

Angular para construção de Web Apps

Curso JS-45



 **caelum**
ensino e inovação

Apostila gerada especialmente para Willian Silva Moreira - moreiraws85@gmail.com



Conheça também:



alura

alura.com.br



Casa do Código
Livros e Tecnologia

casadocodigo.com.br

Blog da Caelum



blog.caelum.com.br

Newsletter



caelum.com.br/newsletter

Facebook



facebook.com.br/caelumbr

Twitter



twitter.com/caelum

Sumário

1 Introdução	1
1.1 Construindo o alicerce da aplicação	1
1.2 AngularJS, a primeira proposta do framework da Google	1
1.3 Angular e o novo paradigma	2
1.4 Angular 2,4,5,6,7... terei que aprender tudo novamente a cada versão?	2
1.5 Typescript	3
1.6 Single Page Application	3
1.7 Web Components	3
1.8 Cmail: o projeto deste curso	4
1.9 Ambiente para o Angular e ferramentas	6
1.10 Repositório de arquivos base do curso:	8
1.11 API para o curso	9
2 Exercício: Criando um projeto Angular	10
2.1 Objetivo	10
2.2 Passo a passo com código	11
3 Uma introdução aos componentes	14
3.1 Componentes Angular	14
3.2 Classes no TypeScript	15
3.3 Metadados de componentes	15
3.4 Definindo um bom selector: prefixos	17
3.5 Controllers, templates e views	18
3.6 Bibliotecas de CSS	18
4 Exercício: CSS e Header Component	20
4.1 Objetivo	20
4.2 Passo a passo com código	21
5 Explorando a sintaxe dos templates Angular	27
5.1 Interpolação nos templates	27

5.2 Property binding	28
5.3 One-way data-binding	28
5.4 Event binding	29
5.5 Diretivas do tipo atributo "de fábrica" (built-in)	29
5.6 A diretiva Ng-class	30
6 Exercício: Dando vida para o menu mobile	32
6.1 Objetivo	32
6.2 Passo a passo com código	33
7 Workspaces e configurações por projeto	35
7.1 Estilos globais	35
7.2 Getters e modificadores de acesso no TypeScript	36
8 Exercício: formulário para enviar email	38
8.1 Objetivo	38
8.2 Passo a passo com código	39
9 Event Binding mais a fundo	42
9.1 \$event a referência ao objeto dos eventos	42
9.2 Recebendo parâmetros em um método de uma classe	43
9.3 Obtendo o valor de inputs com \$event	44
9.4 Propriedades de uma classe: tipos e possibilidades	45
9.5 Validação básica com HTML5	48
9.6 Iterando em listas no template *ngFor	48
9.7 Change detection - detecção de mudanças no Angular	49
10 Exercício: Enviando emails do jeito mais simples do universo	51
10.1 Objetivo	51
10.2 Passo a passo com código	52
11 Template driven forms	55
11.1 Módulos Angular	55
11.2 O módulo principal da aplicação AppModule	56
11.3 Utilizando o módulo de formulários	56
11.4 Variáveis de referência de template	57
11.5 Enviando os dados do formulário	58
11.6 NgModel, a diretiva bidirecional	61
11.7 O evento ideal para formulários (ngSubmit)	62
11.8 Exibindo mensagens de validação	63
11.9 Diretiva email para validação de campo tipo email	65

12 Exercício: Validação e variáveis de template	67
12.1 Objetivo	67
12.2 Passo a passo com código	68
13 Arquitetura de projeto e CLI	72
13.1 Propostas para arquitetura	72
13.2 CLI para tarefas repetitivas	75
14 Roteamento simples	78
14.1 Registrando um módulo de roteamento	78
14.2 Configuração para rotas	79
14.3 Saída do roteamento	80
14.4 Links em uma SPA	81
14.5 Rotas curingas	82
15 Exercício: Roteamento simples	85
15.1 Objetivo	85
15.2 Passo a passo com código	86
16 Exercício: Links para as rotas e redirecionamento	95
16.1 Objetivo	95
16.2 Passo a passo com código	95
17 Componentes, diretivas e ciclo de vida	97
17.1 Ciclo de vida dos componentes	97
17.2 Sequência do ciclo de vida	98
17.3 Interfaces no Typescript	99
17.4 Diretivas	100
17.5 Usando injeção de dependência no Angular	101
17.6 Acesso à elementos do DOM com ElementRef	102
17.7 Incluindo templates dinâmicos com ng-content	103
18 Exercício: Cadastro de contas - um componente para formulários mais eficiente	105
18.1 Objetivo	105
18.2 Passo a passo com código	106
19 Introdução aos formulários reativos	111
19.1 A abordagem de formulário reativos	111
19.2 Criando um formulário reativo	112
19.3 Explorando a API de formulários reativos	112
19.4 Obtendo os dados do formulário	115

20 Exercício: Cadastro de contas - formulários reativos	116
20.1 Objetivo	116
20.2 Passo a passo com código	117
21 Validações nos formulários reativos	120
21.1 Validação nativa funciona	120
21.2 Form Validators	121
21.3 Criando atributos do componente	123
22 Exercício: Cadastro de contas - melhorando o feedback de erros	127
22.1 Objetivo	127
22.2 Passo a passo com o código	128
23 Múltiplas regras de validação	131
23.1 Combinando funções de validações	131
23.2 Trabalhando as mensagens de erro no template	133
23.3 Condições no template com *ngIf	135
24 Exercício: Cadastro de contas - validações múltiplas	137
24.1 Objetivo	137
24.2 Passo a passo com o código	138
25 Requisições HTTP no Angular	140
25.1 HttpClient	140
25.2 Métodos de HttpClient	141
25.3 Criando uma requisição HTTP com Angular: HttpClient	141
25.4 Disparando a requisição HTTP	142
25.5 Trabalhando erros de HTTP	144
25.6 Opções para as requisições	145
25.7 Um pouco de RxJS	145
26 Validações Assíncronas	147
26.1 Validadores assíncronos	147
26.2 Feedback de erros	151
26.3 Controlando o disparo das requisições	151
27 Exercício: Cadastro de contas - validações assíncronas e requisições Http	153
27.1 Objetivo	153
27.2 Passo a passo com código	154
28 Enviado dados do cadastro para a API	157
28.1 Persistência de dados com JS e a necessidade de uma API backend	157

28.2 CMail Back API	158
28.3 Enviando dados pelo HTTP	158
28.4 Data Transfer Object: conceito e criação;	159
28.5 Redirecionando rotas	160
29 Exercício: Enviando dados do formulário com HttpClient	162
29.1 Objetivo	162
29.2 Passo a passo com código	162
30 Módulos no Angular	167
30.1 NgModule	167
30.2 Carregamento preguiçoso	168
30.3 Criação de módulos com a CLI	169
31 Exercício: Trabalhando com módulos	170
31.1 Objetivo	170
31.2 Passo a passo com código	170
32 Subroteamento: Rotas filhas	174
32.1 Módulos por rotas	175
32.2 Adicionado um subroteamento	176
32.3 Árvore de roteamento	177
33 Exercício: Subroteamento com rotas filhas	179
33.1 Objetivo	179
33.2 Passo a passo com código	180
34 Solicitando autenticação	183
34.1 Autenticação e responsabilidade do lado cliente	183
34.2 Armazenamento local	184
35 Exercício: Começando a tela de login	186
35.1 Objetivo	186
35.2 Passo a passo com código	187
36 Serviços no Angular	190
36.1 Criando um serviço	190
36.2 Tipos customizados	193
36.3 Assinatura de métodos no Typescript	193
37 Exercício: Melhorando a qualidade do código, criando um serviço	195
37.1 Objetivo	195
37.2 Passo a passo com código	195

38 Guardiões de rotas	198
38.1 O que é um guard?	198
38.2 Interfaces para o guard	199
39 Exercício: Protegendo as rotas da aplicação com Guards	202
39.1 Objetivo	202
39.2 Passo a passo com código	202
40 HTTP Headers, variáveis de ambiente e casting	205
40.1 Configurando HttpHeaders no HttpClient	205
40.2 Variáveis de ambiente	206
40.3 Casting no retorno do método post	208
41 Exercício: Serviço para enviar emails	210
41.1 Objetivo	210
41.2 Passo a passo com código	210
42 Exercício: Listando componentes para os emails	214
42.1 Objetivo	214
42.2 Passo a passo com código	215
43 Exercício: Aprimorando o componente cmail-list-item	216
43.1 Objetivo	216
43.2 Passo a passo com código	216
44 Exercício: Pegando os emails do servidor	220
44.1 Objetivo	220
44.2 Passo a passo com código	221
45 Transformando dados com Pipes	222
45.1 Usando pipes	222
45.2 Built-In Pipes	223
45.3 Parametrizando um pipe: slicePipe	223
46 Exercício: Transformando dados com Pipes	225
46.1 Objetivo	225
46.2 Passo a passo com código	226
47 Emissão de eventos	227
47.1 Eventos customizados	227
48 Exercício: Apagando emails da API com EventEmitter	230
48.1 Objetivo	230

48.2 Passo a passo com código	231
49 Broadcasting	236
49.1 Transmissor	236
49.2 Subject	237
49.3 Transmitindo dados	237
49.4 Assinando dados	237
50 Exercício: Broadcast do título da página	239
50.1 Objetivo	239
50.2 Passo a passo com código	240
51 Criando seu próprio *pipe*	243
51.1 Decorator Pipe	243
52 Exercício: Criando um pipe de filtro para emails	246
52.1 Objetivo	246
52.2 Passo a passo com código	247
53 Exercício: Filtrando arrays de acordo com o novo Angular	250
53.1 Objetivo	250
53.2 Passo a passo com código	251
54 Exercício Desafio: Desenvolvendo a página interna dos emails	252
54.1 Objetivo	252
54.2 O que deve ser implementado?	252
54.3 Dicas	253

Versão: 23.8.15

INTRODUÇÃO

1.1 CONSTRUINDO O ALICERCE DA APLICAÇÃO

Quando desenvolvemos no server-side, organizamos nosso código em camadas para facilitar a manutenção, o reaproveitamento e a legibilidade de nosso código. É muito comum aplicarmos o modelo Model, View, Controller (MVC), que consiste na separação de responsabilidades pelas camadas propostas, e assim facilitando a reescrita de alguma parte e a manutenção do código.

Porém, não é raro que a mesma pessoa que desenvolve o server-side, quando codifica no client-side acaba por deixar de lado estas práticas. Mesmo aquelas que procuram organizar melhor seu código acabam criando soluções caseiras, e que na maior parte das vezes, são pouco documentadas.

Tendo como base este cenário, frameworks JavaScript inspirados no MVC foram criados. Dentre eles temos o React, Vuejs, Ember, Backbone e outros.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

1.2 ANGULARJS, A PRIMEIRA PROPOSTA DO FRAMEWORK DA GOOGLE

Criado como um projeto interno da Google e liberado para o público em 2009, se tornando muito popular logo em seguida, o AngularJS tem como foco a criação de Single Page Applications (SPA's). Essas aplicações, cuja principal característica é o não recarregamento da página durante seu uso, permitem dessa forma uma experiência de navegação e uso mais fluída e agradável ao usuário final.

1.3 ANGULAR E O NOVO PARADIGMA

Em setembro de 2016, a Google anuncia o lançamento de um framework completamente novo para SPA's. Inicialmente chamado de "Angular 2" e hoje apenas Angular, esse novo framework tem uma diferença interestelar em relação ao antigo AngularJS. Não importando assim, o quão bom você seja com AngularJS, pois estamos falando de um paradigma totalmente diferente!

Um paradigma, em poucas palavras, é a visão do mundo que temos e como procuramos respostas para solucionar problemas. Quando um paradigma muda, todos começam do zero. Dessa forma, se você nunca trabalhou com Angular, ótimo! Se você vem do AngularJS, peço que esvazie a sua mente para novas possibilidades e caminhos deste framework.

1.4 ANGULAR 2,4,5,6,7... TEREI QUE APRENDER TUDO NOVAMENTE A CADA VERSÃO?

O **semantic versioning** é o formato escolhido para fazer o versionamento de vários projetos open source, desde projetos pequenos até mesmo grandes, como **React** (<https://reactjs.org/>) e o próprio **Angular**. A ideia por trás é ter uma forma de verificar facilmente se a atualização que ocorreu pode ou não gerar conflitos com a base de código que usa o mesmo. Em linhas gerais, sempre vemos o versionamento com números como: 10.5.0. Esses números tem significado e, dado um número de versão MAJOR.MINOR.PATCH, devemos incrementar cada um deles de acordo com os seguintes critérios:

- Versão MAJOR quando você faz alterações de API incompatíveis (mudar o nome de algum método ou removê-lo);
- Versão MINOR quando você adiciona funcionalidade de maneira compatível com versões anteriores (alguma funcionalidade é adicionada mas não impacta em outras existentes);
- PATCH versão quando você faz correções de bugs compatíveis com versões anteriores (acontece alguma correção de bug em função de uma minor ou major release).

Este receio de muitas pessoas se dá pelo fato de que quando o Angular saiu da versão 1 para a versão 2 ele praticamente mudou o projeto inteiro, inclusive o nome que antes era AngularJS e agora é somente **Angular**. Felizmente, de acordo com o caminho que o projeto está tomando podemos ficar mais despreocupados com relação a uma mudança radical como a que aconteceu no passado.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

1.5 TYPESCRIPT

É uma linguagem criada pela Microsoft, que na prática é um *superset* do JavaScript. Em outras palavras, a ideia é adicionar comportamentos como tipos de dados estáticos, estruturas mais comuns em outras linguagens orientada a objetos para trabalhar com classes, interfaces e diversos outros recursos em cima do JavaScript convencional. Se você sabe escrever código JavaScript esse código também é um TypeScript válido, a vantagem real de utilizar essa nova forma de escrever é poder tirar proveito dos recursos citados anteriormente para construir projetos mais robustos.

1.6 SINGLE PAGE APPLICATION

O termo se popularizou muito quando muitas empresas começaram a se inspirar no cliente web do GMail para criarem aplicações com uma boa experiência e deixando más práticas de lado, como a de ficar o tempo todo recarregando completamente uma página para mostrar atualizações pontuais na tela. Com isso, uma SPA ganha uma carga gigantesca de código JavaScript para gerenciar desde trocas de página dinâmicamente, sem necessidade de carregar novos arquivos HTML. Devido a esta nova abordagem de desenvolvimento, todas as tecnologias ao redor foram evoluindo, desde micro interações que foram se aprimorando com o passar dos anos com cada vez mais detalhes, até animações com CSS e entre outros pontos.

1.7 WEB COMPONENTS

Um outro tópico interessante sobre o Angular, principalmente a versão mais nova, é que ela foi construída se baseando na especificação (spec) dos Web Components que está vindo para a web. Durante muito tempo o front-end era nada mais que só o resultado com HTML e CSS do que precisava ser exibido em uma tela com o back-end como centro. Os tempos foram passando, os sites foram

ficando cada vez mais complexos o que começou a gerar muitos trechos repetidos de código e uma dificuldade de fazer escalar uma aplicação em que o front-end fosse grande e precisasse de reuso. Muitos frameworks surgiram como o AngularJS com o conceito de diretivas, mas a frente o React, Vue e o próprio W3C (World Wide Web Consortium) começou a ver que o navegador poderia oferecer algo nativo pela plataforma da Web que possibilitasse este desejado maior reuso, para facilitar o desenvolvimento e manutenção de projetos grandes, Deu-se assim o início desta especificação, que hoje é uma das fontes de inspiração para o Angular e o core de um outro projeto da Google chamado Polymer.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

1.8 CMAIL: O PROJETO DESTE CURSO

The screenshot shows the registration page for CMail. At the top, there's a header bar with the title "Cadastro - CMail" and a search bar labeled "Filtrar e-mails...". On the left, a sidebar features the Cmail logo, a placeholder profile picture, the text "Pessoa de Tal", and an email address "pesso.al@cmail.br". Below this are links for "Login", "Cadastro", "Caixa de Entrada", and "Logout". The main form area has fields for "Nome", "Username", "Senha", "Telefone", and "Avatar", each with a corresponding input field. A red "CADASTRAR" button is positioned at the bottom right of the form.

Figura 1.4: CMail - cadastro

The screenshot shows the login page for CMail. The header bar displays "Login - CMail" and the URL "localhost:4200". The main area has a blue polygonal background with the text "Bem vindos/as ao CMail!". It contains fields for "Email" and "Senha". At the bottom left is a link "PRIMEIRO LOGIN? CADASTRE-SE!", and at the bottom right is a red "ENTRAR" button.

Figura 1.3: CMail - login

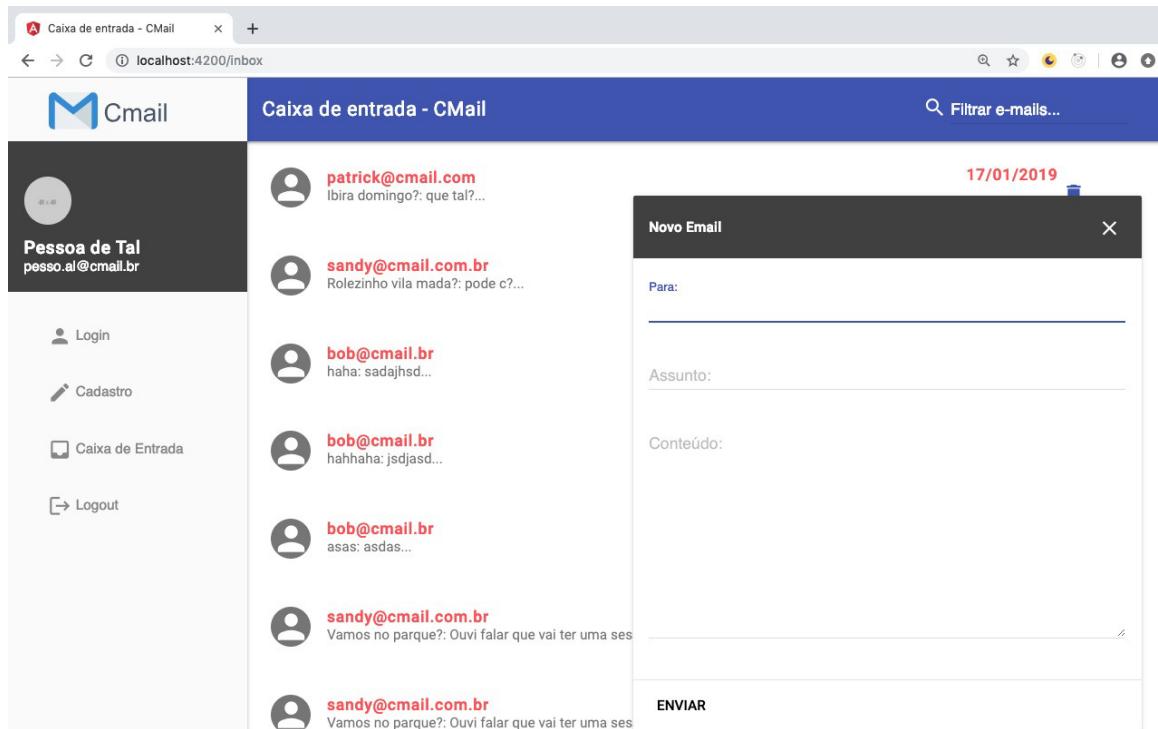


Figura 1.2: CMail - novo email

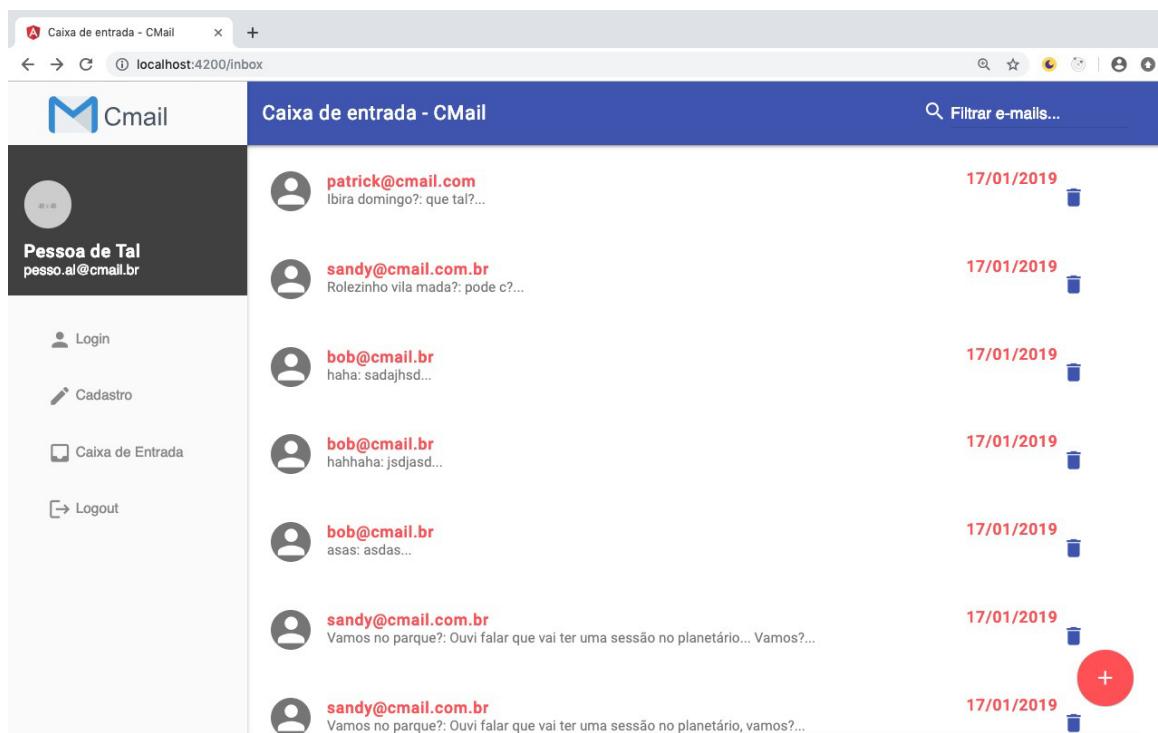


Figura 1.1: CMail - caixa de entrada

1.9 AMBIENTE PARA O ANGULAR E FERRAMENTAS

Instalando o Node.js em casa

6 1.9 AMBIENTE PARA O ANGULAR E FERRAMENTAS

Apostila gerada especialmente para Willian Silva Moreira - moreiraws85@gmail.com

Para instalá-lo, siga as instruções abaixo referentes à sua plataforma:

Em qualquer sistema operacional

Verifique antes se o Node.js já está instalado:

```
nodejs -v
```

ou

```
node -v
```

Este comando exibe a versão do Node.js e é importante que a versão instalada seja igual ou superior a 6.5.0. Caso o comando seja desconhecido, prossiga para instruções da sua plataforma.

