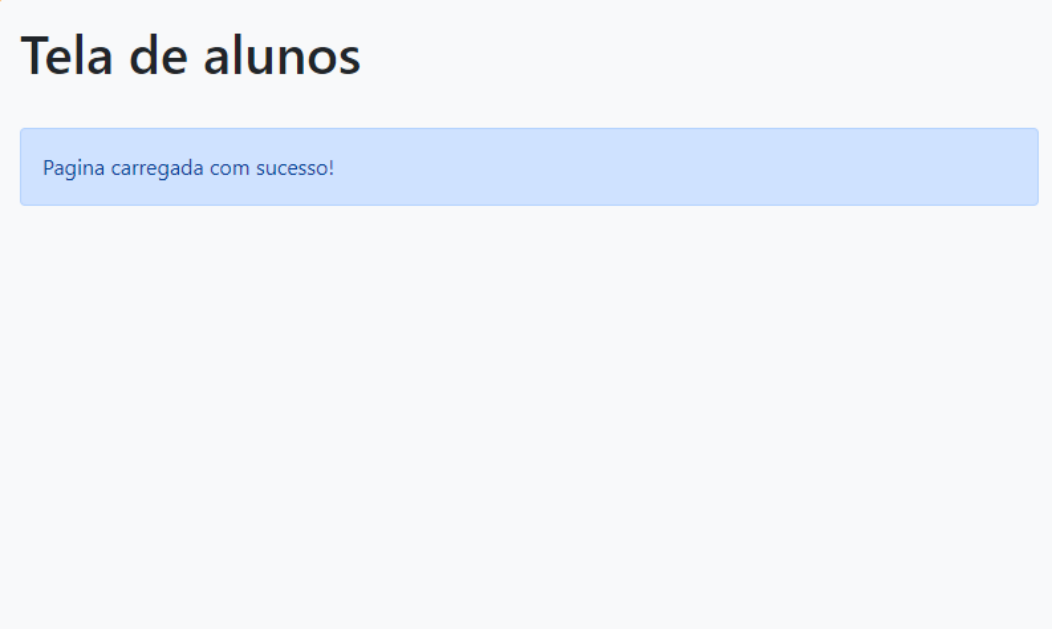
```
onCarregado() {
	this.configuraMenu();

	let grupo = sistema.globalVars.usuarioLogado.grupo.nome;
	if ( grupo === 'ADMIN' ) {
		sistema.carregaComponente( 'admin-menu', 'menu-lateral' );
	} else if ( grupo === 'SECRETARIO' ) {
		sistema.carregaComponente( 'secretario-menu', 'menu-lateral' );
	} else if ( grupo === 'PROFESSOR' ) {
		sistema.carregaComponente( 'professor-menu', 'menu-lateral' );
	} else if ( grupo === 'ALUNO' ) {
		sistema.carregaComponente( 'aluno-menu', 'menu-lateral' );
	}

	sistema.carregaComponente( 'navbar-menu', 'navbar-menu' );

	sistema.carregaPagina( 'aluno-tela' );
}
```

Pronto, agora é só testar!

3 Testando o componente

1. Rode o servidor tomcat pelo STS
2. Acesse a página principal pelo navegador web: localhost:8080/

Deve aparecer a tela abaixo:



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

5 Método onCarregado com parâmetros

Para carregar uma página, você pode utilizar o seguinte método:

```
sistema.carregaPagina( 'pessoa-form', { id : 1, nome : 'Italo' } );
```

Quando esse componente for carregado, o sistema procurará pelo método onCarregado da classe **PessoaForm.js**, que deve se parecer com o que está abaixo:

```
onCarregado() {
	//conteúdo
}
```

Perceba que você passou dois parâmetros em formato **JSON** do **Javascript** para carregamento do componente 'pessoa-form'. Assim, os parâmetros passados, podem ser acessados no método onCarregado que é executado logo depois da página do componente ser carregada. Os parâmetros podem ser acessados pela variável this.props que referencia o **JSON** passado como parâmetro.

Para acessar os parâmetros no método onCarregado de **PessoaForm.js**, veja como é fácil:

```
onCarregado() {
	let id = this.props.id;
	let nome = this.props.nome;
	let msg = ''+id+' - '+nome;
	sistema.mostraMensagem( 'mensagem-el', 'info', msg );
}
```

6 Inserindo variáveis nas páginas componentes

Uma das alterações que deram origem a nova versão das bibliotecas e serviços do sistema, foi a inclusão da possibilidade de criar variáveis dentro das páginas e poder especificar seus valores. Isto é, quando a página é carregada, essas variáveis são substituídas no HTML da página pelos seus valores configurados na operação AJAX de carregamento de HTML.

Vamos a um exemplo bem simples: uma tela de detalhes de pessoas.

Veja o HTML abaixo:

```
<div>
	<h3>Detalhes da pessoa</h3>
	<br />
	<span class=”text-info”>Nome:</span>
	<span class=”text-primary”>#{nome}</span>
</div>
```

Você pode salvar o conteúdo HTML acima em um arquivo qualquer na pasta de componentes. Pode ser:

componentes/app/pessoa/detalhes2/pessoa-detalhes.html

Agora você pode criar o componente no arquivo index.js:

```
{ ‘pessoa-detalhes2’, { pagina : ‘componentes/app/pessoa/detalhes2/pessoa-detalhes.html’ } }
```

Agora carregue o componente ‘pessoa-detalhes2’, conforme a seguir:

```
sistema.carregaPagina( ‘pessoa-detalhes2’, { nome : ‘Italo’ } );
```

Perceba que você passou um parâmetro para a página de nome vars. E esse parâmetro recebe o **JSON** { nome : ‘Italo’ } que é substituído por #{nome} no HTML quando o componente é carregado. Perceba que “nome” no **JSON** atribuído a vars, corresponde a nome de #{nome} na página **HTML**. Se ao invés da variável #{nome} fosse #{telefone} então o carregamento do componente seria conforme abaixo:

```
sistema.carregaPagina( ‘pessoa-detalhes2’, { telefone : ‘000’ } );
```

É possível também passar objetos JSON, isto é, não apenas variáveis no JSON atribuído a vars. Por exemplo, se fosse carregado assim:

```
sistema.carregaPagina( ‘pessoa-detalhes2’, {  pessoa : { nome : ‘Italo’, telefone : ‘000’ } } );
```

Então, seria procurado no HTML alguma variável com o seguinte texto: #{pessoa.nome} e #{pessoa.telefone}, para serem substituídas pelos devidos valores.