Linux (Ubuntu)

No Ubuntu, através do terminal (permissão de administrador necessária) execute o comando abaixo:

```
sudo apt-get install -y nodejs
```

Caso tenha dificuldades em **atualizar** a versão do Node.js, tente executar o seguinte comando:

```
curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

ATENÇÃO: em algumas distribuições Linux, pode haver um conflito de nomes quando o Node é instalado pelo apt-get. Neste caso específico, no lugar do binário ser node, ele passa a se chamar **nodejs**. Isso gera problemas, pois a instrução **npm start** não funcionará, uma vez que ela procura o binário node e não nodejs . Para resolver, use a seguinte instrução no terminal para subir o servidor:

```
sudo ln -s /usr/bin/nodejs /usr/bin/node  
node server
```

Windows

Baxe o instalador clicando no grande botão `install` diretamente da página do Node.js. Durante a instalação, você apenas clicará botões para continuar o assistente. Não troque a pasta padrão do Node.js durante a instalação a não ser que você saiba exatamente o que está fazendo.

Mac OSX

O [homebrew][5] é a maneira mais recomendada para instalar o Node.js em sua máquina, através do comando:

```
brew update  
brew install node
```

Não usa homebrew? Sem problema, baxe o instalador clicando no grande botão `install` diretamente da página do Node.js.

Angular CLI

Junto com o desenvolvimento do Angular, a equipe da Google começou a desenvolver em paralelo a **Angular CLI**. Ela é uma ferramenta de linha de comando que ajuda a montar a infra da sua aplicação e acelerar a criação de componentes.

Instalando a Angular CLI globalmente

Ferramentas de linha de comando são geralmente instaladas globalmente para que possamos utilizá-las em qualquer local da nossa aplicação. **Mas muita atenção, porque instalar pacotes do Node.js requer permissão de administrador. Tenha certeza de ter os privilégios de administrador antes de continuar.** A Angular CLI tem como dependência o Node.js com versão v6.5.0 ou superior, e o **npm** (gerenciador de pacotes do Node.js).

Verifique se o Node.js e o npm estão devidamente instalados e em suas versões mais recentes:

Node.js

```
node -v
```

npm

```
npm -v
```

Agora, para instalar a **Angular CLI** execute o comando:

```
npm install -g @angular/cli
```

Para verificar se a instalação ocorreu corretamente, execute:

```
ng --version
```

Caso queira **desinstalar** a CLI:

```
npm uninstall -g @angular/cli
```

1.10 REPOSITÓRIO DE ARQUIVOS BASE DO CURSO:

Ao longo do curso, em diversos momentos, iremos utilizar trechos de código, principalmente HTML. Para agilizar o desenvolvimento e conseguirmos focar nas features do Angular, tais códigos estão disponíveis no repositório:

- <https://github.com/caelum/projeto-js45>

Saber inglês é muito importante em TI



cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

1.11 API PARA O CURSO

Durante este curso, vamos precisar de um servidor web para que a aplicação se comunique com um back-end e possa persistir os dados. Esse servidor chamado de CMail Back está disponível em:

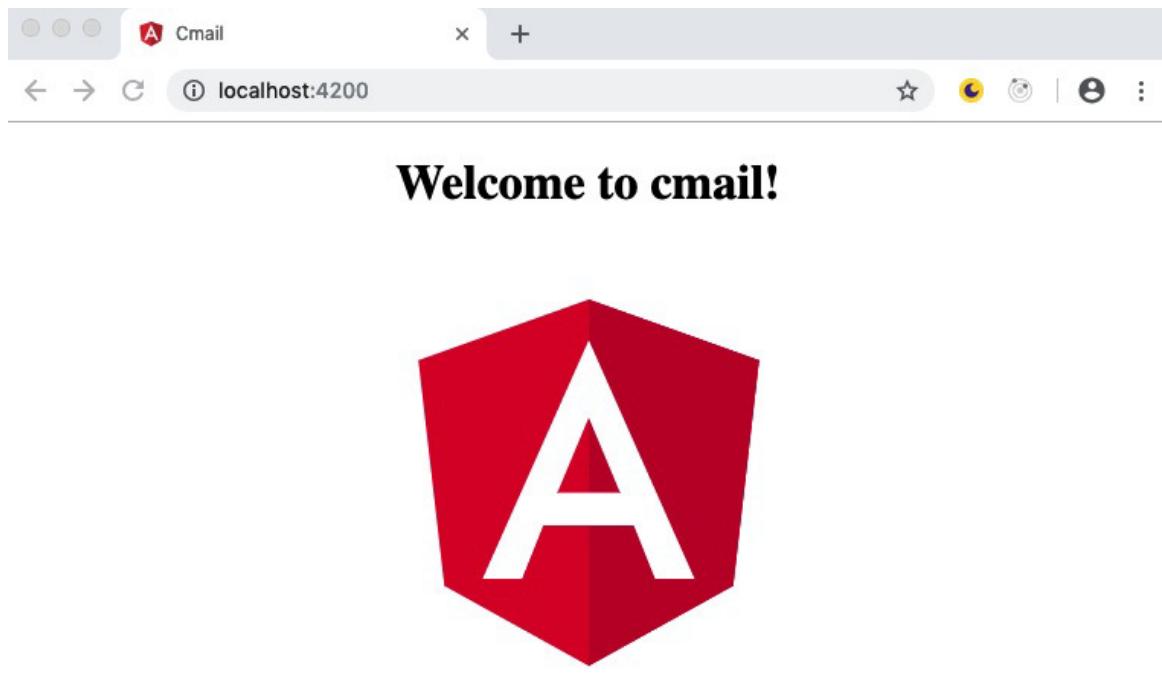
- <https://github.com/caelum/cmail-back>

Após baixar a API em seu computador, acesse a pasta do projeto pelo terminal do sistema operacional, instale as dependências com `npm install` e depois o inicie com `npm start`.

EXERCÍCIO: CRIANDO UM PROJETO ANGULAR

2.1 OBJETIVO

Neste exercício vamos aprender a como começar um novo projeto Angular usando a interface de linha de comando, ou *Command Line Interface* (CLI). Ao final teremos um projeto base funcionando, que será o ponto de partida para criarmos o CMail!



Here are some links to help you start:

- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

Figura 2.1: Welcome to Cmail!

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

2.2 PASSO A PASSO COM CÓDIGO

1. Verifique as versões do Nodejs e NPM no terminal do seu computador:

```
node -v
```

```
e
```

```
npm -v
```

Observação: as versão do Nodejs deve ser maior que 8, e do NPM maior que 5.

2. Verifique se a CLI do Angular está instalada:

```
ng --version
```

Deverá aparecer algo como:

```
v:JS-45 $ ng --version
```



Angular CLI: 7.1.3

Node: 10.2.1

OS: darwin x64

Angular:

—

Package	Version
@angular-devkit/architect	0.11.3
@angular-devkit/core	7.1.3
@angular-devkit/schematics	7.1.3
@schematics/angular	7.1.3
@schematics/update	0.11.3
rxjs	6.3.3
typescript	3.1.6

Figura 2.2: ng --version

Caso não estiver instalada, basta executar o seguinte comando, onde será instalada globalmente:

```
npm install -g @angular/cli
```

3. Agora podemos gerar um novo projeto Angular com a CLI, no nosso caso será o **cmail**:

ng new cmail

Ao executar este comando, duas perguntas serão feitas em seguida.

I) ? Would you like to add Angular routing? (y/N)

****Neste momento vamos responder**: **`N`****

II) ? Which stylesheet format would you like to use? (Use arrow keys)

```
› CSS
SCSS [ http://sass-lang.com ]
SASS [ http://sass-lang.com ]
LESS [ http://lesscss.org ]
Stylus [ http://stylus-lang.com ]
```

Escolheremos **CSS**, para isto basta apertar **enter**.

A ferramenta deixa explícito o suporte aos pré-processadores de CSS mais populares caso o projeto precisar utilizar algum. Você pode navegar nas opções com as setas para cima ou para baixo para escolher uma opção diferente.

Com isto, uma pasta **cmail** é criada, e dentro os arquivos base são gerados e a dependências do

projeto baixadas, podendo demorar alguns minutos para terminar.

4. Utilizando o editor Visual Studio Code (VSCode), ou o seu favorito, abra a pasta `cmail` para visualizarmos a estrutura de projeto criada.

Com o terminal integrado do VSCode, que já é aberto no diretório do projeto, vamos inicializar o projeto com a CLI:

```
ng serve
```

O comando `ng serve` vai fazer *build* do projeto e subir um servidor de desenvolvimento local, com *live-reload*, disponibilizando o projeto no endereço `http://localhost:4200`. Uma das opções deste comando é o parâmetro `-o` que abre o navegador automaticamente no endereço do servidor local.

UMA INTRODUÇÃO AOS COMPONENTES

Assim como outros frameworks e bibliotecas para Single Page Application, o Angular implementa o conceito de componentes seguindo o padrão para Web Componentes que está sendo desenvolvido pelo W3C.

O conceito de um componente é compor uma **unidade de software** isolada em um elemento customizado (custom element), onde esta unidade é composta de HTML (template), CSS (estilos), JS (comportamento e lógica) e dados (que podem ser estáticos ou dinâmicos).



Figura 3.1: Component Model

3.1 COMPONENTES ANGULAR

Um componente Angular controla um trecho de tela chamado **view**. A lógica de um componente é definida dentro de uma classe e esta interage com seu template por meio de uma API de propriedades e métodos.

O Angular cria, atualiza e destrói componentes conforme o usuário navega na aplicação, nos permitindo realizar ações para os componentes a cada momento desta navegação, por meio de ganchos de ciclo de vida opcionais (*lifecycle hooks*) que serão estudados mais a frente.

Além disso, uma aplicação Angular deve conter pelo menos um componente, o componente raiz (*root component*), no qual será conectado o DOM com os componentes Angular.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

3.2 CLASSES NO TYPESCRIPT

Por padrão o framework Angular utiliza TypeScript (TS) no lugar do JavaScript (JS). Contudo, no fim, este TS é transpilado pelo seu compilador (TSC) e é entregue um JS para o navegador. Esta tarefa já vem configurada e pronta para uso quando criamos o projeto usando a CLI.

As classes no TS são muito similares às do JS. Porém temos uma enorme quantidade de recursos a mais, que o JS ainda não suporta nativamente, como a possibilidade implementar interfaces, modificadores de acesso para propriedades e métodos, tipos de dados estáticos dentre outros.

Portanto, cada componente Angular para existir, minimamente deve conter uma classe. Esta classe deve ser acessível por outros módulos (`export`) e ter um nome seguindo a convenção do framework. Por exemplo:

```
export class HeaderComponent { }
```

Seguindo em frente, para definir propriedades da classe, um construtor ou métodos, basta definirmos seus nomes e valores (no caso das propriedades) diretamente no corpo da classe, como no exemplo:

```
export class HeaderComponent {  
  titulo = 'Cmail';  
  constructor(){}
}
```

Porém o que diferencia uma classe TypeScript normal de um componente Angular?

3.3 METADADOS DE COMPONENTES

Na prática, um componente Angular é definido com uma simples classe do TypeScript que contém um **decorator** na linha acima da declaração desta classe.

Os **decorators** são funções que modificam as classes, definindo uma espécie de tipo ou categoria da classe através dos seus metadados, que determinam como esta classe deve funcionar. No código abaixo, você pode ver que neste momento **HeaderComponent** é apenas uma classe comum, sem nenhuma notação Angular ou sintaxe especial. Ela não será um componente até que seja marcada com um **decorator @Component**.

```
export class HeaderComponent {  
  titulo = 'Cmail';  
  
  constructor(){}
}
```

O **decorator @Component**, provido pela biblioteca **core** do Angular, identifica a classe imediatamente abaixo como uma classe de componente e informa seus metadados.

Os metadados de um componente informam ao Angular aonde obter o necessário para criar e apresentar o componente e sua visualização. Para isto, associamos um *template* ao componente, fazendo referência à um arquivo de *template* (HTML) ou o escrevendo de forma *inline*. Juntos, o componente e seu *template* formam uma *view*.

Além de conter ou apontar para um *template*, nos metadados do **@Component** devemos definir como o componente vai ser referenciado em um HTML, através do **selector**, e podemos também definir seus estilos, apontando um arquivo ou escrevendo de forma *inline* como no caso do *template*.

```
import { Component } from "@angular/core";  
  
@Component({  
  selector: 'cmail-header',  
  templateUrl: './header.component.html',  
  styleUrls: [  
    './header.component.css'  
  ]  
})  
export class HeaderComponent {  
  
  titulo = 'Cmail';  
  
  constructor(){}
}
```

No exemplo acima, usamos algumas das opções de configuração de **@Component**:

- **selector** : é um seletor de CSS que diz ao Angular para criar e inserir uma instância deste componente em qualquer lugar que ele encontrar a tag correspondente em um template. Por exemplo, se um HTML da aplicação contiver `<cmail-header></cmail-header>`, então o Angular vai inserir uma instância da view de HeaderComponent entre estas tags.
- **templateUrl** : é o endereço do *template* HTML relativo a este componente. Como alternativa, você pode escrever um *template inline*, como valor da propriedade **template**. Este *template*

define como o componente armazena a *view*.

- `styleUrls` : é uma lista de endereços de arquivos de estilos para o componente. Como alternativa, é possível usar a propriedade `styles` , onde você pode escrever os estilos *inline*, ainda através de uma lista.

3.4 DEFININDO UM BOM SELECTOR: PREFIXOS

Ao definir o valor da propriedade `selector` do *decorator* `@Component` , podemos localizar o elemento daquele selector, caso contrário, se for informado apenas um nome, será criada uma tag daquele nome. Sendo assim, se o selector for `cmail-header` , o Angular criará a tag `<cmail-header>` `</cmail-header>` e dentro delá será exibida a *view* do componente. Criar tags para os componentes é o mais recomendado, pois segue o padrão dos Web Componentes. Porém como garantir que aquele componente será o único com aquele nome de tag em toda a aplicação?

Para garantir de alguma forma que não haverão componentes diferentes com o mesmo nome de tag, é fortemente recomendado (inclusive não passa no *linter* caso não o faça) que sejam criados *selectors* com prefixos, como fizemos já com `cmail-header` , onde o prefixo é `cmail` separado por hífen do nome do componente.

Neste caso, se deixássemos apenas `header` , toda tag `header` receberia uma instância do componente, o que seria um grande risco para um correto funcionamento da aplicação.

Portanto devemos considerar os prefixos dos *selectors* como uma espécie de *namespace* , assim evitando conflitos com outros componentes, inclusive com bibliotecas de componentes que podemos adicionar ao projeto ou até mesmo com as próprias tags padrões do HTML.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

3.5 CONTROLLERS, TEMPLATES E VIEWS

Através da interação em uma instância de um componente, o Angular gerênciaria tudo que um usuário vê e faz.

Algumas pessoas que desenvolvem aplicações estão acostumadas a trabalhar com o modelo **MVC** (model-view-controller) ou **MVVM** (model-view-viewmodel), no Angular a classe do componente assume grande parte do **controller**.

Uma **view** é como chamamos a composição de lógica e template, ou seja, a classe TypeScript e o HTML. O **template** é apenas um pedaço de HTML que formará a view de um componente.

Dentro de uma view, podemos ter outros componentes, pertencentes ao mesmo módulo, e também de outros módulos, se estes estiverem em um mesmo contexto de acordo com a hierarquia das views, representada pela árvore de componentes.

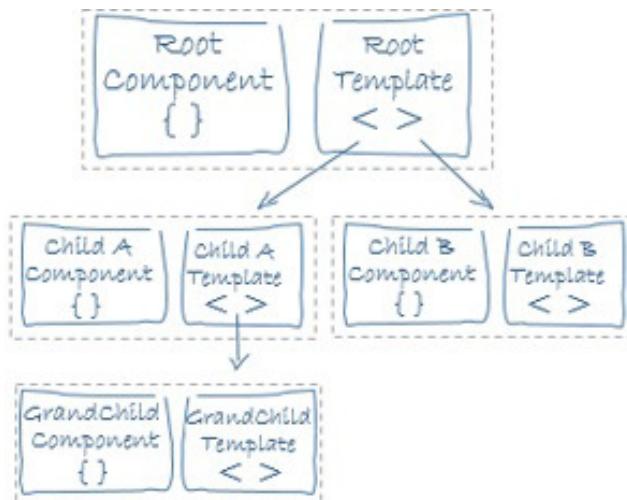


Figura 3.2: Árvore de componentes

3.6 BIBLIOTECAS DE CSS

Para adicionar bibliotecas de terceiros, como o Material Design Lite (MDL), ou Bootstrap, através das CDNs (Content Delivery Network – Rede de Distribuição de Conteúdo) - ou seja, links públicos dos arquivos hospedados em servidores abertos - podemos fazer o link à estes arquivos de CSS diretamente no `index.html`.

Aqui um exemplo de como inserir a biblioteca MDL - <https://getmdl.io/> - pela sua CDN:

```
<!doctype html>
<html lang="pt-br">
<head>
  <meta charset="utf-8">
  <title>CMail</title>
  <base href="/">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="icon" type="image/x-icon" href="favicon.ico">

<!-- Material Design -->
<link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons">
<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Roboto:300,400,500">
<link rel="stylesheet" href="https://code.getmdl.io/1.3.0/material.indigo-red.min.css" />
<!-- ./Material Design -->

</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Referências

1. <https://github.com/w3c/webcomponents>
2. <https://angular.io/guide/architecture>
3. <https://angular.io/guide/architecture-components>
4. <https://angular.io/guide/template-syntax>

EXERCÍCIO: CSS E HEADER COMPONENT

4.1 OBJETIVO

Vamos começar a construir o cabeçalho da página, para que em seguida o CMail fique com a aparência da figura a abaixo. Como o cabeçalho é uma parte do **layout** que utilizamos em várias páginas, vamos isolar seu código para que se seja reutilizável, para isto faremos dele um componente, será o **HeaderComponent**.

<https://github.com/caelum/projeto-js45/tree/master/01/src>

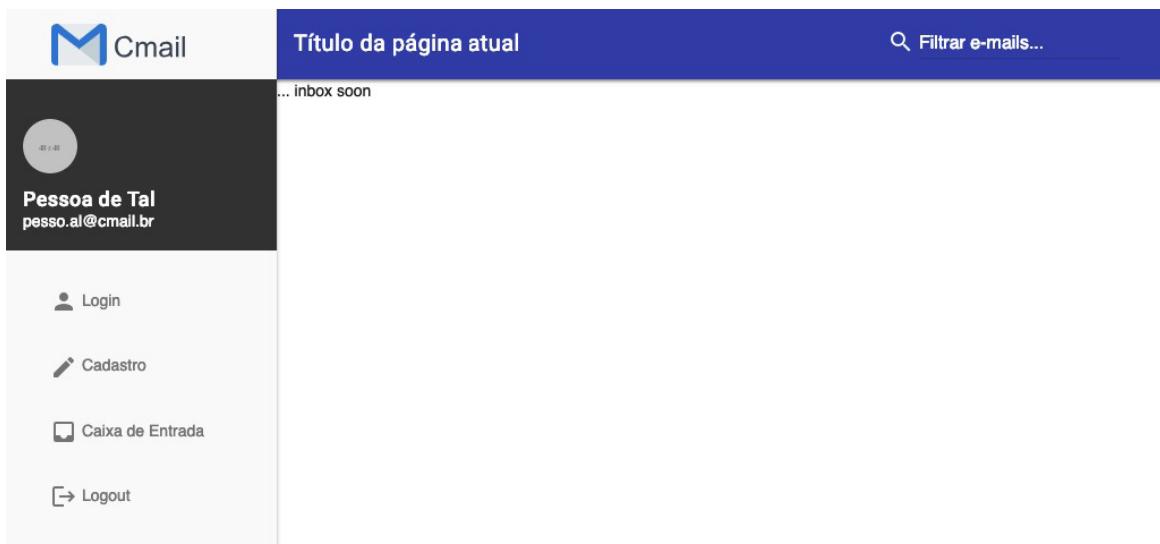


Figura 4.1: CMail - Header básico

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

4.2 PASSO A PASSO COM CÓDIGO

Para que não seja necessário digitar o código dos arquivos de HTML ou CSS, baixe-os a partir do repositório dos arquivos do curso, pasta **01**:

1. Para fazer o estilo da aplicação seguindo o design da Google, vamos usar a biblioteca de CSS [Material Design Lite](#), que também é mantida pelo próprio time da Google. Vamos usar a CDN da biblioteca para facilitar este inicio. Site: <https://getmdl.io/started/>.

No **index** vamos importar a fonte, os ícones e o CSS:

`#./src/index.html`

```
<!doctype html>
<html lang="pt-br">
<head>
  <meta charset="utf-8">
  <title>CMail</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">

  <!-- Material Design -->
  <link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons">
  <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Roboto:300,400,500">
  <link rel="stylesheet" href="https://code.getmdl.io/1.3.0/material.indigo-red.min.css" />
  <!-- ./Material Design -->

</head>
<body>
  <app-root></app-root>
```

```
</body>
</html>
```

2. Adicione o **CSS global** da aplicação:

```
#./src/styles.css
```

```
/* You can add global styles to this file, and also import other style files */
/* Ajuste de altura */
.mdl-layout {
  height: 100vh;
}

/* Ajuste do Menu */
@media (min-width: 1025px) {
  .mdl-layout--fixed-drawer > .mdl-layout__header {
    margin-left: 240px;
    width: calc(100% - 240px);
  }
}
```

3. Na pasta **app** é onde fica tudo que vamos criar, e todo código dos componentes. Então lá identificamos os arquivos base do **AppComponent**, o primeiro a ser carregado na página.

No template de **AppComponent**, percebemos o conteúdo inicial gerado pela CLI; vamos substituir totalmente este conteúdo pelo seguinte HTML:

```
#./src/app/app.component.html
```

```
<div class="mdl-layout mdl-js-layout mdl-layout--fixed-drawer mdl-layout--fixed-header">
  <main class="mdl-layout__content">
    <div class="page-content">
      ... inbox soon
    </div>
  </main>
</div>
```

4. Agora vamos criar nosso primeiro componente, para reutilização de código, **HeaderComponent**. Vamos começar organizando o projeto criando uma pasta `components` e ali criar a pasta `header` que vai conter todos os arquivos necessários para fazer o componente.

```
#./src/app/components/header/header.component.ts
```

```
import { Component } from "@angular/core";

@Component({
  selector: 'cmail-header',
  templateUrl: './header.component.html',
  styleUrls: [
    './header.component.css',
    './header-search.css'
  ]
})
export class HeaderComponent {}
```

Dica: copie dos arquivos do curso o HTML e CSS:

```
#./src/app/components/header/header.component.html

<header class="headerGlobal mdl-layout__header">
  <button class="mdl-layout__drawer-button">
    <span class="material-icons"></span>
  </button>

  <div class="mdl-layout__header-row">
    <span class="headerGlobal__title mdl-layout-title">Título da página atual</span>
    <div class="mdl-layout-spacer"></div>

    <!-- Search -->
    <div class="mdl-textfield mdl-js-textfield mdl-textfield--expandable
      mdl-textfield--floating-label mdl-textfield--align-right is-focused mdl-cell--hide-phone ">
      <!--Para fechar remover is-focused-->
      <label class="mdl-button mdl-js-button mdl-button--icon" for="fixed-header-drawer-exp">
        <span class="material-icons">search</span>
      </label>
      <div class="mdl-textfield__expandable-holder">
        <input class="headerSearch__input mdl-textfield__input" type="text" name="sample" id="fixed
          -header-drawer-exp"
          placeholder="Filtrar e-mails...">
      </div>
    </div>
    <!--./Search-->
  </div>
</header>
<section class="headerGlobal__menuArea mdl-layout__drawer">
  <span class="mdl-layout-title">
    
  </span>

  <header class="headerGlobal__userInfo">
    
    <div class="headerGlobal__userInfoData">
      <h5 class="headerGlobal__userInfoDataName">Pessoa de Tal</h5>
      <span>pesso.al@cmail.br</span>
      <div class="mdl-layout-spacer"></div>
    </div>
  </header>

  <nav class="mdl-navigation">
    <a class="mdl-navigation__link">
      <span class="material-icons">person</span> Login
    </a>
    <a class="mdl-navigation__link">
      <span class="material-icons">create</span> Cadastro
    </a>
    <a class="mdl-navigation__link">
      <span class="material-icons">inbox</span> Caixa de Entrada
    </a>
    <a class="mdl-navigation__link">
      <span class="material-icons">logout</span> Logout
    </a>
  </nav>
</section>
<div class="headerGlobal__menuOverlay"></div>
```

#./src/app/components/header/header.component.css

```
.headerGlobal__title {
```

```
    left: 175px;
}

@media (max-width: 1024px) {
  .headerGlobal {
    display: flex;
    z-index: 50;
    position: relative;
  }
  .headerGlobal__title {
    left: 0px;
  }
}

.headerGlobal__logo {
  max-width: 60%;
}

.headerGlobal__menuArea {
  border: 0;
}

@media (min-width: 1025px) {
  .headerGlobal__menuArea {
    transform: translateX(0);
  }
}

.headerGlobal__menuArea--active {
  transform: translateX(0);
}
.headerGlobal__menuOverlay {
  position: absolute;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background: rgba(0, 0, 0, 0.5);
  pointer-events: none;
  opacity: 0;
  transition: .5s;
  z-index: 4;
}
.headerGlobal__menuArea--active + .headerGlobal__menuOverlay {
  opacity: 1;
  pointer-events: all;
}

.headerGlobal__userInfo {
  background-color: #404040 !important;
  box-sizing: border-box;
  display: flex;
  flex-direction: column;
  justify-content: flex-end;
  padding: 16px;
  height: 151px;
}
.headerGlobal__userInfoAvatar {
  width: 48px;
  height: 48px;
  border-radius: 24px;
}
.headerGlobal__userInfoData {
```

```

color: #ffffff;
padding-top: 14px;
}

.headerGlobal__userInfoDataName {
font-size: 18px;
margin: 0;
}

```

#./src/app/components/header/header-search.css

```

.headerSearch__input::placeholder {
color: #ffffff;
}

```

- Para usar o **HeaderComponent** utilizamos seu selector como uma tag `<cmail-header>` `</cmail-header>` no template de **AppComponent**:

#./src/app/app.component.html

```

<div class="mdl-layout mdl-js-layout mdl-layout--fixed-drawer mdl-layout--fixed-header">
<cmail-header></cmail-header><!-- Aqui inserimos HeaderComponent -->
<main class="mdl-layout__content">
<div class="page-content">
    ... inbox soon
</div>
</main>
</div>

```

Ao abrir o CMail no navegador, temos tela em branco, e ao verificar o console do navegador, percebemos um erro:

```

✖ Uncaught Error: Template parse errors:
  'cmail-header' is not a known element:
  1. If 'cmail-header' is an Angular component, then verify that it is part of this module.
  2. If 'cmail-header' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this
     message. ("<div class="mdl-layout mdl-js-layout mdl-layout--fixed-drawer mdl-layout--fixed-header">
      [ERROR]   <><cmail-header></cmail-header>
      <main class="mdl-layout__content">
        <div class="page-content">
          ")": ng:///AppModule/AppComponent.html@1:2
          at syntaxError (compiler.js:2427)
          at TemplateParser.push../node\_modules/@angular/compiler/fesm5/compiler.js.TemplateParser.parse \(compiler.js:20311\)
          at JitCompiler.push../node\_modules/@angular/compiler/fesm5/compiler.js.JitCompiler.\_parseTemplate \(compiler.js:25857\)
          at JitCompiler.push../node\_modules/@angular/compiler/fesm5/compiler.js.JitCompiler.\_compileTemplate \(compiler.js:25844\)
          at compiler.js:25787
          at Set.forEach (<anonymous>)
          at JitCompiler.push../node\_modules/@angular/compiler/fesm5/compiler.js.JitCompiler.\_compileComponents \(compiler.js:25787\)
          at compiler.js:25697
          at Object.then (compiler.js:2418)
          at JitCompiler.push../node\_modules/@angular/compiler/fesm5/compiler.js.JitCompiler.\_compileModuleAndComponents \(compiler.js:25696\)

```

Figura 4.2: Template parse errors: 'cmail-header' is not a known element

- Para corrigir temos que informar ao módulo principal do Angular, tudo o que criamos. Informamos **HeaderComponent** no **AppModule**:

#./src/app/app.module.ts

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
//Importação e referência ao HeaderComponent

```

```
import { HeaderComponent } from './components/header/header.component';

@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent //HeaderComponent aqui!
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

7. Então, percebemos que ainda falta a **logo do CMail**. Copie do repositório dos arquivos do curso a logo da aplicação, que deverá ficar localizada no seguinte diretório do projeto:
`src/assets/img/logo.svg`.

EXPLORANDO A SINTAXE DOS TEMPLATES ANGULAR

HTML é a linguagem para os templates. Praticamente toda a sintaxe HTML é suportada nos templates. A grande exceção é a tag `<script>`, ela é proibida, assim eliminando o risco de ataques de injeção de scripts. Na prática, ela é ignorada e um aviso é exibido no console do navegador.

Os outros elementos HTML que são suportados porém não fazem sentido usar nos templates dos componentes são: `<html>`, `<body>`, e `<base>`, devido a própria definição de Single Page Application e Web Componentes, onde já temos um `index.html` onde tudo será carregado dentro do seu `body`, e portanto os templates são trechos de página encapsulados em componentes. No mais, todas as outras tags do HTML podem ser utilizadas.

É possível extender o HTML, ou seja, criar novos elementos e atributos, através do recurso de criação de componentes e diretivas do Angular. Estes recursos do framework permitem que a atualização dinâmica dos dados, valores de atributos e elementos seja uma tarefa simples, padronizada e performática.

5.1 INTERPOLAÇÃO NOS TEMPLATES

A interpolação permite incorporar trechos variáveis ou de "script" (com limites) ao texto entre tags e atributos dos elementos HTML.

A sintaxe da interpolação usa como delimitador as chaves duplas `{{...}}`. O valor impresso nesta expressão é convertido para *string*, e pode ser tanto o valor de uma propriedade de uma classe, ou um método da classe que retorna uma *string*, ou até mesmo fazer uma expressão, por exemplo:

Vai imprimir o valor da propriedade `titulo` da classe deste template:

```
 {{titulo}}
```

Vai imprimir o retorno deste método:

```
 {{nomeDoSite()}}
```

Vai calcular a expressão:

```
 {{28+14}}
```

É importante lembrar que aqui o valor sempre será uma **string**. Dentro da expressão de interpolação não é possível acessar objetos globais como `window`, `document`, `console` etc.

Saber inglês é muito importante em TI



cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

5.2 PROPERTY BINDING

A associação via propriedade, é denotada pro um par de colchetes `[prop]` ao redor de algum atributo de componentes ou elemento HTML.

Com ela, conseguimos passar como valor deste atributo objetos inteiros, valores booleanos, propriedades e métodos da classe. Por exemplo:

```
<img [src]="logoUrl">
```

Neste exemplo, `logoUrl` pode ser uma propriedade ou um método getter da classe deste componente em questão.

Ele é o mais recomendado quando precisamos passar dados para valores de atributos nos elementos do *template*.

Um detalhe importante é que, tanto na *property binding* quanto na interpolação, só podemos receber informações, não é possível fazer saída de dados (*output*).

5.3 ONE-WAY DATA-BINDING

Ou "**unidirectional data flow**", ou "**one-way in**", é quando passamos o valor de uma camada para a outra. Pode ser direcionada das seguintes formas:

1) One-way da fonte de dados (classe) até a view:

```
{ {expression} }  
[target] = "expression"
```

2) One-way da view até a fonte de dados:

```
(target) = "statement"
```

5.4 EVENT BINDING

A associação por evento (*event binding*) permite que você escute por determinados eventos como cliques, teclar, associando algum método com este evento ocorrido em algum elemento do template.

A sintaxe do event binding consiste no nome do evento envolvido com parênteses à esquerda do sinal de atribuição (sinal de igual), e a direita do sinal, entre aspas, um método da classe. Por exemplo, um botão que escuta o evento `click` e este ser ao ser disparado, executará método `onSave()` da classe, semelhante a um *callback*:

```
<button (click)="onSave()">Save</button>
```

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

5.5 DIRETIVAS DO TIPO ATRIBUTO "DE FÁBRICA" (BUILT-IN)

Ou "*built-in attribute directives*" escutam e modificam o comportamento e valores de elementos, atributos, propriedades e componentes no HTML. Ou seja, são como funções prontas que o framework nos oferece para configurar e trabalhar nas mais diversas situações de *template*. As mesmas são aplicadas no *template* dentro dos elementos com a mesma sintaxe de um atributo HTML.

As diretivas que já vem no Angular, são disponíveis de acordo com seus módulos. Quando criamos

uma aplicação com a CLI, esta já importa o `BrowserModule` que re-exporta o `CommonModule`, no qual contém uma série de diretivas úteis para trabalhar com lógica nos *templates*.

Pouco a pouco vamos ir conhecendo estas diretivas, e aqui vamos conhecer uma das mais utilizadas a **NgClass**.

5.6 A DIRETIVA NG-CLASS

Geralmente quando precisamos alternar estados de apresentação de um elemento, fazemos isto alterando classes. Esta diretiva é a melhor escolha para gerenciar a inserção ou remoção de um conjunto de classe de um elemento simultaneamente, com base em uma condição booleana.

Para usar, basta adicionar a diretiva `ngClass` como um atributo, porém omo ele receberá um valor dinâmico, deverá ser envolvido com a sintaxe de **bind** (os colchetes `[]`), ficando assim: `[ngClass]`. O valor deste atributo pode ser dado de diversas formas. A que usamos no exercício, passamos um objeto como valor, onde a **chave** do objeto é uma string com o nome da classe, e o **valor** é deve ser um booleano (*true* ou *false*), que bom base nisso a classe é inserida ou não, este valor pode ser gerenciado por algum método ou atributo da classe.

```
<nav [ngClass]="{ 'menu-aberto': isMenuOpen }">  
...  
</nav>
```

No exemplo acima, a classe a ser inserida é `menu-aberto`, e o valor booleano é fornecido pela referência `isMenuOpen`, no qual pode ser uma propriedade da classe, ou um método `getter`.

Outras formas de uso para o `ngClass`:

```
<some-element [ngClass]="'first second'">...</some-element>  
  
<some-element [ngClass]=["first", "second"]>...</some-element>  
  
<some-element [ngClass]={'first': true, 'second': true, 'third': false}>...</some-element>  
  
<some-element [ngClass]= "stringExp|arrayExp|objExp">...</some-element>  
  
<some-element [ngClass]={'class1 class2 class3' : true}>...</some-element>
```

Caso não queira usar o `ngClass`, o class binding pode ser uma alternativa caso a lógica seja apenas adicionar ou remover uma única classe:

```
<!-- alternando a classe "special" usando a propriedade nativa class -->  
<div [class.special]="isSpecial">The class binding is special</div>
```

Referências

1. <https://angular.io/guide/template-syntax>
2. <https://angular.io/guide/template-syntax#template-statements>
3. <https://angular.io/guide/architecture-components#template-syntax>
4. <https://angular.io/guide/attribute-directives>
5. <https://angular.io/api/common/CommonModule>

EXERCÍCIO: DANDO VIDA PARA O MENU MOBILE

6.1 OBJETIVO

Quando a tela for menor que 1026px de largura, o menu lateral fica escondido, porém ao clicar no ícone do menu queremos que ele volte a se tornar visível na tela. Na prática isto será feito pelo CSS, porém o Javascript terá que gerenciar a inserção ou remoção de uma classe do CSS ao clicarmos no ícone, usando o evento **click**.



Figura 6.1: Menu responsivo do CMail

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

6.2 PASSO A PASSO COM CÓDIGO

1. No `<button>` inserimos o evento `(click)` como um atributo, e que receberá como valor a função de *callback* `toggleMenu()`, que será definida posteriormente.

`#./src/app/components/header/header.component.html`

```
<header class="headerGlobal mdl-layout__header">
    <!-- inserção do evento click e a função de callback --->
    <button class="mdl-layout__drawer-button" (click)="toggleMenu()">
        <span class="material-icons"></span>
    </button>

<!-- código posterior omitido --->
```

2. Na classe do componente, criamos uma propriedade privada `_isMenuOpen`, usando o modificador **private**, esta propriedade guardará o valor do estado do menu. Definiremos o método `toggleMenu`, que é o *callback* do evento `click`, este vai alterar o valor estado da propriedade `_isMenuOpen`. Para o template conseguir acessar este valor, definimos também um método **getter**, o `isMenuOpen()`.

`#./src/app/components/header/header.component.ts`

```
import { Component } from "@angular/core";

@Component({
    selector: 'cmail-header',
    templateUrl: './header.component.html',
    styleUrls: [
        './header.component.css',
        './header-search.css'
    ]
})
export class HeaderComponent {
    private _isMenuOpen = false
```

```

get isMenuOpen() {
  return this._isMenuOpen
}

toggleMenu() {
  this._isMenuOpen = !this.isMenuOpen
}

}

```

- Voltando ao template do **HeaderComponent** localizamos a `<section>` do menu lateral e inserimos o atributo do Angular `[ngClass]`, que vai alternar uma classe do CSS, verificando um valor booleano que é retornado pelo *getter* que criamos, **isMenuOpen**:

`#./src/app/components/header/header.component.html`

```

<!-- Código anterior omitido -->
<section class="headerGlobal__menuArea mdl-layout__drawer">
<section class="headerGlobal__menuArea mdl-layout__drawer" [ngClass]="{ 'headerGlobal__menuArea--active': isMenuOpen }">
  <span class="mdl-layout-title">
    
  </span>
<!-- código posterior omitido -->

```

- Um último detalhe, para melhorar a experiência do usuário (UX), vamos adicionar a função **toggleMenu** também quando clicarmos fora da área do menu:

`#./src/app/components/header/header.component.html`

```

<!-- Código anterior omitido -->
<div class="headerGlobal__menuOverlay" (click)="toggleMenu()"></div>

```

WORKSPACES E CONFIGURAÇÕES POR PROJETO

Quando executamos o comando `ng new` da CLI do Angular, estamos na verdade criando um diretório para um **workspace**. Dentro deste diretório, uma estrutura inicial com pastas e arquivos é gerada automaticamente, chamamos esta estrutura de **scaffolding**. Nela dois projetos (aplicações) são criados dentro deste workspace. A aplicação principal (ou padrão) está na pasta `src` e a segunda aplicação está na `e2e`. Porém aonde isto é definido?

Na raiz do diretório do **workspace**, um único arquivo é criado para definir as configurações gerais do workspace e específicas de cada projeto, como por exemplo, opções de comando da CLI, e formatos de build. Este arquivo é o `angular.json`.

7.1 ESTILOS GLOBAIS

No momento em que a estrutura básica do projeto foi criada, dentro da pasta `src` foram gerados o `index.html` e também um arquivo `styles.css`. Porém, é possível notar que este arquivo de CSS não está incluído no `index` por tag `<link>`, porém se adicionarmos estilos neste arquivo de CSS, os mesmos serão interpretados. Como isto é possível?

Os arquivos globais de CSS, são inseridos como uma configuração de projeto, portanto devem ser informados no arquivo `angular.json`.

Dentro da propriedade `"projects"` temos o projeto `cmail`, e nele podemos localizamos navegando nas propriedades `architect/build/options/styles` respectivamente. Assim identificamos uma lista, onde nela é armazenada, neste momento, a localização caminho do único arquivo de estilos globais da aplicação até o momento:

```
"styles": [
  "src/styles.css"
],
```

Caso seja necessário incluir mais estilos globais, basta adicionarmos mais um item na lista, apontando o caminho deste arquivo corretamente.

```
"styles": [
  "src/styles.css",
```

```
"src/assets/css/globalFab.css",
"src/assets/css/tooltip.css",
"src/assets/css/errorMessages.css",
"src/assets/css/newEmail.css"
],
```

Lembrando que a ordem dos estilos importa, pois o Angular vai incluir os arquivos com base na ordem declarada na lista.

Ao inspecionar o `index.html` com o DevTools no navegador, é possível ver como o Angular insere os arquivos. Estes ficam embutidos na tag `<style>` no `<head>` do documento.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

7.2 GETTERS E MODIFICADORES DE ACESSO NO TYPESCRIPT

Propriedades de uma classe no TS por padrão são públicas, permitindo ter seus valores consultados e modificados por quem instancia um objeto à partir desta classe. Da mesma forma, no contexto dos componentes, é possível acessar estas propriedades pelo template do componente.

Porém, caso seja necessário, o TS oferece o modificador de acesso `private`, que deve ser usado à esquerda da declaração da propriedade, deixando a propriedade inacessível externamente. Por convenção também colocamos um underline antes do nome da propriedade para que haja uma sinalização que ela é privada.

```
export class Menu {  
  
    private _isMenuAberto = false;  
  
    isMenuAberto(){  
        return this._menuAberto;  
    }  
}
```

Desta forma, só é possível acessar o valor de `_menuAberto` se for criado um método que retorna o

valor da propriedade. Como no trecho de código anterior, o método `isMenuAberto()`.

Porém, de acordo com padrões de projeto, podemos fazer o encapsulamento desta propriedade criando um método `getter`.

Usando a palavra `get` ao lado de um método, e fazendo este retornar um valor, tornamos este método um `getter`.

```
export class Menu {  
  
    private _isMenuAberto = false;  
  
    get isMenuAberto(){  
        return this._isMenuAberto;  
    }  
  
}
```

A diferença entre um método comum e um método `getter`, é que para invocar um método precisamos usar o parênteses logo após sua referência. Por exemplo:

```
toggleMenu();
```

Um método `getter`, se comporta como uma propriedade, não havendo a necessidade de invocar os parênteses logo após sua referência:

```
export class Menu {  
  
    private _isMenuAberto = false;  
  
    get isMenuAberto(){  
        return this._isMenuAberto;  
    }  
  
    toggleMenu(){  
        //usando o getter aqui  
        return this._isMenuAberto = !this.isMenuAberto  
    }  
  
}
```

Referências

1. <https://angular.io/guide/workspace-config>
2. <https://www.typescriptlang.org/docs/handbook/classes.html#accessors>

CAPÍTULO 8

EXERCÍCIO: FORMULÁRIO PARA ENVIAR EMAIL

8.1 OBJETIVO

Vamos começar a construir a caixa de entrada de emails. Teremos um *floating action button* (fab) que ao ser clicado abrirá o formulário (ou janela) para enviar um email, na parte inferior direita da tela:

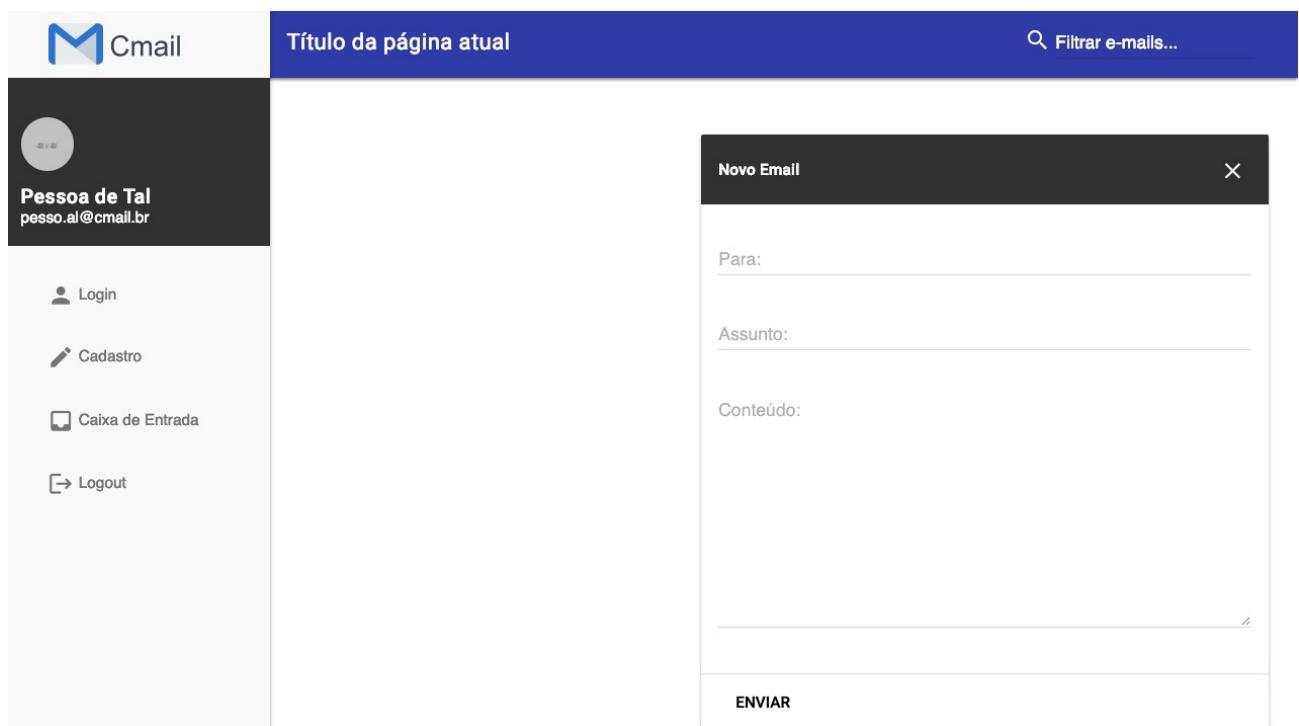


Figura 8.1: Caixa de entrada com o formulário de novo email aberto

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

8.2 PASSO A PASSO COM CÓDIGO

Para que não seja necessário digitar o código dos arquivos de HTML ou CSS, baixe-os a partir do repositório dos arquivos do curso, pasta **03**.

1. A partir dos arquivos do curso, copiaremos e adicionaremos mais 3 arquivos globais de CSS, e os colocaremos na pasta `src/assets/css`. E então vamos inserir eles no projeto através do arquivo **angular.json** linha 25:

`#./angular.json:25`

```
"styles": [  
  "src/styles.css",  
  "src/assets/css/globalFab.css",  
  "src/assets/css/tooltip.css",  
  "src/assets/css/errorMessages.css",  
  "src/assets/css/newEmail.css"  
,
```

Toda vez que fizermos modificações neste arquivo (`angular.json`) é necessário **parar** a execução do **ng serve** e **iniciar novamente**.

2. Vamos adicionar o template do botão e do formulário em **AppComponent**:

`#./src/app/app.component.html:6`

```
... inbox soon  
  
<div class="mdl-grid">  
  <!-- Floating Multiline Textfield -->
```

```

<button class="globalFab tooltip btn mdl-button mdl-js-button mdl-button--fab mdl-button--colored"
    role="tooltip" aria-label="Criar novo email">
    <span class="material-icons">add</span>
</button>

<!-- newEmail--active -->
<form action="#" class="newEmail cmailForm">
    <div class="newEmail__card demo-card-wide mdl-card mdl-shadow--2dp">
        <div class="newEmail__titleArea mdl-card__title">
            <h2 class="newEmail__title mdl-card__title-text">Novo Email</h2>
        </div>
        <div class="newEmail__body mdl-card__supporting-text">
            <!-- Form Fields -->
            <!-- [Para] -->
            <div class="cmailInputForm">
                <div class="mdl-textfield mdl-textfield--floating-label">
                    <input name="para" placeholder=" " required="" type="email" id="para" class="mdl-textfield__input">
                    <span class="mdl-textfield__error">Informar um email é obrigatório!</span>
                    <label class="mdl-textfield__label" for="email"> Para: </label>
                    <span class="mdl-textfield__formline"></span>
                </div>
            </div><!-- ./[Para] -->

            <!-- [Assunto] -->
            <div class="cmailInputForm">
                <div class="mdl-textfield mdl-textfield--floating-label">
                    <input name="assunto" placeholder=" " required="" type="text" id="assunto" class="mdl-textfield__input">
                    <span class="mdl-textfield__error">Informar um assunto é obrigatório!</span>
                    <label class="mdl-textfield__label" for="email"> Assunto: </label>
                    <span class="mdl-textfield__formline"></span>
                </div>
            </div><!-- ./[Assunto] -->

            <!-- [Conteúdo] -->
            <div class="cmailInputForm">
                <div class="mdl-textfield mdl-textfield--floating-label">
                    <textarea class="mdl-textfield__input" type="text" rows="3" id="sample5" placeholder=" ">
                </div>
            </div><!-- ./[Conteúdo] -->

            <!-- ./Form Fields -->
        </div>
        <div class="mdl-card__actions mdl-card--border">
            <button class="mdl-button">
                Enviar
            </button>
        </div>
        <div class="newEmail__topMenu mdl-card__menu">
            <button type="button" class="mdl-button mdl-button--icon">
                <span class="material-icons">close</span>
            </button>
        </div>
    </div>
</form>
</div>

```

3. Seguindo a mesma ideia do menu mobile, vamos criar um método que vai alternar um estado boleando, para depois ser usado como condição para exibir ou esconder o formulário de novo email. Criaremos o método **toggleNewEmailForm** e a propriedade **isNewEmailFormOpen**:

#./src/app/app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'cmail';
  private _isNewEmailFormOpen = false;

  get isNewEmailFormOpen(){
    return this._isNewEmailFormOpen;
  }

  toggleNewEmailForm() {
    this._isNewEmailFormOpen = !this.isNewEmailFormOpen
  }
}
```

4. Voltamos ao template e vamos adicionar o evento click no botão flutuante, e depois no botão de fechar o formulário:

#./src/app/app.component.html:8

```
<!-- Floating Multiline Textfield -->
<button class="globalFab tooltip btn mdl-button mdl-js-button mdl-button--fab mdl-button--colored"
role="tooltip" aria-label="Criar novo email" (click)="toggleNewEmailForm()">
  <span class="material-icons">add</span>
</button>
```

#./src/app/app.component.html:59

```
<button type="button" class="mdl-button mdl-button--icon" (click)="toggleNewEmailForm()">
  <span class="material-icons">close</span>
</button>
```

5. Para finalizar, precisamos usar o **[ngClass]** para alterar a classe do formulário de novo email:

#./src/app/app.component.html:14

```
<form action="#" class="newEmail cmailForm" [ngClass]="{{'newEmail--active': isNewEmailFormOpen}}>
```

EVENT BINDING MAIS A FUNDO

O ***event binding*** pode ser utilizado como uma maneira simples, porém limitada, para trabalhar com formulários.

No formulário podemos escutar o evento `submit` para associar com um método da classe do componente, que poderá prosseguir com a lógica de envio dos dados.

```
<form (submit)="cadastrar()">
<!-- ... -->
</form>
```

Ao submeter um formulário, o comportamento padrão do navegador é recarregar a página, o que não faz sentido para uma SPA. Quando queremos ter este efeito de enviar os dados sem recarregar a página, fazemos uma requisição assíncrona com JavaScript - popularmente conhecida como "AJAX" - para enviar os dados do formulário, fazendo com que a página não recarregue. Porém não é bem o AJAX que impede o navegador recarregar, e sim um método presente no objeto que representa os eventos do JS, este método impede que o evento emitido se propague assim não ocorrendo este comportamento padrão do navegador, este método é o `preventDefault()`.

Porém como ter acesso ao objeto do evento no Angular?

9.1 \$EVENT A REFERÊNCIA AO OBJETO DOS EVENTOS

Quando o evento é gerado, o *handler* de eventos executa a instrução do modelo. A instrução de modelo geralmente envolve um "receptor", que executa uma ação em resposta ao evento.

No momento deste "*bind*" (associação) entre o evento e ação de consequência (*callback*), podemos transmitir informações sobre o evento. Essas informações podem incluir valores de dados, objeto do elemento, um texto ou número, para isto o framework nos fornece a referencia `$event`. Este pode ser utilizado da seguinte maneira:

```
<form (submit)="cadastrar($event)">
<!-- ... -->
</form>
```

O tipo do evento que determina a forma do objeto `$event`. Se o evento for de algum elemento nativo do DOM `$event` será um objeto DOM igual ao do JavaScript, com a conhecida propriedade `target` e com o método `preventDefault()`.

Portando, passando o objeto `$event` como parâmetro do método cadastrar, conseguiremos dentro deste método evitar o recarregamento da janela do navegador.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

9.2 RECEBENDO PARÂMETROS EM UM MÉTODO DE UMA CLASSE

Na classe de um componente podemos criar propriedades e métodos. Para criarmos um método que recebe parâmetros, basta definir nos parênteses da declaração do método um bom nome para representar o parâmetro:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  cadastrar(eventoSubmit){
    console.log(eventoSubmit)
  }
}
```

Podemos também definir o tipo do parâmetro para ajudar a autocompletar na hora de usar métodos ou propriedades deste objeto. Como sabemos que ele é um evento padrão do DOM, podemos definir o tipo com a interface **Event**, que representa qualquer evento do DOM da maneira mais genérica, onde o próprio TypeScript nos fornece, não havendo necessidade de importar uma referência de bibliotecas.

Após definir o nome do parâmetro, adicionamos a direita dois pontos `:` e em seguida o nome do tipo, que pode ser uma classe, interface, um objeto literal, ou os tipos clássicos.

```
import { Component } from '@angular/core';

@Component({
```

```

        selector: 'app-root',
        templateUrl: './app.component.html',
        styleUrls: ['./app.component.css']
    })
export class AppComponent {

    cadastrar(eventoSubmit: Event){
        console.log(eventoSubmit)
    }
}

```

Agora com o objeto recebido por parâmetro e seu tipo definido, podemos executar o método `preventDefault()`, assim impedindo o carregamento do navegador:

```

import { Component } from '@angular/core';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {

    cadastrar(eventoSubmit: Event){
        console.log(eventoSubmit);
        eventoSubmit.preventDefault();
    }
}

```

9.3 OBTENDO O VALOR DE INPUTS COM \$EVENT

Da mesma forma, é possível usar um evento em um elemento `<input>` para obter o valor dele, ou seja, o que um usuário digitou no campo.

Para isto, devemos pensar qual evento contempla melhor este momento, pois sabemos por experiência com JavaScript que um elemento de formulário emite vários eventos ao sofrer alguma interação, como o `keydown`, `change`, `blur` etc. Porém todos eles tem alguma limitação, por isto, o que melhor aborda esta situação é o evento `input`, no qual ocorre tanto quanto um valor é digitado no campo, quanto quando um dado é inserido pela função de colar texto (botão direito colar, ou, `ctrl v`).

Agora que já identificamos o evento JS mais apropriado, podemos pensar como obter o valor do campo. Sabendo que entre as aspas, temos o objeto `$event` disponível, e com isto podemos acessar o elemento que está sofrendo este evento através da propriedade `target`, assim: `$event.target`.

Como `$event.target` nos retorna a referência para um elemento, podemos então acessar à todas as propriedades do DOM deste elemento, inclusive a propriedade `value` que é a referência para o valor digitado/informado no campo específico, portanto usamos a expressão `$event.target.value` para esta atividade.

```
<input type="text" name="assunto" (input)="event.target.value">
```

Já estamos conseguindo acessar o valor do campo, porém não estamos fazendo nada com ele. Para conseguirmos enviar este valor para algum método da classe, precisamos armazenar este valor em uma propriedade da classe, assim permitindo o método em questão acessar esta propriedade e seu valor, no momento que é invocado.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  //criamos a propriedade que guardará o valor de um campo
  assunto = "";

  cadastrar(eventoSubmit: Event){
    eventoSubmit.preventDefault();
    //acessando a propriedade da classe:
    console.log(this.assunto);
  }
}
```

Agora, toda vez que o evento `input` acontecer no elemento `<input>` devemos enviar seu novo valor para a propriedade da classe. Para isto, é possível fazer uma simples expressão de atribuição, como "callback" do evento:

```
<input type="text" name="assunto" (input)="assunto = $event.target.value">
```

9.4 PROPRIEDADES DE UMA CLASSE: TIPOS E POSSIBILIDADES

Para cada campo de um formulário na view do componente, será necessário criar um propriedade da classe para armazenar o valor deste campo. Porém, um formulário pode ser muitos campos, e a quantidade de propriedades ficar muito extensa, e ainda confundir com outras propriedades que não tem relação com um campo do formulário.

Para isto podemos explorar outros formatos para armazenar estes valores. Uma ideia que pode facilitar aqui, é criar uma propriedade que armazena um objeto, e este objeto ter propriedades para cada campo. Por exemplo:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  formCadastro = {
    email: '',
    assunto: ''
  }
```

```

        categoria: '',
        mensagem: '',
        data: ''
    }

    //código omitido
}

```

Desta forma, podemos fazer a associação dos valores da seguinte maneira:

```

<form (submit)="cadastrar($event)">

    <input type="email" name="email" (input)="formCadastro.email = $event.target.value">

    <input type="text" name="assunto" (input)="formCadastro.assunto = $event.target.value">

    <select name="categoria" (change)="formCadastro.categoria = $event.target.value">
        <option value="urgente">Urgente</option>
        <option value="normal">Normal</option>
        <option value="sem-pressa">Sem pressa</option>
    </select>

    <textarea name="mensagem" (input)="formCadastro.mensagem = $event.target.value"></textarea>

    <input type="date" name="data" (input)="formCadastro.data = $event.target.value">

    <button>enviar</button>

</form>

```

O objeto `formCadastro` criado no momento da declaração da propriedade, agora se tornará também o seu modelo, possibilitando com que o compilador do TS faça a verificação de tipos caso alguém o preencha erroneamente.

Agora toda vez que este formulário for enviado, queremos exibir os enviados numa lista ao lado do formulário, ainda na mesma página. Ou seja, precisamos de uma estrutura para armazenar uma lista de objetos "cadastro" preenchidos pelo formulário, ou seja, uma propriedade da classe do tipo lista, que tanto no JS quanto no TS são os *Arrays*. Para inicializarmos uma propriedade como array, basta atribuirmos um par de colchetes como seu valor:

```

import { Component } from '@angular/core';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {

    //tipo array
    listaCadastro = [];

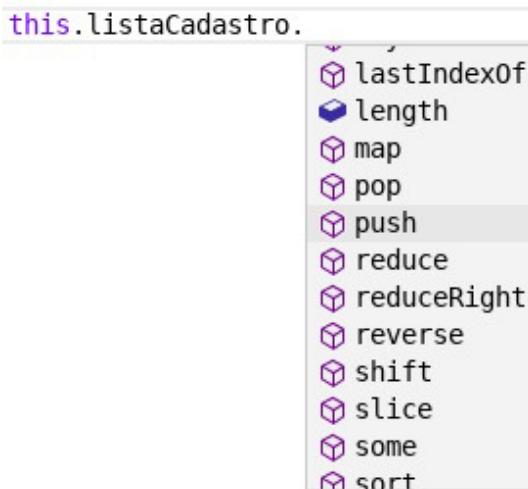
    formCadastro = {
        email: '',
        assunto: '',
        categoria: '',
        mensagem: '',
        data: ''
    }
}

```

```
}

//código omitido
}
```

Agora a cada envio do formulário podemos enviar o objeto cadastro para a lista de cadastro, pois como a mesma é um *array*, todos os métodos deste tipo estarão disponíveis para uso:



```
//código anterior omitido
cadastrar(eventoSubmit: Event){
  eventoSubmit.preventDefault();

  this.listaCadastro.push(this.formCadastro);

  console.log(this.listaCadastro);

}
//código posterior omitido
```

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

9.5 VALIDAÇÃO BÁSICA COM HTML5

Quando trabalhamos com formulários, sempre temos que dar uma atenção mínima à entrada de dados, principalmente se os campos obrigatórios estão sendo preenchidos. A maneira mais simples de fazer isto é utilizando os próprios recursos do HTML5, como tipos específicos para os campos `input`, e atributos como `required`, `min`, `max` etc.

Mesmo o formulário sendo impedido de ser submetido pelo JavaScript, a validação nativa do HTML5 é disparada, sendo possível verificar as mensagens nativas de validação do navegador para cada tipo de erro.

9.6 ITERANDO EM LISTAS NO TEMPLATE *NGFOR

Sabemos que toda vez que o formulário é submetido, estamos guardando os dados na propriedade `listaCadastro` e exibindo no *console* do navegador. Porém, como fazer para visualizar estes dados no template?

Como estamos em uma estrutura de lista, para acessar cada objeto nesta lista, e a partir dos dados deste objeto adicionarmos uma tag HTML para exibir os dados, vamos precisar de uma estrutura de laço, ou seja um `for`. Porém como seria possível fazer um `for` dentro de um arquivo HTML?

O Angular, contém uma **diretiva estrutural de template**, chamada `*ngFor`, e é do tipo estrutura pois ela modifica a estrutura do documento ao ser utilizada, estes tipos de diretiva começam com asterisco no seu nome. Neste caso do `*ngFor`, ele repete o elemento HTML até o fim do laço.

O `*ngFor`, é um `for of` do JavaScript, ou seja, temos que declarar uma variável que vai representar o objeto daquela posição da lista, de qual lista que será percorrida. A declaração do `for of` é bem mais simples, e podemos usar sempre que tivermos um laço que percorre do início ao fim da lista, e itera de 1 em 1.

```
<ul>
  <li *ngFor="let cadastro of listaCadastro">
    {{cadastro.email}}, {{cadastro.assunto}}, {{cadastro.data}}
  </li>
</ul>
```

Aqui na declaração do `*ngFor`, em `let cadastro` estamos criando uma variável `input` de template (*template input variable*), que vai existir apenas no escopo do `*ngFor`, ou seja, entre as tags ``.

9.7 CHANGE DETECTION - DETECÇÃO DE MUDANÇAS NO ANGULAR

Ao coletar dados do formulário e exibir os mesmos, ao lado do formulário, em uma lista, percebemos que a cada envio do formulário a lista é atualizada automaticamente na `view` com o novo objeto que guarda os dados do formulário, sem a necessidade de atualizar a página ou de fazer qualquer outra ação, isto ocorre pois o Angular tem um mecanismo de detecção de mudanças.

Este mecanismo é acionado em alguns momentos diferentes, estes momentos são: quando um evento assíncrono é disparado pela `view`, ou seja, os próprios eventos do navegador `input`, `click`, `submit`, `keydown`, como disparo de requisições `XHR` (`fetch` ou `XHRHttpRequest`). O segundo é quando alguma função temporal é disparada, como `setInterval` ou `setTimeout`. A terceira é quando alguma propriedade de uma classe recebe um novo valor.

Neste caso, estamos fazendo os eventos `input` e `submit`, que após ocorrer o Angular atualiza a `view` para garantir que ela esteja sempre atualizada com novos dados.

Aqui agora temos apenas um problema. A cada `submit` estamos enviando sempre o mesmo objeto para a lista. No formulário estamos sempre alterando o mesmo objeto, ou seja, a mesma referência de memória. Para corrigir este problema, devemos antes de adicionar o objeto na lista, criar um novo objeto a partir dos dados do objeto armazenado na propriedade `cadastro` da classe.

```
//código anterior omitido
cadastrar(eventoSubmit: Event){

  eventoSubmit.preventDefault();

  let novoObjeto = {
    email: this.formCadastro.email,
    assunto: this.formCadastro.assunto,
    categoria: this.formCadastro.categoria,
```

```
mensagem: this.formCadastro.mensagem,  
data: this.formCadastro.data  
}  
  
//push do novo objeto  
this.listaCadastro.push(novoObjeto);  
  
console.log(this.listaCadastro);  
  
}  
//código posterior omitido
```

Desta forma, a cada vez que o método cadastrar é invocado, uma nova variável novoObjeto é declarada, portanto uma nova referência de memória, e seu valor preenchido com um objeto que recebe os dados da propriedade da classe que é preenchida pelo formulário.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

EXERCÍCIO: ENVIANDO EMAILS DO JEITO MAIS SIMPLES DO UNIVERSO

10.1 OBJETIVO

Agora vamos fazer funcionar o formulário para escrever um novo email. Devemos conseguir preencher os dados em cada campo do formulário, coletar eles, e exibir os emails em uma lista bem simples na caixa de entrada:

The screenshot displays a web-based email application with a dark-themed interface. On the left, there's a sidebar with user information ('Pessoa de Tal' and email 'pesso.al@cmail.br'), navigation links ('Login', 'Cadastro', 'Caixa de Entrada', 'Logout'), and a list of recent emails with their recipient and subject.

The main area shows the 'Título da página atual' (Page title) at the top, followed by a search bar ('Filtrar e-mails...'). Below this, a 'Novo Email' (New Email) form is open. The 'Para:' field contains 'bob@cmail.br'. The 'Assunto:' field contains 'Vamos no parque?'. The 'Conteúdo:' field is empty. At the bottom of the form is a blue 'ENVIAR' (Send) button.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

10.2 PASSO A PASSO COM CÓDIGO

1. Na classe de **AppComponent** vamos criar os atributos **emailList** que será inicializada com um *array* vazio para ter o tipo inferido, e **email** que receberá um objeto literal com as propriedades que guardarão os valores dos campos do formulário de novo email:

```
#./src/app/app.component.ts

import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  private _isNewEmailFormOpen = false;
  emailList = [];
  email = {
    destinatario: '',
    assunto: '',
    conteudo: ''
  }

  get isNewEmailFormOpen(){
    return this._isNewEmailFormOpen;
  }

  toggleNewEmailForm() {
    this._isNewEmailFormOpen = !this.isNewEmailFormOpen
  }
}
```

2. Agora temos que relacionar cada campo do formulário com a propriedade equivalente no objeto armazenado no atributo **email** da classe **AppComponent**. Vamos fazer isto usando o evento **input**:

```
#./src/app/app.component.html:24
```

```
<input name="para" placeholder=" " required=""
       type="email" id="para" class="mdl-textfield__input"
       (input)="email.destinatario = $event.target.value" [value]="email.destinatario"
>
```

```
#./src/app/app.component.html:36
```

```
<input name="assunto" placeholder=" " required=""
       type="text" id="assunto" class="mdl-textfield__input"
       (input)="email.assunto = $event.target.value" [value]="email.assunto"
>
```

```
#./src/app/app.component.html:48
```

```
<textarea class="mdl-textfield__input" rows="3" id="conteudo" placeholder=" "
          (input)="email.conteudo = $event.target.value" [value]="email.conteudo">
</textarea>
```

3. Precisamos também fazer com que o formulário, ao ser submetido, salve os dados dentro de `emailList`. Para isto utilizaremos o evento `submit` e associá-lo com um método da classe que criaremos, `handleNewEmail`, e este método vai inserir os dados na lista:

```
#./src/app/app.component.ts
```

```
//código anterior omitido
handleNewEmail(event: Event) {
  event.preventDefault();

  this.emailList.push(this.email)

  this.email = {
    destinatario: '',
    assunto: '',
    conteudo: ''
  }
}
//código posterior omitido
```

```
#./src/app/app.component.html:14
```

```
<form (submit)="handleNewEmail($event)" action="#" class="newEmail cmailForm" [ngClass]="{{ 'newEmail--active': isNewEmailFormOpen }}"
>
```

4. Por último, precisamos percorrer pela `emailList` no template, e para isto usamos a diretiva estrutural `*ngFor`, que ao iterar pela lista também injeta HTML no template:

```
#./src/app/app.component.html:6
```

```
<div class="mdl-grid">
<ul>
  <li *ngFor="let email of emailList">
    <strong>Para:</strong>{{email.destinatario}} <br>
    <strong>Assunto:</strong>{{email.assunto}}
  </li>
</ul>
```

```
</ul>
</div>
```

TEMPLATE DRIVEN FORMS

Na maior parte das aplicações que desenvolvemos temos que lidar com formulários, pois através deles permitimos que usuários criem suas contas, efetuem login, atualizem seus dados, insiram informações confidenciais e executem outras muitos tipos de tarefas que exigem inserção de dados.

O Angular fornece duas abordagens diferentes para lidar com a entrada de dados inseridas pelo usuário por meio de formulários, a de **formulários reativos** e a **orientados a template**. Ambos capturam eventos de entrada do usuário da visualização, validam a entrada do usuário, criam um modelo de formulário e um modelo de dados para atualizar e fornecem uma maneira de rastrear alterações.

Formulários reativos e orientados a template processam e gerenciam dados de formulários de maneira diferente. Cada um oferece diferentes vantagens.

No geral:

Os formulários orientados por modelos são úteis para adicionar um formulário simples à um aplicativo, como um formulário de inscrição para uma lista de e-mail, ou uma newsletter. Eles são fáceis de adicionar em uma aplicação, mas não são dimensionados como formulários reativos. Se você tiver requisitos de forma básicos e lógica que possam ser gerenciados somente no HTML, use formulários orientados à template.

As formas reativas são mais robustas: são mais escalonáveis, reutilizáveis e testáveis. Se os formulários forem uma parte essencial de seu aplicativo ou se você já estiver usando padrões reativos para criar seu aplicativo, use formulários reativos.

O framework do Angular dá suporte para lidar com todas as atividades envolvidas com formulários, como *two-way data binding*, rastreio de mudanças, validação, e erros.

Para começar usar a abordagem de formulários via template, precisamos importar na aplicação o módulo que trás uma série de classes, diretivas e eventos para trabalhar com o formulário, ou seja, o `FormsModule`.

11.1 MÓDULOS ANGULAR

Os módulos do Angular foram criados para organizar a aplicação. Com os módulos onde podemos "isolar" um conjunto de componentes, diretivas, classes, serviços, rotas etc, criando contextos para que

possam ser carregados sob demanda através do *lazy-load*, ou globalmente, e assim organizarmos estes contextos como bibliotecas, ou páginas da aplicação, ou *plugins* etc.

Um módulo Angular, tecnicamente falando, é uma classe decorada com `@NgModule`, e através disso, declaramos a existência de componentes, diretivas ou pipes, e também podemos exportá-los para além do escopo do módulo, também podemos importar outros módulos, registrar serviços e rotas etc.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

11.2 O MÓDULO PRINCIPAL DA APLICAÇÃO APPMODULE

Quando a `CLI` criou um novo projeto Angular, já é feita a estrutura mínima necessária para a aplicação funcionar, e um primeiro módulo é criado, o `AppModule`. Este é o primeiro e principal módulo da aplicação, ele declara a existência do `AppComponent` e informa que este é o primeiro componente a ser carregado no `index.html`, através da propriedade `bootstrap` de `@NgModule`, também importa o `BrowserModule`, que carrega serviços específicos para exibição do DOM.

O `AppModule` que inicia a árvore de dependências e contextos da aplicação. Caso a aplicação for simples e com poucos componentes, apenas este módulo será o suficiente para o funcionamento da aplicação.

11.3 UTILIZANDO O MÓDULO DE FORMULÁRIOS

Para utilizarmos todos os recursos disponíveis para a abordagem de formulários via template, é necessário importar `FormsModule` na aplicação, para isto, é necessário localizar o módulo da página que estamos, no caso aqui `AppModule`.

Na propriedade `imports` de `@NgModule` informamos a lista de módulos que pertencem a este contexto, portanto basta adicionar a referência da classe `FormsModule`, lembrando que a mesma vem do pacote `@angular/forms`:

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
//Referência ao pacote da classe
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule //importamos aqui!
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

11.4 VARIÁVEIS DE REFERÊNCIA DE TEMPLATE

Quando trabalhamos com a abordagem de formulários de template, precisamos com alguma frequência consultar detalhes dos elementos, como o valor, id, classes etc, ou seja, precisamos ter acesso ao objeto do elemento HTML.

Em uma aplicação **não Angular**, em um código JavaScript puro, uma forma de obter o objeto de um elemento é através da função `querySelector`, **por exemplo**:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Objetos no JS :)</title>
</head>
<body>
  <input value="123" id="numero">
  <script>
    let numero = document.querySelector('numero');
    console.log(numero.value);
  </script>
</body>
</html>

```

No caso de um template Angular, podemos fazer o equivalente, desta maneira:

```

import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <input #numero value="123" id="numero">
    {{numero.value}}
  `
})
export class AppComponent {}

```

Perceba que é exibido o valor de uma variável `numero`, que não é uma propriedade da classe do componente, e sim uma variável de referência de template, ou seja, existe apenas no escopo do template do componente.

Para declarar uma referência ao elemento, basta colocarmos a **hashtag** dentro da tag do elemento e definirmos um nome qualquer:

```
<input #numero value="123" id="numero">
```

Neste exemplo criamos a referência `numero`, e com isto podemos usá-la para acessar as propriedades do objeto do elemento em qualquer momento, e apenas no template.

Caso seja declarada uma variável de referência de template com o mesmo nome de uma propriedade da classe, a prioridade no *template* é para as referências.

Este recurso faz parte da sintaxe dos **templates Angular**, e não é algo exclusivo do `FormsModule`, porém é frequentemente usado com a abordagem de formulários via template, para auxiliar com mensagens de validação dos campos. As variáveis de referência de template podem ser usadas a qualquer momento.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

11.5 ENVIANDO OS DADOS DO FORMULÁRIO

Uma das primeiras coisas que percebemos ao importar o `FormsModule` é que o envio do formulário começa a ignorar a validação nativa do HTML. Isto ocorre pois o `FormsModule` quando carregado, modifica as tags `<form>` dos templates, e adiciona um atributo `novalidate`, para que possamos adicionar nossas próprias mensagens de validação customizadas.

Para pelo menos evitar que o formulário seja submetido em branco, podemos verificar se o formulário está válido, porém como obter este `status`? Este dado provavelmente deve estar no objeto do

elemento `<form>` , portanto criando uma referência na tag e interpolando esta referência no template podemos ter algumas respostas.

```
<form #formEmail (submit)="enviar($event)">
{{formEmail}}
```

Isto exibirá `[object HTMLElement]` no navegador, ou seja, o tipo nativo do objeto do elemento, exibido como um texto. E fazendo o exercício de buscar a tag `<form>` com `querySelectorAll` no console do navegador, percebemos que o objeto do elemento não contém nenhuma informação se este formulário está válido ou não, dificultando a lógica que precisamos desenvolver.

NgForm - formulários Angular

O `FormsModule` , trás uma série de recursos para formulários, e um deles é justamente adicionar mais poderes ao objeto do formulário, modificando seu tipo `HTMLElement` para um `NgForm` .

Para isto, a variável de referência de template, criada para o formulário, vai receber uma atribuição de valor, como se fosse um atributo HTML, onde o valor será `ngForm` .

Assim verificamos que o tipo do objeto do formulário mudou, quando interpolamos a variável de template, que passa a ser um `[object Object]` , ou seja, um objeto customizado, não nativo do HTML.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <form #formEmail="ngForm" (submit)="enviar($event, formEmail)">
      {{formEmail}}
      <label for="email">Assine!</label>
      <input required type="email" id="email" placeholder="nome@seumail.com">
      <button>enviar</button>
    </form>
  `
})
export class AppComponent {

  enviar(eventoSubmit: Event, form){
    eventoSubmit.preventDefault();
    console.log(form);
  }
}
```

Porém como podemos ver os detalhes do objeto para entender se neste objeto contém algo sobre o status do formulário? Uma maneira é passar a variável como parâmetro no método `enviar` , e dentro método imprimir seus detalhes no console do navegador.

```

▼ NgForm {submitted: true, _directives: Array(1), ngSubmit: EventEmitter, form: FormGroup} ⓘ
  control: (...)

  controls: (...)

  dirty: (...)

  disabled: (...)

  enabled: (...)

  errors: (...)

  ▶ form: FormGroup {validator: null, asyncValidator: null, _onCollectionChange: f, pristine: ...}

  formDirective: (...)

  invalid: (...)

  ▶ ngSubmit: EventEmitter {_isScalar: false, observers: Array(1), closed: false, isStopped: true}

  path: (...)

  pending: (...)

  pristine: (...)

  status: (...)

  statusChanges: (...)

  submitted: true

  touched: (...)

  untouched: (...)

  valid: (...)

  value: (...)

  valueChanges: (...)

  ▶ _directives: [NgModel]

  ▶ __proto__: ControlContainer

```

Figura 11.1: Detalhes do NgForm

Perceba a quantidade de informações e recursos agora disponíveis para trabalharmos o formulário!

Utilizando o editor de texto Visual Studio Code temos algumas vantagens, como por o recurso de *autocomplete* quando escrevemos código. Na classe do componente, no parâmetro que recebe o objeto do formulário, podemos informar seu tipo, o `NgForm`. Com isto é possível agora verificar não apenas as propriedades do objeto do tipo `NgForm` mas também seus métodos. Por exemplo, podemos usar o método `form.resetForm()`, que vai limpar o valor e estado de validação de todos os campos do formulário de uma vez só, muito útil após os dados serem enviados com sucesso.

Também, podemos fazer uma condição, onde se o formulário estiver inválido, o método não segue a execução, utilizando a propriedade `invalid`, conseguimos fazer a lógica para a condição, **por exemplo:**

```

enviar(eventoSubmit: Event, form: NgForm){
  eventoSubmit.preventDefault();
  console.log(form);

  if(form.invalid) return;

  form.resetForm(form);
}

```

Porém, desta forma o formulário está válido, e a propriedade `invalid` está armazenando o valor `false`. Isto acontece porque ainda não temos associado nenhum `control` (campo de formulário) à este formulário em específico, pois apenas colocar elementos de formulário dentro do elemento `<form>` não é o suficiente, e podemos verificar que não existem `controls` na propriedade de `NgForm` de mesmo nome. Portanto, para associar campos de formulário com seu devido `NgForm`

devemos em cada campo do formulário usar uma diretiva especial, a `NgModel`.

11.6 NGMODEL, A DIRETIVA BIDIRECIONAL

Da mesma maneira que modificamos o objeto do elemento `<form>`, o mesmo será necessário para cada campo do formulário. Vamos ter que adicionar uma variável de referência de template e atribuir com o tipo `ngModel`. Porém ao fazer isto, um erro é exibido no console do navegador:



Figura 11.2: Detalhes do NgForm

Isto acontece pois já temos que definir onde vamos guardar os dados do campo, portanto devemos declarar uma propriedade na classe do componente, que receberá o dado e então usar a **diretiva ngModel** para fazer a associação.

```
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-root',
  template: `
    <form #formEmail="ngForm" (submit)="enviar($event, formEmail)">
      {{formEmail}}
      <label for="email">Assine!</label>
      <input
        name="email"
        [(ngModel)]="emailAddress"
        #email="ngModel"
        required
        type="email"
        id="email"
        placeholder="nome@seumail.com">
      <button>enviar</button>
    </form>
  `
})
export class AppComponent {

  emailAddress = '';

  enviar(eventoSubmit: Event, form: NgForm){
    eventoSubmit.preventDefault();
    console.log(form);

    if(form.invalid) return;

    form.resetForm(form);
  }
}
```

Um detalhe importante, é que a diretiva `[(ngModel)]` exige o atributo `name` do HTML definido no campo do formulário, pois é partir deste nome que `NgForm` guardará a referência do objeto do campo informado dentro de sua propriedade `controls`.

Neste momento você deve estar achando um pouco estranha a sintaxe da diretiva `[(ngModel)]`. Vamos entender um pouco mais.

Banana in a box `[]`: a sintaxe de input e output

Quando precisamos exibir no *template* de um componente Angular um valor que está na classe, armazenado em um atributo ou no retorno de um método, ou dentro de uma variável de referência de template ou em uma variável de input de template, usamos a sintaxe de interpolação `{}{}`, que fará com que o valor seja sempre um texto. Caso seja necessário passar este valor para um atributo de um elemento, usamos a sintaxe de associação unidirecional, onde envolvemos o atributo com um par de colchetes `[]`, **por exemplo:**

```
<input [value]="email" name="email">
```

Aqui o atributo `value` do elemento `input` está associado com o valor da propriedade `email` da classe do seu respectivo componente.

Agora se fosse necessário atualizar este valor pelo que foi digitado no campo, teremos que usar algum evento para fazer a atribuição, **por exemplo:**

```
<input (input)="email = $event.target.value" [value]="email" name="email">
```

Ou seja, a sintaxe para inserção de dados caracterizada por um evento é dada por um par de parênteses envolvendo o nome do evento, e para a saída de dados é caracterizada pelo par de colchetes no atributo onde o valor será exibido. Portanto é desta origem que a diretiva `[(ngModel)]` tem esta notação peculiar, para indicar que ela faz tanto associação de dado do template para a classe, como da classe para o template, ou seja, uma **associação dados em duas direções** (bidirecional), a conhecida expressa *two-way data-binding*.

11.7 O EVENTO IDEAL PARA FORMULÁRIOS (NGSUBMIT)

Além de diretivas e classes, o `FormsModule` também inclui um evento para os formulários. Este evento serve para substituir o evento nativo `(submit)`, e assim nos poupa de ter que tratar o problema do comportamento padrão da tag `<form>` de recarregar a janela do navegador, com `preventDefault` ao submetermos um formulário. Então a tag `<form>` e o método `enviar` vão ficar assim:

```
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-root',
```

```

template: `
<form #formEmail="ngForm" (ngSubmit)="enviar(formEmail)">
  <label for="email">Assine!</label>
  <input
    name="email"
    [(ngModel)]="emailAddress"
    #email="ngModel"
    required
    type="email"
    id="email"
    placeholder="nome@seumail.com">
  <button>enviar</button>
</form>
`)

export class AppComponent {
  emailAddress = '';
  enviar(form: NgForm){
    console.log(form);
    if(form.invalid) return;
    form.resetForm(form);
  }
}

```

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

11.8 EXIBINDO MENSAGENS DE VALIDAÇÃO

Da mesma forma que tivemos acesso ao objeto do formulário através da criação da variável de referência na tag `<form>`, agora com a referência também criada para o campo do formulário conseguimos acessar o objeto dele, que agora é um `control` do tipo `ngModel`.

Este objeto contém várias propriedades que podem nos ajudar na lógica de validação e também na exibição de mensagens. Abaixo **algumas** propriedades e seus tipos/retornos:

```

name: string | null
value: any

```

```

valid: boolean | null
invalid: boolean | null
pending: boolean | null
disabled: boolean | null
enabled: boolean | null
errors: ValidationErrors | null
pristine: boolean | null
dirty: boolean | null
touched: boolean | null
status: string | null
untouched: boolean | null

```

Podemos consultar os detalhes sobre quando cada uma recebe um novo valor, através da documentação do tipo `AbstractControlDirective`, que é implementado por `NgModel`.
<https://angular.io/api/forms/AbstractControlDirective>

No template do componente vamos fazer uma interpolação em uma de suas propriedades que vamos usar para a condição de exibição da mensagem de validação:

```

<form #formEmail="ngForm" (ngSubmit)="enviar(formEmail)">
  <label for="email">Assine!</label>
  <input name="email" [(ngModel)]="emailAddress" #email="ngModel" required type="email" id="email"
placeholder="nome@seumail.com">

  {{email.invalid}}

  <button>enviar</button>
</form>

```

Agora podemos criar classes que exibem ou escondem o elemento que elas contiverem:

```

.error-msg {
  display: none;
}
.is-invalid {
  display: inline-block;
}

```

E no HTML vamos criar agora uma mensagem que é exibida ou escondida com base na condição armazenada em `email.invalid`, usando a diretiva `[ngClass]`:

```

<form #formEmail="ngForm" (ngSubmit)="enviar(formEmail)">
  <label for="email">Assine!</label>
  <input name="email" [(ngModel)]="emailAddress" #email="ngModel" required type="email" id="email"
placeholder="nome@seumail.com">

  {{email.invalid}}

  <span class="error-msg" [ngClass]="{'is-invalid': email.invalid}">
    Preencha um email!
  </span>
  <button>enviar</button>
</form>

```

Porém desta forma a mensagem está sempre sendo exibida. Vamos adicionar à condição a propriedade `touched` que muda seu valor quando o campo é tocado.

<span

```

class="error-msg"
[ngClass]="{'is-invalid': email.invalid && email.touched}" >
Preencha um email!
</span>

```

No componente, podemos adicionar mensagens no console para verificar a validação, e o resultado final seria algo como:

```

import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-root',
  template: `
    <form #formEmail="ngForm" (ngSubmit)="enviar(formEmail)">
      <label for="email">Assine!</label>
      <input
        name="email"
        [(ngModel)]="emailAddress"
        #email="ngModel"
        required
        type="email"
        id="email"
        placeholder="nome@seumail.com">

      {{email.invalid}}
      <span class="error-msg" [ngClass]="{'is-invalid': email.invalid}">
        Preencha um email!
      </span>
      <button>enviar</button>
    </form>
    `

  , styles: [
    .error-msg {
      display: none;
    }
    .is-invalid {
      display: inline-block;
    }
  ]
})
export class AppComponent {

  emailAddress = '';

  enviar(form: NgForm){
    if(form.invalid) {
      console.error('form invalido');
      return;
    }
    console.log('form valido');
    form.resetForm(form);
  }
}

```

11.9 DIRETIVA EMAIL PARA VALIDAÇÃO DE CAMPO TIPO EMAIL

Perceba que desta forma, quando o campo do tipo email é preenchido de qualquer forma, ele se torna

válido. Porém o formato de email ("user@domain") não é verificado. Para voltar a validação do campo email da mesma maneira com que a validação nativa do HTML faz, basta fazermos uso de uma diretiva que também está disponível no `FormsModule`, a diretiva `email`. Para usar, basta escrever `email` dentro do campo que você quer que a validação seja feita, **por exemplo**:

```
<input email>
```

Para o exemplo que estamos usando, ficaria desta maneira:

```
<input  
email  
name="email"  
[(ngModel)]="emailAddress"  
#email="ngModel"  
required  
type="email"  
id="email"  
placeholder="nome@seumail.com">
```

Referências

1. <https://angular.io/guide/forms#template-driven-forms>
2. <https://angular.io/guide/ngmodules>
3. <https://angular.io/guide/template-syntax#html-attribute-vs-dom-property>
4. <https://angular.io/api/forms/NgModel>
5. <https://angular.io/api/forms/AbstractControlDirective>

EXERCÍCIO: VALIDAÇÃO E VARIÁVEIS DE TEMPLATE

12.1 OBJETIVO

Queremos evitar que um email seja enviado sem preencher ao menos o **email de destino** e **assunto**. Portanto precisamos tratar os campos obrigatórios do formulário e também exibir as mensagens de validação abaixo dos respectivos campos.

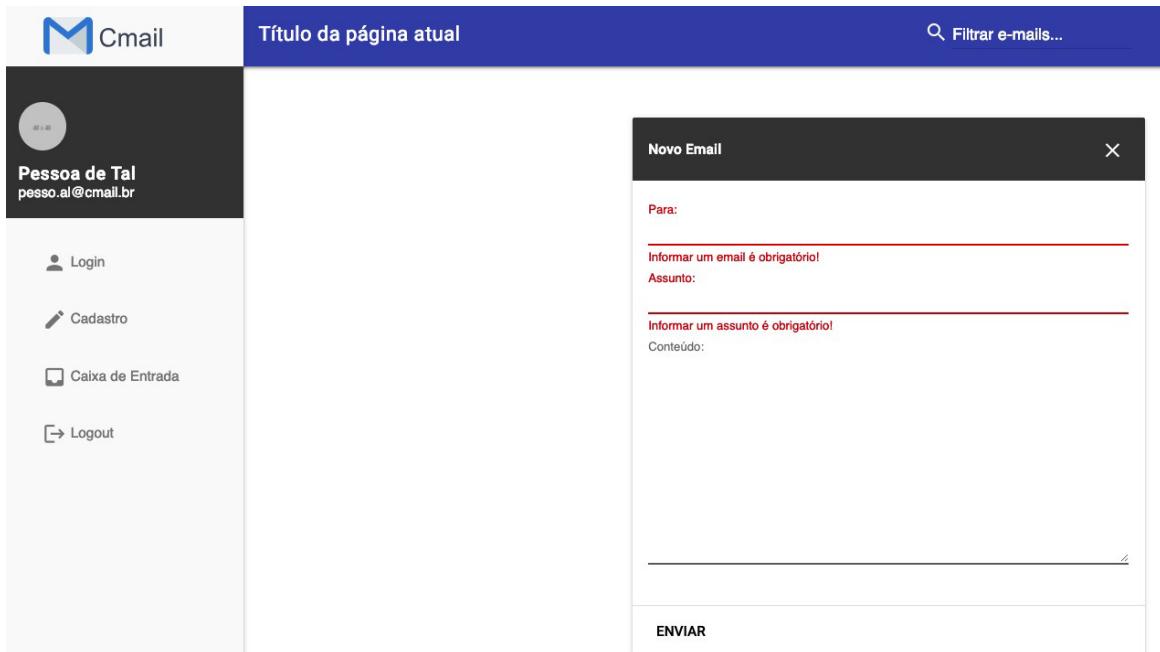


Figura 12.1: Validação via template

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

12.2 PASSO A PASSO COM CÓDIGO

1. Primeiro precisamos pegar a **referência** do formulário, para obter seu status, e usar isto para impedir que ele seja submetido caso esteja inválido. Para isto vamos criar uma variável de template `#formEmail` (ou uma referência), porém para que conseguirmos acessar ela como um formulário Angular, vamos associar seu valor com o tipo `ngForm` :

`#./src/app/app.component.html`

```
<form  
  #formEmail="ngForm"  
  class="newEmail cmailForm"  
  [ngClass]="{ 'newEmail--active': isNewEmailFormOpen }"  
  (submit)="handleNewEmail($event)">
```

Com isto vamos ter um problema de compilação do Angular, pois para usar o tipo `ngForm` no template, é necessário importar sua referência, que é fornecida com o módulo de formulários.

2. Vamos importar o **FormsModule** em **AppModule**:

`#./src/app/app.module.ts`

```
import { BrowserModule } from '@angular/platform-browser';  
import { NgModule } from '@angular/core';  
  
import { AppComponent } from './app.component';  
import { HeaderComponent } from './components/header/header.component';  
  
import { FormsModule } from "@angular/forms";
```

```

@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

3. Agora vamos enviar a referência do `#formEmail` como parâmetro do método de *callback* enviado no evento *submit*. Porém, com o **FormsModule** importado, temos vários outros recursos para trabalhar com formulários. Com isto vamos usar o **ngSubmit** que facilitará a tarefa:

`#./src/app/app.component.html`

```

<form
  #formEmail="ngForm"
  class="newEmail cmailForm"
  [ngClass]="{ 'newEmail--active': isNewEmailFormOpen }"
  (ngSubmit)="handleNewEmail(formEmail)">

```

4. Vamos ajustar o método `handleNewEmail` para receber um formulário Angular agora, ao invés de um evento:

`#./src/app/app.component.ts`

```

//código anterior omitido
handleNewEmail(formEmail: NgForm) {

  if(formEmail.invalid) return;

  this.emailList.push(this.email)

  this.email = {
    destinatario: '',
    assunto: '',
    conteudo: ''
  }

  formEmail.reset();

}
//código posterior omitido

```

Definindo o tipo do parâmetro, o VSCode já importará a referência para o `NgForm`, certifique-se que isto ocorreu na linha 2 do arquivo:

`#./src/app/app.component.ts:2`

```
import { NgForm } from '@angular/forms';
```

Porém ainda não será o suficiente para o formulário saber quais campos estão inválidos. Precisamos também associar eles com uma diretiva especial para isto, a **NgModel**.

5. Para cada campo do formulário, criaremos uma variável de template que receberá como valor **ngModel**. Vamos ter uma outra vantagem de usar a **ngModel**, ela já faz o two-way data-binding (*input* e *output* dos dados), e podemos retirar o evento (*input*) e a associação [*value*] :

Input para:

```
#./src/app/app.component.html
```

```
<input  
    #destinatario="ngModel"  
    [(ngModel)]="email.destinatario"  
    required type="email" name="para" id="para" class="mdl-textfield__input">
```

Input assunto:

```
#./src/app/app.component.html
```

```
<input  
    #assunto="ngModel"  
    [(ngModel)]="email.assunto"  
    required type="assunto" name="assunto" id="assunto" class="mdl-textfield__input">
```

Textarea conteúdo:

```
#./src/app/app.component.html
```

```
<textarea  
    #conteudo="ngModel"  
    [(ngModel)]="email.conteudo" name="conteudo"  
    class="mdl-textfield__input" id="conteudo"></textarea>
```

Por último, podemos aproveitar que temos as referências dos campos que necessitam validação armazenadas em variáveis de template e pegar o status individual de cada uma, usando como condição para mostrar as mensagens de validação na tela.

6. Adicione a diretiva **[ngClass]** na *<div>* antes de cada *<input>*. Podemos usar as propriedades **invalid** e **touched** combinadas como condição para inserir a classe **is-invalid** :

```
#./src/app/app.component.html
```

```
<!-- [Para] -->  
<div class="cmailInputForm">  
  <div class="mdl-textfield mdl-textfield--floating-label"  
    [ngClass]="{'is-invalid': destinatario.invalid && destinatario.touched}"  
  >  
  <!-- código posterior omitido -->
```

```
#./src/app/app.component.html
```

```
<!-- [Assunto] -->
<div class="cmailInputForm">
  <div class="mdl-textfield mdl-textfield--floating-label"
    [ngClass]="'is-invalid': assunto.invalid && assunto.touched}"
  >
<!-- código posterior omitido -->
```

Agora já temos o formulário preparado e a lógica feita para **evitar** que ele seja submetido sem ter seus dados preenchidos.

ARQUITETURA DE PROJETO E CLI

A medida que a aplicação vai crescendo, temos que tomar decisão em relação a organização da estrutura de pastas e arquivos. A estrutura gerada inicialmente pela CLI nos dá um padrão inicial onde todos os projetos Angular devem seguir.

Todo o código fonte de nossa autoria deve estar dentro do diretório `src`, e *features* devem conter seus próprios diretórios e respectivos `NgModule`'s. Cada "coisa" que é desenvolvida, como componentes, *pipes*, *services*, deve ter seu próprio arquivo e estes arquivos devem seguir a convenção de nomes proposta pelo *framework* na documentação[2].

Todo código incluído que foi feito por terceiros (*third party vendor*) deve estar incluído em outro diretório, fora de `src`.

13.1 PROPOSTAS PARA ARQUITETURA

A documentação do Angular propõe algumas diretrizes que guiam como devem ser a organização dos diretórios e arquivos da aplicação. São elas:

LIFT - Locale, Identify, Flat, T-DRY

Localizar - Locate

A localização do código deve ser intuitivo, simples e rápido.

Por quê? Para trabalhar com eficiência, você deve conseguir localizar arquivos rapidamente, especialmente quando não souber (ou não lembrar) os nomes dos arquivos. Manter os arquivos relacionados próximos uns dos outros em um local intuitivo economiza tempo. Uma estrutura de pastas descriptiva faz uma super de diferença para você e para as outras pessoas que irão trabalhar no projeto.

Identificar - Identify

Nomeie o arquivo de forma que você saiba instantaneamente o que ele contém e representa.

Seja descriptivo com nomes de arquivos e mantenha o conteúdo do arquivo com apenas um componente.

Evite arquivos com vários componentes, vários serviços e coisas misturadas.

Por quê? Para gastar menos tempo caçando o código e tornar-se mais eficiente. Nomes de arquivos longos são muito melhores do que nomes abreviados curtos portanto obscuros.

Plano - Flat

Mantenha uma estrutura de pastas simples o maior tempo possível.

Considere criar subpastas quando uma pasta atingir **sete ou mais** arquivos.

Considere configurar seu editor de texto ou IDE, para ocultar arquivos irrelevantes e que causam distração, como arquivos .js e .js.map gerados.

Por quê? Ninguém quer procurar um arquivo através de sete níveis de pastas. Uma estrutura plana é fácil de explorar.

Por outro lado, os psicólogos dizem que os humanos tem dificuldades quando o número de coisas interessantes excede nove. Portanto, quando uma pasta tiver dez ou mais arquivos, talvez seja hora de criar subpastas.

Baseie sua decisão no seu nível de conforto. Use uma estrutura mais plana até que haja um valor óbvio para criar uma nova pasta.

T-DRY - Tente Não Se Repetir

Seja DRY (*Don't Repeat Yourself*).

Porém, evite ser tão "não-repetitivo" ao ponto de sacrificar a legibilidade.

Por quê? Ser DRY é importante, mas não crucial, se isto for sacrificar os outros elementos do LIFT. E é por isso que é chamado T-DRY (tente não se repetir). Por exemplo, é redundante nomear um arquivo de *template* de *cadastro-view.component.html* pois este já tem a extensão .html, que nos diz obviamente que é uma **view** exibição. Porém se algo não é óbvio ou se afasta de uma convenção, então descreva.

Proposta de estrutura

Abaixo um exemplo de como é uma estrutura de pastas e arquivos seguindo as diretrizes:

```
| -- <project root>
| -- app
| -- modules
| -- home
|   -- [+] components
|   -- [+] pages
|   -- home-routing.module.ts
|   -- home.module.ts
| -- core
|   -- [+] authentication
|   -- [+] footer
```

```

|-- [+] guards
|-- [+] http
|-- [+] interceptors
|-- [+] mocks
|-- [+] services
|-- [+] header
|-- core.module.ts
|-- ensureModuleLoadedOnceGuard.ts
|-- logger.service.ts
|
|-- shared
|   |-- [+] components
|   |-- [+] directives
|   |-- [+] pipes
|   |-- [+] models
|
|-- [+] configs
|-- assets
|   |-- scss
|     |-- [+] partials
|     |-- _base.scss
|     |-- styles.scss
|-- node_modules/...
|-- ...

```

Obs.: O `[+]` indica que naquele diretório pode haver arquivos adicionais.

Crie um diretório, seguindo a diretriz LIFT, para:

- Uma nova funcionalidade (*feature*) e seu respectivo `NgModule` ;
- Componentes compartilhados (*shared*), que estarão presentes em várias páginas e funcionalidades, e crie apenas um único `NgModule` que exporta todos estes componentes;
- Páginas ou funcionalidades que não tem a necessidade de entrarem no primeiro carregamento da aplicação (carregamento tardio/preguiçoso *lazy-load*)

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

13.2 CLI PARA TAREFAS REPETITIVAS

A Command Line Interface - (CLI) nos ajuda em uma série de tarefas repetitivas, como criar um projeto do zero, iniciar o servidor de desenvolvimento, gerar o *build* para produção, executar os testes, e claro que não seria diferente para criar componentes e módulos, pois os que difere na estrutura inicial é apenas o nome, todos começam iguais.

Configurando o `schematics` no `angular.json`

Os comandos `ng generate` e `ng add` tomam como argumento o artefato ou biblioteca a ser gerada ou adicionada ao projeto atual. Além de quaisquer opções gerais, cada artefato ou biblioteca define suas próprias opções em um "esquema" (*schematic*). As opções de esquemas (*schematics*) são fornecidas para o comando da mesma forma que as opções para estes comandos.

No arquivo `angular.json`, gerado pelo `ng new`, localizado na pasta raiz do *workspace*, é possível configurar opções de como queremos que a CLI gere estes artefatos, como por exemplo: pular importação no módulo, não criar uma pasta, não gerar um arquivo de CSS etc.

Podemos adicionar estas opções de maneira que afete todo o workspace, ou por projeto.

Para adicionar para toda a workspace, basta usar a propriedade *schematics* no primeiro nível:

```
{  
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",  
  "version": 1,  
  "newProjectRoot": "projects",  
  "projects": {},  
  "defaultProject": "teste",  
  "cli": {},  
  "schematics": {}  
}
```

As opções devem ser passadas no formato `"schematic-package:schematic-name (object):"` {objeto contendo as informações para este esquema}, as opções são:

```
"@schematics/angular:component"  
"@schematics/angular:directive"  
"@schematics/angular:module"  
"@schematics/angular:service"  
"@schematics/angular:pipe"  
"@schematics/angular:class"
```

O autocomplete do Visual Studio Code, nos ajuda fazer este trabalho, tanto para as opções (propriedades) quanto para os valores delas.

Para configurar as opções do *schematics* por projeto, basta procurar a propriedade de mesmo nome dentro do objeto `projects`, geralmente está na linha 11 do arquivo, porém isto pode sofrer modificações de acordo com a versão da CLI.

Criando módulos

Primeiramente vamos criar módulos, pois iremos exemplificar como gerar componentes que terão o papel de páginas, e páginas devem ser "*lazy-loaded*", portanto devem ter seus próprios `NgModule`'s.

No terminal, abra o caminho onde você quer que o módulo e sua pasta sejam gerados, caso contrário o mesmo será criado na raiz da pasta `app`.

No exemplo abaixo vamos criar a pasta `modules` e também será gerada uma pasta como nome do modulo, no caso `cadastro`:

Digite o comando `ng generate module <nomedir>/<nomedomodulo>`, por exemplo:

```
~/cmail ng generate module modules/cadastro
```

Existe um atalho para este comando, que o torna bem mais fácil de executá-lo:

```
~/cmail ng g m modules/cadastro
```

Criando componentes

Para criar componentes automaticamente usando a CLI, primeiro vamos pensar se existe alguma opção do `schematic` que vamos querer alterar no projeto. Uma sugestão aqui seria:

```
"schematics": {  
  "@schematics/angular:component": {  
    "inlineStyle": true,  
    "prefix": "cmail",  
    "spec": false,  
    "flat": true,  
    "export": true  
  }  
}
```

Com `inlineStyle` a CLI não vai gerar um arquivo de CSS, será adicionado um prefixo no selector do componente `cmail`, não será gerado um arquivo de testes `.spec`, e o `flat` não cria uma pasta com o mesmo nome do componente, pois aqui já foi criada a pasta `cadastro` no momento em que geramos o módulo, e `export` fará com que o componente seja exportado pelo módulo acima dele (permitindo que o componente possa ser utilizado por outros módulos).

Com isto, agora basta abrir no terminal a pasta onde você quer que o componente seja gerado, caso contrário o mesmo será criado na raiz da pasta `app`, e o componente será automaticamente declarado em `AppModule` - o que não é o que queremos agora.

No exemplo abaixo vamos criar um componente que será uma página de cadastro, portanto estará na pasta `modules/cadastro`.

Nesta pasta digite o comando `ng generate component <nomedocomponente>`, por exemplo:

```
~/cmail/src/app/modules/cadastro ng generate component cadastro
```

Existe um atalho para este comando, que o torna bem mais fácil de executá-lo:

```
~/cmail/src/app/modules/cadastro ng g c cadastro
```

Caso não queira entrar na pasta do componente, uma maneira de fazer é:

```
~/cmail ng g c modules/cadastro/cadastro
```

O resultado será:

```
CREATE src/app/modules/cadastro/cadastro.component.html (27 bytes)
CREATE src/app/modules/cadastro/cadastro.component.ts (250 bytes)
UPDATE src/app/modules/cadastro/cadastro.module.ts (267 bytes)
```

Perceba que por padrão a CLI já declara o componente no próximo módulo que ela encontrar, por isto criamos o módulo primeiro, caso contrário ela importaria o componente em `AppModule`, e teríamos que remove-lo de lá e depois declará-lo no próprio módulo quando este for criado. Caso não seja desejado esse comportamento de importar o componente no módulo mais próximo dele, basta usar a opção do `schematic skipImport` para componentes.

Referências

1. <https://angular.io/guide/file-structure>
2. <https://angular.io/guide/styleguide>
3. <https://medium.com/@motcowley/angular-folder-structure-d1809be95542>
4. <https://itnext.io/choosing-a-highly-scalable-folder-structure-in-angular-d987de65ec7>
5. <https://angular.io/cli/generate>
6. https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two
7. <https://github.com/angular/angular-cli/wiki/angular-workspace>

ROTEAMENTO SIMPLES

Até o momento a aplicação tem apenas uma página, e não temos navegação entre outras páginas. Porém, como é possível uma Single Page Application conter mais de uma página?

Isto só é possível graças ao **Angular Router**. Este sistema modifica a navegação das URLs da aplicação, fazendo com que o navegador não faça novas requisições para o mesmo documento HTML, e sim exiba os componentes no documento já carregado através da manipulação do DOM.

O mecanismo de roteamento é acionado de acordo com as seguintes interações do usuário na aplicação:

- Ao digitar uma URL na barra de endereços, e o navegador direciona até a página correspondente;
- Ao entrar em links nas páginas e o navegador direciona para esta nova página;
- Ao usar os botões de voltar ou avançar do navegador, e o navegador direciona para o histórico de páginas visitas.

14.1 REGISTRANDO UM MÓDULO DE ROTEAMENTO

Para configurarmos o sistema de roteamento, iremos precisar do `RouterModule` disponível pelo pacote `@angular/router`, e o mesmo deve ser importando em `AppModule`.

Porém, em `AppModule`, devemos importar uma instância de `RouterModule` já configurada com as rotas suportadas pela aplicação, e não apenas a referência do módulo.

Para isto podemos invocar o método `forRoot` de `RouterModule`, que constrói esta instância do roteamento, recebendo a lista de rotas, e é o retorno deste método que importamos em `AppModule`. Podemos fazer isto desta forma:

```
#src/app/app.module.ts
```

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot([
      { path: '/', component: InboxComponent }
    ])
  ],
})
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

14.2 CONFIGURAÇÃO PARA ROTAS

As rotas devem ser passadas como parâmetro para o método `forRoot`, e caso comece a se tornar uma lista muito grande é possível armazená-las em uma referência constante em um arquivo separado `app.routes.ts` para melhor organização. Por exemplo:

```
#src/app/app.routes.ts

import { Routes } from '@angular/router';

export const appRoutes: Routes = []
```

Lembrando que `appRoutes` deve ser uma lista, e colocamos `export` antes para que esta possa ser importada em `AppModule`. Porém aqui a lista ainda está vazia, sem rotas configuradas.

Como definimos o tipo da lista de rotas com `Routes`, ao usarmos o recurso do *autocomplete* do Visual Studio Code, ele já nos ensina como criar uma rota, que é um objeto literal, com propriedades e valores.

Quando a aplicação é iniciada, a URL inicial na barra do navegador é algo como:

localhost:4200

Para fazer uma rota bem simples para a página principal da aplicação, devemos usar a propriedade `path` que indica o caminho da rota e seu valor será uma *string*, e em seguida a propriedade `component` que indica qual componente será carregado, e seu valor deverá ser a referência para a classe do componente:

```
#src/app/app.routes.ts

import { Routes } from '@angular/router';
import { LoginComponent } from './modules/login/login.component';
```

```
export const appRoutes: Routes = [
  {path: '', component: LoginComponent}
]
```

O `path` para a página principal é uma *string* vazia. Para as demais páginas, basta escrever o nome do seguimento de URL **sem iniciar com barra**.

```
#src/app/app.routes.ts
```

```
export const appRoutes: Routes = [
  {path: 'inbox', component: InboxComponent},
  ,{path: 'cadastro', component: CadastroComponent},
  ,{path: '', component: LoginComponent}
]
```

É possível criar rotas que recebem parâmetros na URL, estes parâmetros são passados como seguimentos (trechos) de URL, para isto devemos registrar a rota que recebe parâmetro, e com `/:` indicamos que aquele será uma variável e neste caso damos um nome a esta variável como `id`:

```
#src/app/app.routes.ts
```

```
export const appRoutes: Routes = [
  {path: 'inbox', component: InboxComponent},
  ,{path: 'cadastro', component: CadastroComponent},
  ,{path: 'editar/:id', component: CadastroComponent},
  ,{path: '', component: LoginComponent}
]
```

Por fim, importamos as rotas em `AppModule`, que deverá ficar similar ao exemplo abaixo:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';

import { AppComponent } from './app.component';
import { CadastroModule } from './modules/cadastro/cadastro.module';
import { appRoutes } from './app.routes';
import { LoginModule } from './modules/login/login.module';
import { InboxModule } from './modules/inbox/inbox.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    CadastroModule,
    LoginModule,
    InboxModule,
    RouterModule.forRoot(appRoutes)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

14.3 SAÍDA DO ROTEAMENTO

Importar o `RouterModule` e configurá-lo não é o suficiente para que as rotas começarem a funcionar. Percebemos isto ao digitar na barra de endereços do navegador alguma das rotas que definimos, e nada acontecer, na verdade, apenas uma tela em branca é exibida. Isto acontece pois é necessário indicar onde será a saída do roteamento, ou seja, em qual template ele entrará em ação.

Por padrão, indicamos que o roteamento inicia no template de `AppComponent`, então, deixamos este componente apenas como uma espécie de moldura (*frame*) para ali dentro exibir os componentes de acordo com a URL da barra de endereços. Para fazer este "*frame*" usamos o componente `<router-outlet></router-outlet>`, que é disponibilizado por `RouterModule`.

Com isto `AppComponent`, poderá ficar o mais simples possível, apenas com a saída do roteamento:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<router-outlet></router-outlet>`
})
export class AppComponent {}
```

Pronto! Agora já temos URLs e o roteamento funcionando, e podemos verificar isto digitando na barra de endereços os *paths* definidos na lista de rotas.

14.4 LINKS EM UMA SPA

Para criarmos links entre páginas de uma SPA, não podemos usar o atributo `href` da tag `<a>`, pois este atributo sempre faz o navegador fazer uma nova requisição para o documento HTML do endereço informado. Usamos apenas para links para sites externos, porém para páginas da própria aplicação, usamos uma diretiva específica para isto, e disponibilizada pelo `RouterModule`, a `routerLink`:

```
<h1>Login</h1>
<form>
  ...
</form>
<a routerLink="/cadastro">Cadastre-se!</a>
```

Neste caso, se for passar apenas um (1) segmento de url, podemos passar direto uma string como valor da diretiva, porém, esta deverá ter a barra no inicio.

Caso passarmos um valor que é uma variável, podemos envolver a diretiva com colchetes, e como valor devemos informar uma lista:

```
<h1>Login</h1>
<form>
  ...
</form>
<a [routerLink]=["cadastro", register]>Cadastre-se!</a>
```

Um detalhe é que para que esta diretiva esteja disponível para `LoginModule`, devemos importar

RouterModule nele:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { LoginComponent } from './login.component';
import { RouterModule } from '@angular/router';

@NgModule({
  declarations: [LoginComponent],
  imports: [
    CommonModule,
    RouterModule
  ],
  exports: [LoginComponent]
})
export class LoginModule { }
```

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

14.5 ROTAS CURINGAS

Até momento sabemos como definir uma lista de determinadas rotas. Caso seja digitado na barra de endereços uma rota que não foi definida na lista de rotas informada ao `RouterModule`, uma tela em branco será exibida e um erro no console:

```
core.js:15724
○ > ERROR: Error: Uncaught (in promise): Error: Cannot match any routes. URL Segment: 'user'
Error: Cannot match any routes. URL Segment: 'user'
at ApplyRedirects.push../node_modules/@angular/router/fesm5/router.js.ApplyRedirects.noMatchError (router.js:2469)
at CatchSubscriber.selector [router.js:2450]
at CatchSubscriber.push../node_modules/rxjs/_esm5/internal/operators/catchError.js.CatchSubscriber.error (catchError.js:34)
at MapSubscriber.push../node_modules/rxjs/_esm5/internal/Subscriber.js.Subscriber.error (Subscriber.js:80)
at MapSubscriber.push../node_modules/rxjs/_esm5/internal/Subscriber.js.Subscriber.error (Subscriber.js:60)
at MapSubscriber.push../node_modules/rxjs/_esm5/internal/Subscriber.js.Subscriber.error (Subscriber.js:80)
at MapSubscriber.push../node_modules/rxjs/_esm5/internal/Subscriber.js.Subscriber.error (Subscriber.js:60)
at MapSubscriber.push../node_modules/rxjs/_esm5/internal/Subscriber.js.Subscriber.error (Subscriber.js:80)
at MapSubscriber.push../node_modules/rxjs/_esm5/internal/Subscriber.js.Subscriber.error (Subscriber.js:60)
at TapSubscriber.push../node_modules/rxjs/_esm5/internal/operators/tap.js.TapSubscriber._error (tap.js:61)
at resolvePromise (zone.js:831)
at resolvePromise (zone.js:788)
at zone.js:899
at ZoneDelegate.push../node_modules/zone.js/dist/zone.js.ZoneDelegate.invokeTask (zone.js:423)
at Object.onInvokeTask (core.js:17290)
at ZoneDelegate.push../node_modules/zone.js/dist/zone.js.ZoneDelegate.invokeTask (zone.js:422)
at Zone.push../node_modules/zone.js/dist/zone.js.Zone.runTask (zone.js:195)
at drainMicroTaskQueue (zone.js:601)
```

Figura 14.1: Path "user" não existe

Devemos então adicionar uma rota curinga (*wildcard route*) para interceptar URLs inválidas e devidamente tratá-las. Esta rota é universal, e seu caminho consiste em dois asteriscos, ou seja, o path

com valor *double-star* `'**'`, que significa "ou qualquer outra coisa" [4], e deve ser o último na lista de rotas, pois a partir do carregamento desta rota, todas as outras são ignoradas.

```
#src/app/app.routes.ts
```

```
const appRoutes: Routes = [
  {path: 'inbox', component: InboxComponent},
  ,{path: 'cadastro', component: CadastroComponent},
  ,{path: 'editar/:id', component: CadastroComponent}
  ,{path: '', component: LoginComponent}
  ,{path: '**', component: PaginaNaoEncontradaComponent }
]
```

Uma rota curinga pode navegar para um componente "404 página não encontrada" personalizada ou redirecionar para alguma rota existente.

Redirecionamento

Sabemos que a rota da página inicial, é representada por `''`. Neste ponto caso ela ainda não tivesse sido definida, ao acessarmos ela, ela cairia na condição `'**'` e o `PaginaNaoEncontradaComponent` seria exibido. Portanto é suma importância definir um componente para a rota principal (também chamada de rota padrão).

No exemplo que estamos usando, o componente carregado na rota padrão é o `LoginComponent`, porém é interessante que termos uma rota `login` também para links mais semânticos. Então o ideal é que tenhamos a rota `login`, e a rota padrão redirecione para o `LoginComponent`.

Para isto usamos a propriedade `redirectTo` das rotas, que recebe como valor uma *string* com o *path* de uma rota:

```
#src/app/app.routes.ts
```

```
const appRoutes: Routes = [
  {path: 'inbox', component: InboxComponent},
  ,{path: 'cadastro', component: CadastroComponent},
  ,{path: 'editar/:id', component: CadastroComponent}
  ,{path: 'login', component: LoginComponent}
  ,{path: '', redirectTo: 'login'}
  ,{path: '**', component: PaginaNaoEncontradaComponent}
]
```

A rota padrão, deve ser adicionada em algum lugar acima da rota curinga, e por convenção ela pode ser usada logo acima da rota curinga para que fique fácil de localizá-la.

Referências

1. <https://angular.io/start/routing>
2. <https://angular.io/guide/router>
3. <https://angular.io/api/router/Route>
4. <https://angular.io/api/router/Route#path>
5. <https://angular.io/guide/router#set-up-redirects>

EXERCÍCIO: ROTEAMENTO SIMPLES

15.1 OBJETIVO

Agora que já temos a **caixa de entrada** do CMail pronta, precisamos começar a pensar nas outras páginas da aplicação, como a de **login** e a **cadastro** para novas contas de email, para que depois seja possível enviar emails de uma conta para a outra. Para isto vamos ter que separar o código da caixa de entrada do **AppComponent** em um novo componente, que será este o **CaixaDeEntradaComponent**. Além disso vamos fazer a estrutura inicial de **LoginComponent** e **CadastroComponent**.

Com os três componentes criados, poderemos ter rotas específicas para acessar cada componente. Criaremos então um roteamento simples para acessarmos cada um destes componentes em diferentes caminhos da aplicação:

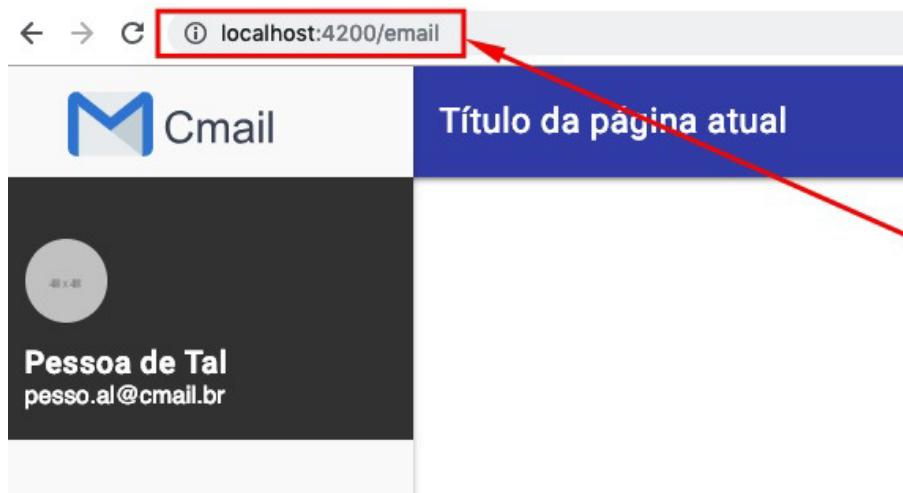


Figura 15.1: Componente Caixa de Entrada na rota /email

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

15.2 PASSO A PASSO COM CÓDIGO

Para que não seja necessário digitar o código dos arquivos de HTML ou CSS, baixe-os a partir do repositório dos arquivos do curso, pasta **06**:

1. Vamos criar o **CaixaDeEntradaComponent** e passar o código que fizemos em **AppComponent**, deixando este último apenas com a estrutura básica inicial:

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.ts
```

```
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'cmail-caixa-de-entrada',
  templateUrl: './caixa-de-entrada.component.html',
  styles: []
})
export class CaixaDeEntradaComponent {

  private _isNewEmailFormOpen = false;

  emailList = [];
  email = {
    destinatario: '',
    assunto: '',
    conteudo: ''
  }

  get isNewEmailFormOpen() {
    return this._isNewEmailFormOpen;
  }
}
```

```

toggleNewEmailForm() {
  this._isNewEmailFormOpen = !this.isNewEmailFormOpen
}

handleNewEmail(formEmail: NgForm) {
  if (formEmail.invalid) return;

  this.emailList.push(this.email)

  this.email = {
    destinatario: '',
    assunto: '',
    conteudo: ''
  }

  formEmail.reset();
}

}

```

#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.html

```

<div class="mdl-layout mdl-js-layout mdl-layout--fixed-drawer mdl-layout--fixed-header">
  <cmail-header></cmail-header>
  <main class="mdl-layout__content">
    <div class="page-content">

      <div class="mdl-grid">
        <ul>
          <li *ngFor="let email of emailList">
            <strong>Para:</strong>{{email.destinatario}} <br>
            <strong>Assunto:</strong>{{email.assunto}}
          </li>
        </ul>
      </div>

      <div class="mdl-grid">
        <!-- Floating Multiline Textfield -->
        <button class="globalFab tooltip btn mdl-button mdl-js-button mdl-button--fab mdl-button--colored" role="tooltip"
          aria-label="Criar novo email" (click)="toggleNewEmailForm()" #rmEmail="ngForm">
          <span class="material-icons">add</span>
        </button>

        <!-- newEmail--active -->
        <form class="newEmail cmailForm" [ngClass]="{ 'newEmail--active': isNewEmailFormOpen }" #fo
          (ngSubmit)="handleNewEmail(formEmail)">

          <div class="newEmail__card demo-card-wide mdl-card mdl-shadow--2dp">
            <div class="newEmail__titleArea mdl-card__title">
              <h2 class="newEmail__title mdl-card__title-text">Novo Email</h2>
            </div>
            <div class="newEmail__body mdl-card__supporting-text">
              <!-- Form Fields -->
              <!-- [Para] -->
              <div class="cmailInputForm">
                <div class="mdl-textfield mdl-textfield--floating-label" [ngClass]="{'is-invalid': destinatario.invalid && destinatario.touched}">
                  <input #destinatario="ngModel" [(ngModel)]="email.destinatario" required type="em

```

```

ail" name="para" id="para"
    class="mdl-textfield__input">
        <span class="mdl-textfield__error">Informar um email é obrigatório!</span>
        <label class="mdl-textfield__label" for="para">Para: </label>
        <span class="mdl-textfield__formline"></span>
    </div>
</div><!-- ./[Para] -->

<!-- [Assunto] -->
<div class="cmailInputForm">
    <div class="mdl-textfield mdl-textfield--floating-label" [ngClass]="{'is-invalid': assunto.invalid && assunto.touched}">
        <input #assunto="ngModel" [(ngModel)]="email.assunto" required type="assunto" name="assunto" id="assunto"
            class="mdl-textfield__input">
            <span class="mdl-textfield__error">Informar um assunto é obrigatório!</span>
            <label class="mdl-textfield__label" for="email"> Assunto: </label>
            <span class="mdl-textfield__formline"></span>
    </div>
</div><!-- ./[Assunto] -->

<!-- [Conteúdo] -->
<div class="cmailInputForm">
    <div class="mdl-textfield mdl-textfield--floating-label">
        <textarea #conteudo="ngModel" [(ngModel)]="email.conteudo" name="conteudo" class="mdl-textfield__input"
            id="conteudo"></textarea>
        <span class="mdl-textfield__error">Informar um conteúdo é obrigatório!</span>
        <label class="mdl-textfield__label" for="conteudo">Conteúdo: </label>
        <span class="mdl-textfield__formline"></span>
    </div>
</div><!-- ./[Conteúdo] -->

<!-- ./Form Fields -->
</div>
<div class="mdl-card__actions mdl-card--border">
    <button class="mdl-button">
        Enviar
    </button>
</div>
<div class="newEmail__topMenu mdl-card__menu">
    <button type="button" class="mdl-button mdl-button--icon" (click)="toggleNewEmailForm()">
        <span class="material-icons">close</span>
    </button>
</div>
</form>
</div>
</div>
</div>
</main>
</div>

```

AppComponent ficará assim:

#./src/app/app.component.ts

```

import { Component } from '@angular/core';

@Component({

```

```

    selector: 'app-root',
    template: '',
    styles: []
})
export class AppComponent {}

```

Observação: Perceba que agora como não temos nenhum HTML e nem CSS para **AppComponent**, podemos apagar estes arquivos e usar a opção de **inline template** e **styles** no *decorator @Component* .

- Vamos adicionar **CaixaDeEntradaComponent** em **AppModule** para que possa funcionar:

#./src/app/app.module.ts

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { HeaderComponent } from './components/header/header.component';

import { FormsModule } from "@angular/forms";
import { CaixaDeEntradaComponent } from './modules/caixa-de-entrada/caixa-de-entrada.component';

@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent,
    CaixaDeEntradaComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}

```

- Vamos gerar a estrutura base de **LoginComponent** e **CadastroComponent**. Porém, fazer isto manualmente é um pouco trabalhoso não? Para isto então vamos usar a CLI do Angular que é capaz de realizar este trabalho para nós!

No terminal do VSCode, digite o comando da CLI para gerar novos componentes:

```
ng generate component modules/login
```

Este comando retornará o seguinte:

```

CREATE src/app/modules/login/login.component.css (0 bytes)
CREATE src/app/modules/login/login.component.html (24 bytes)
CREATE src/app/modules/login/login.component.spec.ts (621 bytes)
CREATE src/app/modules/login/login.component.ts (265 bytes)
UPDATE src/app/app.module.ts (683 bytes)

```

Perceba que além de gerar a estrutura base de arquivos (TS, HTML, CSS e Spec) dentro da pasta **modules** , o novo componente foi automaticamente declarado em **AppModule**.

4. Para criar o próximo componente usando a extensão do VSCode chamada Angular Schematics. Ela faz uma interface gráfica para a CLI, facilitando seu uso. Então vamos instalar ela:

The screenshot shows the Angular Schematics extension page on the Visual Studio Code Marketplace. At the top, there's a large red icon with a white 'A' inside. To its right, the text 'Angular Schematics' is displayed, followed by the GitHub handle 'cyrilletuzi.angular-schematics'. Below this, it says 'Cyrille Tuzi | 233,028 | ★★★★★ | Repository | License'. A note below states 'Angular schematics (CLI commands) from files Explorer or Command Palette.' There are two buttons at the bottom: 'Disable' and 'Uninstall'. Below the main title, there are three links: 'Details', 'Contributions', and 'Changelog'. The main content area has a heading 'Angular schematics extension for Visual Studio Code'. It includes a note about the extension being a 'Visual Studio Code extension allowing you to launch Angular schematics (CLI commands) with a Graphical User Interface, directly inside VS Code!'. A section titled 'Why?' lists 'Productivity!' and a bulleted list: 'Save time', 'No more typo errors = no more mess cleaning', 'No more chaotic search in the CLI wiki, all options available will be proposed and documented', and 'Promote good practices for component types'. Another section discusses other tools like the Angular Console and highlights the benefits of this extension. A sidebar on the right contains a link to '@ngx-pwa/local-storage: 1st Angular library for local storage' and 'Library @ngx-pwa/offline', along with a 'Popular Angular posts on Medium' link.

Figura 15.2: Extensão do VSCode Angular Schematics

5. Para gerar o **CadastroComponent** com a extensão, clicamos com o botão direito em cima da pasta `modules` que criamos, e escolhemos a opção **Angular: Generate a Component**:

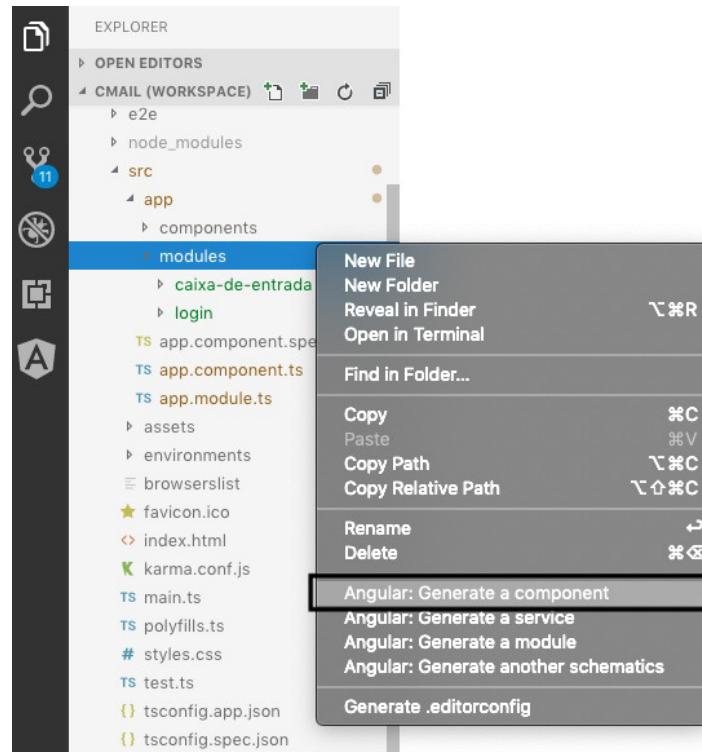


Figura 15.3: Gerando componente

Depois irá abrirá uma entrada para confirmar a pasta e o nome do componente:

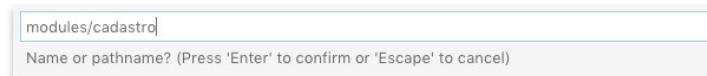


Figura 15.4: Path e nome

Vamos escolher o tipo do componente: Classic Component

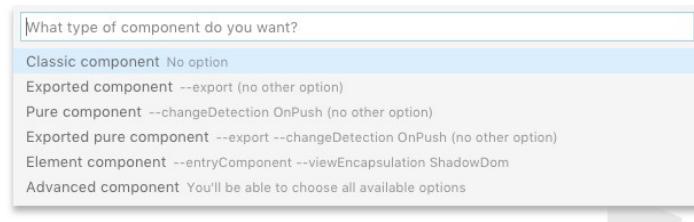


Figura 15.5: Tipo do componente

Confirmar:

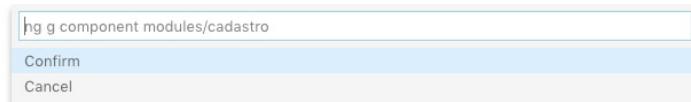


Figura 15.6: Confirmar

Resultado:

```

1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-cadastro',
5   templateUrl: './cadastro.component.html',
6   styleUrls: ['./cadastro.component.css']
7 })
8 export class CadastroComponent implements OnInit {
9
10   constructor() { }
11
12   ngOnInit() {
13   }
14
15 }
16

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Angular schematics

ng g component modules/cadastro

CREATE src/app/modules/cadastro/cadastro.component.css (0 bytes)

CREATE src/app/modules/cadastro/cadastro.component.html (27 bytes)

CREATE src/app/modules/cadastro/cadastro.component.spec.ts (642 bytes)

CREATE src/app/modules/cadastro/cadastro.component.ts (277 bytes)

UPDATE src/app/app.module.ts (781 bytes)

Figura 15.7: Saída/Resultado

Agora que já temos cada página do CMail, separado em componentes, quando acessamos `localhost:4200` vemos uma página em branco, e está correto, pois não há nada no template de `AppComponent`. Este é o momento em que temos que por em ação o roteamento do Angular, com o **RouterModule**.

- Criaremos agora um arquivo na raiz da pasta `app` para armazenar a nossa lista de rotas, o `app.routes.ts`. A lista de rotas, traduzindo em código, é um *array* do tipo `Routes`, e referenciamos este *array* em uma constante, `rotas`:

```
#./src/app/app.routes.ts
```

```
import { Routes } from "@angular/router";
const routes: Routes = []
```

- Para cada rota, criamos um objeto com as propriedades `path` que recebe como valor a *string* da rota, e `component` que recebe a classe do component que será carregado na respectiva rota.

```
#./src/app/app.routes.ts
```

```
import { Routes } from "@angular/router";
import { LoginComponent } from './modules/login/login.component';
import { CaixaDeEntradaComponent } from './modules/caixa-de-entrada/caixa-de-entrada.component';
import { CadastroComponent } from './modules/cadastro/cadastro.component';

const rotas: Routes = [
  {path: '', component: LoginComponent},
  {path: 'inbox', component: CaixaDeEntradaComponent },
  {path: 'cadastro', component: CadastroComponent}
]
```

8. Agora trazemos a classe **RouterModule** e executamos o método dela `forRoot` que recebe como parâmetro um array de rotas, ou seja, a constante `rotas` que criamos. O método `forRoot` retorna um tipo `ModuleWithProviders`, que é o módulo de roteamento configurado com as rotas que definimos. Este retorno pode ser guardado em uma referência constante e exportada para que seja importada no módulo principal da aplicação.

```
#./src/app/app.routes.ts
```

```
import { Routes, RouterModule } from "@angular/router";
import { LoginComponent } from './modules/login/login.component';
import { CaixaDeEntradaComponent } from './modules/caixa-de-entrada/caixa-de-entrada.component';
import { CadastroComponent } from './modules/cadastro/cadastro.component';

const rotas: Routes = [
  {path: '', component: LoginComponent},
  {path: 'inbox', component: CaixaDeEntradaComponent },
  {path: 'cadastro', component: CadastroComponent}
]

export const ModuloRoteamento = RouterModule.forRoot(rotas);
```

9. Agora que já temos nossas rotas configuradas, devemos importá-las em **AppModule**:

```
#./src/app/app.module.ts
```

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { HeaderComponent } from './components/header/header.component';

import { FormsModule } from "@angular/forms";
import { CaixaDeEntradaComponent } from './modules/caixa-de-entrada/caixa-de-entrada.component';
import { LoginComponent } from './modules/login/login.component';
import { CadastroComponent } from './modules/cadastro/cadastro.component';
import { ModuloRoteamento } from './app.routes'; //Importação da referência

@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent,
    CaixaDeEntradaComponent,
    LoginComponent,
    CadastroComponent
  ]
```

```
],
imports: [
  BrowserModule,
  FormsModule,
  ModuloRoteamento //Módulo de Roteamento aqui!
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

O CMail segue com a tela em branco. Nos falta um último detalhe, usar o componente que indica aonde no template da aplicação roteamento vai atuar.

10. No template de **AppComponent**, inserimos a tag `<router-outlet></router-outlet>` , que é fornecida pelo **RouterModule**:

`#./src/app/app.component.ts`

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: '<router-outlet></router-outlet>',
  styles: []
})
export class AppComponent {}
```

Pronto! Acessadas manualmente pela barra de endereços do navegador conseguimos testar cada rota funcionando!

Para saber mais: `ng generate`

Existe também os atalhos para os comandos `generate` e `component`. Este mesmo comando pode ser executando com atalhos, ficando desta forma: `ng g c login`.

EXERCÍCIO: LINKS PARA AS ROTAS E REDIRECIONAMENTO

16.1 OBJETIVO

No menu lateral temos os links para as páginas de *login*, cadastro, caixa de entrada e *logout*. Vamos aprender como fazê-los utilizando os recursos do **RouterModule**, e também fazer um tratamento caso uma rota não declarada for acessada.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

16.2 PASSO A PASSO COM CÓDIGO

1. Usamos a diretiva **routerLink** que irá receber como valor o **path** da rota:

```
#./src/app/components/header/header.component.html
```

```
<!-- código anterior omitido -->
<nav class="mdl-navigation">
  <a routerLink="" class="mdl-navigation__link">
    <span class="material-icons">person</span> Login
  </a>
  <a routerLink="/cadastro"
    routerLinkActive="mdl-navigation__link--current"
    class="mdl-navigation__link">
    <span class="material-icons">create</span> Cadastro
  </a>
```

```

<a routerLink="/inbox"
  routerLinkActive="mdl-navigation__link--current"
  class="mdl-navigation__link">
  <span class="material-icons">inbox</span> Caixa de Entrada
</a>
<a routerLink="/logout" class="mdl-navigation__link">
  <span class="material-icons">logout</span> Logout
</a>
</nav>
<!-- código posterior omitido -->

```

2. Vamos adicionar mais uma rota, a `**` que representa a última condição, ou seja, caso não seja nenhuma das rotas acima, e podemos usar o `** redirectTo` para redirecionar para uma outra rota já declarada:

`#./src/app/app.routes.ts`

```

import { Routes, RouterModule } from "@angular/router";
import { LoginComponent } from './modules/login/login.component';
import { CaixaDeEntradaComponent } from './modules/caixa-de-entrada/caixa-de-entrada.component';
import { CadastroComponent } from './modules/cadastro/cadastro.component';

const rotas: Routes = [
  { path: 'login', component: LoginComponent },
  { path: 'inbox', component: CaixaDeEntradaComponent },
  { path: 'cadastro', component: CadastroComponent },
  {
    path: '',
    pathMatch: 'full',
    redirectTo: 'inbox'
  },
  {
    path: '**',
    pathMatch: 'full',
    redirectTo: 'login'
  }
]

export const ModuloRoteamento = RouterModule.forRoot(rotas);

```

COMPONENTES, DIRETIVAS E CICLO DE VIDA

Com os componentes podemos reaproveitar o código de templates e da lógica envolvida em várias páginas da aplicação, facilitando a manutenção. Porém em alguns casos queremos apenas modificar apenas atributos de tags, ou adicionar lógica em certos momentos dos componentes. A seguir veremos quais recurso o Angular nos oferece e como utilizá-los.

17.1 CICLO DE VIDA DOS COMPONENTES

Os componentes e diretivas do Angular, são instanciados na *view* agindo de forma similar a objetos, porém aqui, estas instâncias contém gatilhos que são disparados em certos momentos da existência deste objeto, entendendo que ele pode ser criado, exibido, sofrer modificações e por fim ser destruído. Chamamos estes momentos de ganchos de ciclo de vida (*lifecycle hooks*).

Estes ganchos na verdade tratam-se de funções nativas e de uso opcional que Angular fornece, e que podemos usar na classe de um componente ou diretiva. Estas funções representam cada momento que uma instância está passando, onde podemos então elaborar alguma atividade ou lógica para ser executada.

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

17.2 SEQUÊNCIA DO CICLO DE VIDA

Após um componente ou diretiva ser chamado por seu método construtor, o Angular começa a chamar e executar os métodos do ciclo de vida de seus momentos específicos na seguinte sequência:

- `ngOnChanges()` : momento quando o Angular redefine o valor de alguma propriedade do tipo `input`. Método recebe um objeto `SimpleChanges` que contém a referência para o valor antigo e para o novo.
Este método é executado antes do `ngOnInit` e toda vez que uma propriedade muda de valor.
- `ngOnInit()` : inicializa a diretiva ou componente assim que o Angular exibir os valores passados para as propriedades, e definir os valores de propriedades `input`.
É executado uma única vez depois do primeiro `ngOnChanges()`.
- `ngDoCheck()` : detecta todo evento que acontece na `view` e dispara neste momento. É disparado também após os primeiros `ngOnChanges()` and `ngOnInit()` e depois a cada detecção de mudança (*change detection*).
- `ngAfterContentInit()` : referente ao momento depois que o Angular projeta conteúdo externo dentro da `view` de um componente ou da `view` que a diretiva está. É executado uma única vez após o primeiro `ngDoCheck()`.
- `ngAfterContentChecked()` : referente ao momento após o Angular o verificar se o conteúdo do componente/diretiva foi projetado. É chamado após o `ngAfterContentInit()` e depois a cada

ngDoCheck() subsequente.

- ngAfterViewInit() : é o momento depois que o angular inicializa a view do componente e dos seus componentes filhos, ou que a view que a diretiva está presente. É chamado uma única vez depois do primeiro ngAfterContentChecked() .
- ngAfterViewChecked() : é quando o Angular verifica a view do componente e de seus componentes filhos, ou que a view que a diretiva está presente. É chamada depois de ngAfterViewInit() e a cada ngAfterContentChecked() subsequente.
- ngOnDestroy() : "Limpa tudo" assim que o Angular for acionado para destruir o componente ou diretiva. Faz unsubscribe de Observables e tira escutadores de eventos para evitar vazamento de memória (*memory leaks*). É chamada no momento antes do antes do Angular remover completamente as referências à diretiva ou componente.

Cada método do ciclo de vida contém também sua própria interface a ser implementada, por exemplo a OnInit que já vem pronta para ser usada quando geramos um componente pela CLI:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<h1>{{title}}</h1>`
})
export class AppComponent implements OnInit {

  title = '';

  constructor() { }

  ngOnInit() {
    this.title = 'Welcome!';
  }
}
```

17.3 INTERFACES NO TYPESCRIPT

Para implementar corretamente os métodos de ciclo de vida para um componente ou diretiva, o Angular também oferece uma interface para cada momento do ciclo. As interfaces são um recurso disponível pelo TypeScript, permitindo usar mais conceitos de orientação à objetos na aplicação.

Com as interfaces, é possível criar padrões para as classes, onde todas compartilham uma arquitetura em comum, porém sendo permitido que elas contenham suas especificidades. **Por exemplo:**

```
export interface School {

  name: string;

  enroll(id: number): boolean;
```

```
}
```

Agora uma classe de uma escola específica implementando a interface:

```
import { School } from './school.interface';

export class Caelum implements School {
  name: 'Caelum';
  address: 'Rua Vergueiro, 3185';

  enroll(id: number) {
    //alguma lógica
    return true
  }
}
```

17.4 DIRETIVAS

As diretivas são funcionalidades de template que adicionamos dentro dos elementos do template como atributos das tags, podendo receber ou não valores;

Existem diretivas que já vem prontas com o Angular como o `*ngFor`, e também a `[(ngModel)]`; Porém é possível construirmos nossas próprias diretivas que podem nos auxiliar a definir valores padrão para atributos de elementos, como um conjunto de classes etc.

Para criar uma diretiva basta usar o *decorator Directive* imediatamente antes de uma classe:

```
import { Directive } from '@angular/core';

@Directive({
  selector: '[cmailFormField]',
})
export class CmailFormFieldDirective {

  constructor() { }

}
```

Entendendo o `selector`

O `selector` de uma diretiva deve ser preferencialmente um atributo, por isto o nome dele está envolvido entre colchetes, pois é assim que é a sintaxe de seletor de atributo no CSS. Podemos passar o nome de uma tag, ou classe, porém o mais recomendado é criar nossos próprios atributos e usarmos isto como seletor, pois caso contrário, toda tag ou classe, que foi informada do `selector` será modificada pela diretiva.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

17.5 USANDO INJEÇÃO DE DEPENDÊNCIA NO ANGULAR

Injeção de dependência, em inglês *dependency injection* (DI), é um *design pattern* importante para aplicações. O Angular tem seu próprio framework para DI, fazendo aumentar a eficiência do uso de código e auxilia na modularidade e desacoplamento, testes, manutenção etc.

As dependências são serviços ou objetos de uma classe que trazemos à uma outra classe para que possamos utilizar suas funcionalidades. A classe que tem uma dependência injetada, solicita uma "instância pronta" da dependência, não sendo necessário a própria classe saber como construir (instânciar) a dependência. Aqui, a própria aplicação instancia os objetos injetáveis para nós e nos fornece para usá-los, assim só é necessário trazermos a sua referência à classe que necessita deste objeto.

Com o Angular, declaramos as dependências de uma classe nos parâmetros do seu construtor, então quando esta classe é instanciada, as dependências são fornecidas pelo framework, à ela. **Por exemplo:**

```
//código anterior omitido
export class CadastroComponent {

  constructor(private service: CadastroService){}

  cadastrar(){
    this.service.enviar();
  }
}
```

No exemplo acima, temos o trecho de código de um componente, onde no seu construtor criarmos uma propriedade da classe chamada `service`, esta deixou de ser apenas um simples parâmetro do construtor e se tornou uma propriedade da classe devido ao modificador de acesso `private` informado

à sua esquerda, e à direita, seguido de dois pontos, definimos o tipo como `CadastroService`, uma classe que foi construída para ser injetada, e quando definimos o tipo de um parâmetro do construtor de uma classe, estamos fazendo **injeção de dependência** da classe do tipo, sendo seu objeto guardado na referência `service`.

Tornamos `service` uma propriedade da classe para que o mesmo possa ser usado em outros momentos da classe `CadastroComponente` e não apenas no escopo do construtor.

17.6 ACESSO À ELEMENTOS DO DOM COM ELEMENTREF

Quando trabalhamos com o desenvolvimento de páginas, existem vários objetos que necessitamos acessar para obter valores, informações, métodos etc. Um destes objetos pode ser um elemento nativo do DOM que desejamos adicionar uma classe, ou modificar um valor de forma programática (e não direto no template).

Para isto, o Angular criou diferentes classes que podemos usar para as mais diversas atividades. E no caso, para acessarmos o objeto de um elemento do DOM usamos a classe `ElementRef`. Esta classe não deve ser instânciada pois ela representa o objeto do DOM do componente ou diretiva onde ela é chamada, ou seja seu objeto já foi construído e já existe, precisamos apenas da referência a este objeto, portanto para usá-lo precisamos injetá-lo como uma dependência da classe. **Por exemplo:**

```
import { Directive, ElementRef, OnInit } from '@angular/core';

@Directive({
  selector: '[cmailFormField]'
})
export class CmailFormFieldDirective implements OnInit {

  constructor(private campo: ElementRef) { }

  ngOnInit() {
    const campo = this.campo.nativeElement;

    if(campo.name) {
      campo.id = campo.name;
      campo.setAttribute('placeholder', ' ');
      campo.classList.add('mdl-textfield_input');
    }
    else {
      throw new Error("Atributo 'name' é obrigatório");
    }
  }
}
```

Na diretiva acima, usamos o `ElementRef` para que acesse o objeto do elemento onde a diretiva está presente. Injetamos ela no construtor e sua referência é dada através da propriedade da classe que criamos para isto, a `campo`.

No método de ciclo de vida, o `ngOnInit`, acessamos o elemento do DOM, através do atributo `nativeElement`, e então modificamos suas propriedades, adicionamos classes, da mesma forma que no JavaScript.

17.7 INCLUINDO TEMPLATES DINÂMICOS COM NG-CONTENT

O `<ng-content></ng-content>` é um elemento disponibilizado pelo Angular, que funciona como uma espécie de *placeholder* para conteúdos externos, ou seja, ele faz a inclusão no template do componente de tudo que estiver entre as tags da instância do componente (do nosso *custom element*). Permitindo este ter um template muito mais flexível e funcionando como um elemento HTML normalmente funciona.

Caso tentarmos inserir conteúdo entre as tags de um *custom element*, por padrão este conteúdo não é exibido, pois não tem como o Angular saber onde colocar este conteúdo em relação ao template já feito para o componente, então `ng-content` tem o papel de indicar este local no template do componente.

Por exemplo:

```
import { Component } from '@angular/core';

@Component({
  selector: 'banner',
  template: `
    <figure>
      <ng-content></ng-content>
    </figure>
  `
})
export class BannerComponent {}

import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <banner>
      <h2>Titulo do banner</h2>
      
    </banner>

    <banner>
      
      <figcaption>Novidades</figcaption>
    </banner>
  `
})
export class AppComponent {}
```

Referências

1. <https://angular.io/guide/lifecycle-hooks>
2. <https://www.typescriptlang.org/docs/handbook/interfaces.html>
3. <https://angular.io/guide/dependency-injection-in-action>

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

EXERCÍCIO: CADASTRO DE CONTAS - UM COMPONENTE PARA FORMULÁRIOS MAIS EFICIENTE

18.1 OBJETIVO

Vamos começar a desenvolver o formulário de cadastro. E como vimos, o código para fazer 1 campo do formulário é bem extenso e trabalhoso... Então vamos transformá-lo em um componente, para facilitar a manutenção.

The screenshot shows a web application interface for 'Cmail'. On the left, there's a sidebar with icons for 'Login', 'Cadastro', 'Caixa de Entrada', and 'Logout'. The main area has a blue header bar with the text 'Título da página atual'. Below it is a registration form with fields for 'Nome' (Name), 'Username', 'Senha' (Password), and 'Avatar'. A red 'CADASTRAR' (Register) button is at the bottom right of the form. To the left of the form, there's a placeholder for an 'Avatar' with the text 'Pessoa de Tal' and an email address 'pesso.al@cmail.br'.

Figura 18.1: Formulário de cadastro

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

18.2 PASSO A PASSO COM CÓDIGO

Para que não seja necessário digitar o código dos arquivos de HTML ou CSS, baixe-os a partir do repositório dos arquivos do curso, pasta **07** :

1. Vamos adicionar um template inicial para a **página de cadastro**:

`#./src/app/modules/cadastro/cadastro.component.html`

```
<div class="mdl-layout mdl-js-layout mdl-layout--fixed-drawer mdl-layout--fixed-header">
<cmail-header></cmail-header>
<main class="mdl-layout__content">
  <div class="page-content">
    <div class="mdl-grid">
      <form>

        <div class="mdl-card__actions">
          <button class="mdl-button mdl-js-button mdl-button--raised mdl-button--accent">
            Cadastrar
          </button>
        </div>

      </form>
    </div>
  </main>
</div>
```

2. Antes de criar o novo componente vamos fazer alguns ajustes no **Schematics** do projeto, para geração de componentes e diretivas:

`#./angular.json:11`

```
"schematics": {
```

```

"@schematics/angular:component": {
  "prefix": "cmail",
  "inlineStyle": true
},
"@schematics/angular:directive": {
  "prefix": "cmail"
}
},

```

Nesta configuração, definimos o prefixo dos selectores para os componentes e diretivas. E no caso dos componentes, pulamos a criação de um arquivo separado para o CSS.

3. Inspirando-se no modelo que a biblioteca **Angular Material** criou o componente para *inputs*, vamos fazer o nosso:

<https://material.angular.io/components/input/overview>

Com a extensão Schematics vamos gerar o componente **cmail-form-group** :

```

ng g component components/cmail-form-group
CREATE src/app/components/cmail-form-group/cmail-form-group.component.html (35 bytes)
CREATE src/app/components/cmail-form-group/cmail-form-group.component.spec.ts (686 bytes)
CREATE src/app/components/cmail-form-group/cmail-form-group.component.ts (272 bytes)
UPDATE src/app/app.module.ts (981 bytes)

```

Figura 18.2: Output da geração do novo componente

4. O template de **CmailFormGroupComponent** conterá toda a estrutura necessária para formar um campo de entrada, porém este campo pode ser uma hora um `<input>`, outra hora um `<textarea>` etc, portanto no local do elemento de formulário deixaremos uma lacuna, representada pela tag `<ng-content></ng-content>`, permitindo ser exibido outros elementos dentro do template do componente:

`#./src/app/components/cmail-form-group/cmail-form-group.component.html`

```

<div class="cmailInputForm">
  <div class="mdl-textfield mdl-textfield--floating-label">
    <ng-content></ng-content>
    <label class="mdl-textfield__label" [for]="idCampo"> {{ textoDaLabel }} </label>
    <span class="mdl-textfield__formline"></span>
  </div>
</div>

```

Perceba também que o `<label>` do campo exibe uma variável `textoDaLabel` como conteúdo, e `idCampo` como valor do atributo `for`, ambos deverão ter seus valores preenchido na classe do componente, sendo estes propriedades da mesma.

5. Na classe, declaramos as propriedades `textoDaLabel` e `idCampo`. Depois injetamos no construtor o tipo **ElementRef**, pois com ele temos a capacidade de buscar por elementos do DOM e usar suas propriedades e valores na classe Typescript. Em seguida usamos o método **ngOnInit**, que faz parte

do **ciclo de vida** (*Lifecycle Hooks*) dos componentes Angular. Este é executado logo após o construtor, momento quando o componente é inicializado, podendo assim executar alguma lógica, neste caso, que busca elementos em seu template:

```
#./src/app/components/cmail-form-group/cmail-form-group.component.ts

import { Component, ElementRef, OnInit } from '@angular/core';

@Component({
  selector: 'cmail-form-group',
  templateUrl: './cmail-form-group.component.html',
  styles: []
})
export class CmailFormGroupComponent implements OnInit {

  textoDaLabel = '';
  idCampo = '';

  constructor(private elemento: ElementRef) {}

  ngOnInit() {
    const campo = this.elemento.nativeElement.querySelector('input')
    this.textoDaLabel = campo.name.replace(campo.name[0], campo.name[0].toUpperCase());
    this.idCampo = campo.name;
  }
}
```

6. Agora podemos testar o **CmailFormGroupComponent** no template de **CadastroComponente**, inserimos entre as tags `<cmail-form-group>` o `<input>` e o `` com a mensagem de validação:

```
#./src/app/modules/cadastro/cadastro.component.html

<div class="mdl-layout mdl-js-layout mdl-layout--fixed-drawer mdl-layout--fixed-header">
  <cmail-header></cmail-header>
  <main class="mdl-layout__content">
    <div class="page-content">
      <div class="mdl-grid">
        <form>

          <cmail-form-group>
            <input autofocus required type="text" name="nome" id="nome" placeholder=" " class="mdl-textfield__input">
            <span class="mdl-textfield__error">Informar um nome é obrigatório!</span>
          </cmail-form-group>

          <div class="mdl-card__actions">
            <button class="mdl-button mdl-js-button mdl-button--raised mdl-button--accent">
              Cadastrar
            </button>
          </div>

        </form>
      </div>
    </main>
  </div>
```

Muito bem! Agora antes de fazermos os próximos campos, podemos pensar numa maneira de facilitar o trabalho de configurar os atributos obrigatórios que são repetitivos em um campo, como o `id`, `placeholder=""` e `class="mdl-textfield__input"`. Para isto, vamos utilizar os poderes de uma **diretiva**!

7. Na pasta de **CmailFormGroupComponent** crie um novo arquivo chamado `cmail-form-field.directive.ts`. E nele faça a estrutura básica da classe para uma diretiva:

```
#./src/app/components/cmail-form-group/cmail-form-field.directive.ts
```

```
import { Directive } from '@angular/core';

@Directive({
  selector: '[cmailFormField]'
})
export class CmailFormFieldDirective {

  constructor() { }

}
```

Também é possível usar a CLI para gerar com o comando:

```
ng g directive components/cmail-form-group/cmail-form-field
```

Lembrando que as diretivas também devem ser declaradas por um módulo para poderem ser utilizadas!

8. Usando **ElementRef** para modificar o atual elemento no qual a diretiva foi aplicada, e com base no valor do atributo `name`, definimos o valor dos outros atributos.

```
#./src/app/components/cmail-form-group/cmail-form-field.directive.ts
```

```
import { Directive, ElementRef, OnInit } from '@angular/core';

@Directive({
  selector: '[cmailFormField]'
})
export class CmailFormFieldDirective implements OnInit {

  constructor(private campo: ElementRef) { }

  ngOnInit() {
    const campo = this.campo.nativeElement;

    campo.id = campo.name;
    campo.setAttribute('placeholder', '');
    campo.classList.add('mdl-textfield__input');
  }
}
```

9. Porém, se o atributo `name` não for definido, vamos ter problemas, portanto melhor garantir isto fazendo uma condição que verifica a existência do atributo:

```
#./src/app/components/cmail-form-group/cmail-form-field.directive.ts

import { Directive, ElementRef, OnInit } from '@angular/core';

@Directive({
  selector: '[cmailFormField]'
})
export class CmailFormFieldDirective implements OnInit {

  constructor(private campo: ElementRef) { }

  ngOnInit() {
    const campo = this.campo.nativeElement;

    if(campo.name) {
      campo.id = campo.name;
      campo.setAttribute('placeholder', ' ');
      campo.classList.add('mdl-textfield__input');
    }
    else {
      throw new Error("Atributo 'name' é obrigatório");
    }
  }
}
```

10. Agora faça os campos para outros dados que precisaremos para o formulário de cadastro: usuário, senha e avatar.

Deverá ficar desta maneira:

```
#./src/app/modules/cadastro/cadastro.component.html

<cmail-form-group>
  <input autofocus required type="text" name="nome" cmailFormField>
  <span class="mdl-textfield_error">Informar um nome é obrigatório!</span>
</cmail-form-group>

<cmail-form-group>
  <input required type="text" name="username" cmailFormField>
  <span class="mdl-textfield_error">Informar um nome de usuário é obrigatório!</span>
</cmail-form-group>

<cmail-form-group>
  <input required type="password" name="senha" cmailFormField>
  <span class="mdl-textfield_error">Informar uma senha é obrigatório!</span>
</cmail-form-group>

<cmail-form-group>
  <input type="text" name="avatar" cmailFormField>
</cmail-form-group>
```

INTRODUÇÃO AOS FORMULÁRIOS REATIVOS

Formulários reativos provém uma abordagem orientada à modelo para lidar com entradas de formulários nos quais os valores vão mudando com o tempo. Vamos aqui ver como criar e atualizar um formulário simples até opções mais avançadas e conhecer as vantagens desta abordagem.

19.1 A ABORDAGEM DE FORMULÁRIO REATIVOS

Os formulários reativos usam uma abordagem explícita e imutável para gerenciar o estado de um formulário em um determinado ponto no tempo. Cada alteração no estado do formulário retorna um novo estado, que mantém a integridade do modelo entre as alterações. Formulários reativos são construídos em torno de fluxos observáveis (*observable streams*), onde as entradas de formulário e valores são fornecidos como fluxos de valores de entrada, no qual podem ser acessados de forma síncrona.

Esta abordagem facilita os testes automatizados, pois você tem a garantia de que seus dados são consistentes e previsíveis quando forem solicitados. Todos que consumirem os fluxos de dados, têm acesso para manipular esses dados com segurança.

Qual abordagem usar?

Formulários reativos fornecem mais previsibilidade com acesso síncrono ao modelo de dados, imutabilidade com operadores observáveis e rastreamento de mudanças por meio de fluxos observáveis.

Se você preferir o acesso direto para modificar dados no template, os formulários orientados à template serão menos explícitos, pois dependem de diretivas incorporadas no modelo, além de dados mutáveis para acompanhar as alterações de forma assíncrona.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

19.2 CRIANDO UM FORMULÁRIO REATIVO

A primeira coisa que temos que fazer é importar o módulo de formulários reativos na aplicação o `ReactiveFormsModule`. Lembrando que a importação deste módulo deve ser feita dentro do módulo da página atual, ou caso a aplicação não esteja usando *lazy-load* então o módulo pode ser importado em `AppModule`, **por exemplo**:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { CadastroComponent } from './cadastro.component';
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  declarations: [CadastroComponent],
  exports: [CadastroComponent],
  imports: [
    CommonModule,
    ReactiveFormsModule
  ]
})
export class CadastroModule {}
```

Este módulo, irá nos trazer também uma série de classes e diretivas para nos ajudar no momento de trabalhar com os formulários.

19.3 EXPLORANDO A API DE FORMULÁRIOS REATIVOS

É possível construir um formulário com apenas um campo ou mais com mais, vamos entender o que difere na construção dos dois:

Um campo na classe: `FormControl()` e `formControl`

Caso seu formulário tenha apenas um campo, basta criar uma propriedade da classe que recebe como

valor uma instância de `FormControl('')`. Depois basta associar a propriedade com o campo no template, através da diretiva `FormControl`, **por exemplo**:

```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'cmail-cadastro',
  template: `
    <label for="email">Cadastre-se!</label>
    <input
      [FormControl]="email"
      id="email"
      placeholder="nome@seumail.com">
    <button (click)="cadastrar()">enviar</button>

    {{email.value}}
  `,
})
export class CadastroComponent {

  email = new FormControl('');

  cadastrar(){
    console.log(this.email.value);
  }

}
```

Neste exemplo criamos a propriedade `email`, que recebe uma instância de `FormControl('')`. O parâmetro passado é o valor inicial do campo, no caso, uma *string* vazia.

Depois na tag `input` adicionamos a diretiva `[FormControl]` que será associada com o objeto do modelo, no caso a propriedade da classe `email`.

Usando a expressão `{{email.value}}` conseguimos já verificar o valor do campo quando digitamos. Depois no botão associamos o envio dos dados com um método da classe.

Um formulário na classe: `FormGroup()` e `formControlName`

Agora se o formulário contiver mais de um campo o ideal é criar um grupo de formulário, ou seja, um objeto que contém controles de formulário. Este agrupamento é muito útil, principalmente para as validações.

Na classe do componente criamos uma atributo que guardará a instância `FormGroup({})`, este recebe como parâmetros um objeto literal, onde cada propriedade será a chave para acessar o objeto de um campo do formulário.

No exemplo abaixo, apenas a instância de `FormGroup({})`:

```
import { Component } from '@angular/core';
import { FormGroup } from '@angular/forms';
```

```

@Component({
  selector: 'cmail-cadastro',
  templateUrl: './cadastro.component.html'
})
export class CadastroComponent {

  formCadastro = new FormGroup({});

}

```

Agora adicionamos as propriedades do objeto com o nome de cada campo que teremos no formulário, onde o valor de cada campo é uma instância de `FormControl`:

```

import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'cmail-cadastro',
  templateUrl: './cadastro.component.html'
})
export class CadastroComponent {

  formCadastro = new FormGroup({
    nome: new FormControl(''),
    email: new FormControl(''),
    senha: new FormControl('')
  });
}

}

```

No template, na tag

adicionamos a diretiva `[formGroup]` que fará a associação naquele elemento HTML com a propriedade da classe "formCadastro".

E em cada campo do formulário adicionamos a propriedade `name` do HTML solicitada pela diretiva `FormControlName` que deve receber exatamente o nome da propriedade passada na instância de `FormGroup` na classe, associando então o elemento HTML com o objeto `FormControl`.

```

<form [formGroup]="formCadastro" (ngSubmit)="cadastrar()">
  <div>
    <label for="nome">Nome:</label>
    <input type="text" id="nome" name="nome" formControlName="nome">
  </div>

  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" formControlName="email">
  </div>

  <div>
    <label for="senha">Senha:</label>
    <input type="password" id="senha" name="senha" formControlName="senha">
  </div>

  <button>Enviar</button>
</form>

```

19.4 OBTENDO OS DADOS DO FORMULÁRIO

Podemos o evento `(ngSubmit)` para chamar um método da classe, da mesma forma que os formulários template-drive:

```
<form [formGroup]="formCadastro" (ngSubmit)="cadastrar()">
```

Para exibir os dados do formulário basta acessarmos a propriedade `value` do objeto `FormGroup`:

```
cadastrar(){
  console.log(this.formCadastro.value);
}
```

Referências

1. <https://angular.io/guide/reactive-forms>

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

EXERCÍCIO: CADASTRO DE CONTAS - FORMULÁRIOS REATIVOS

20.1 OBJETIVO

Usando a abordagem de **formulários reativos** do Angular, vamos aprender como fazer que as regras de validação venham à partir da classe, e não sejam baseadas apenas nas regras do template (atributos dos campos). Esta abordagem possibilita fazermos regras mais avançadas de validação. Neste primeiro passo, vamos preparar o terreno.

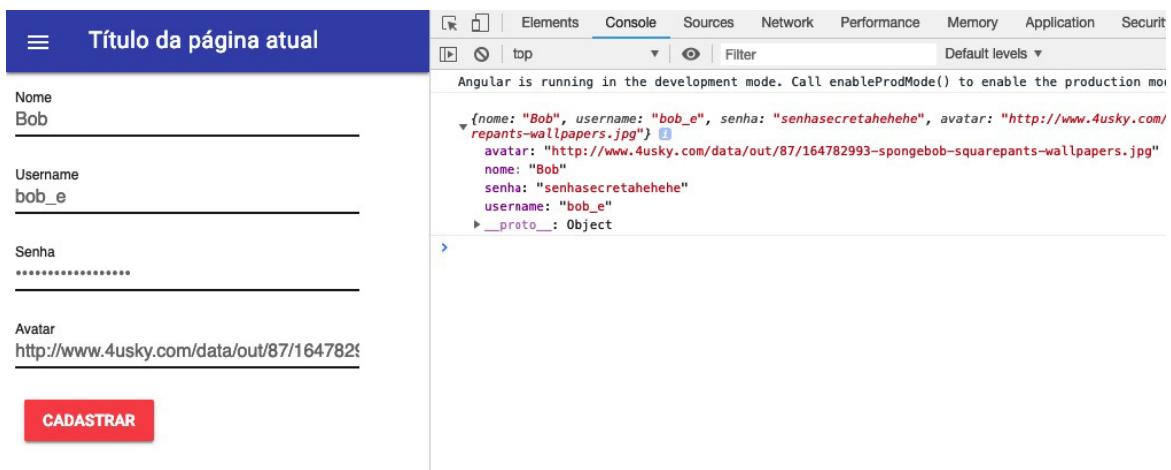


Figura 20.1: Formulários Reativos exibindo valores

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

20.2 PASSO A PASSO COM CÓDIGO

1. Primeiro vamos ter que criar a representação do formulário como um objeto javascript. Esta representação, na prática é um atributo da classe, o **formCadastro**, que receberá uma instância de **FormGroup**. No momento da construção de FormGroup, passamos um objeto, onde cada propriedade deste objeto representa um campo do formulário, sendo estas propriedades, instâncias de **FormControl**:

`#./src/app/modules/cadastro/cadastro.component.ts`

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-cadastro',
  templateUrl: './cadastro.component.html',
  styleUrls: ['./cadastro.component.css']
})
export class CadastroComponent implements OnInit {

  formCadastro = new FormGroup({
    nome: new FormControl(),
    username: new FormControl(),
    senha: new FormControl(),
    avatar: new FormControl(),
  })

  constructor() {}

  ngOnInit() {}
}
```

2. Agora precisamos relacionar a propriedade **formCadastro** com a tag `<form>` do template, para que **formCadastro** represente definitivamente o formulário no contexto da classe. Para isto utilizaremos a diretiva `[FormGroup]` :

```
#./src/app/modules/cadastro/cadastro.component.html:6
```

```
<form [FormGroup]="formCadastro">
```

Só que encontramos um problema, esta diretiva não é reconhecida nativamente pelo Angular. Pois ela pertence ao Módulo de Formulário Reativos do Angular, então vamos importar ele em **AppModule** para finalmente podermos utilizar os recursos desse módulo:

```
#./src/app/app.module.ts:25
```

```
imports: [
  BrowserModule,
  FormsModule,
  ModuloRoteamento,
  ReactiveFormsModule //Importado aqui!
]
```

3. Vamos precisar agora associar cada campo do formulário com a classe, para isto usamos a diretiva **FormControlName**, que recebe como valor o nome da propriedade do objeto criado na instância **FormGroup**:

```
#./src/app/modules/cadastro/cadastro.component.html:9
```

```
<input formControlName="nome" autofocus required type="text" name="nome" cmailFormField>
```

```
#./src/app/modules/cadastro/cadastro.component.html:14
```

```
<input formControlName="username" required type="text" name="username" cmailFormField>
```

```
#./src/app/modules/cadastro/cadastro.component.html:19
```

```
<input formControlName="senha" required type="password" name="senha" cmailFormField>
```

```
#./src/app/modules/cadastro/cadastro.component.html:24
```

```
<input formControlName="avatar" type="text" name="avatar" cmailFormField>
```

4. Vamos fazer um método em **CadastroComponent** que exibirá no console os dados dos campos, caso o formulário seja submetido e os campos estejam válidos:

```
#./src/app/modules/cadastro/cadastro.component.ts:23
```

```
handleCadastrarUsuario() {
  if(this.formCadastro.valid){
    console.log(this.formCadastro.value);
  }
  else {
    console.log('Campos precisam ser preenchidos!')
}
```

```
 }  
 }
```

5. Para invocar o métodos, associamos ele com o evento **ngSubmit** no template:

```
#./src/app/modules/cadastro/cadastro.component.html:6  
  
<form [formGroup]="formCadastro" (ngSubmit)="handleCadastrarUsuario()">
```

Ótimo! Conseguimos ver os dados no console, porém ainda não informamos quais campos são obrigatórios, vamos ver como, no próximo capítulo.

VALIDAÇÕES NOS FORMULÁRIOS REATIVOS

As validações em um formulário são importantes para verificar o preenchimento e garantir a qualidade geral da entrada de dados feita pelo usuário, para que estes dados estejam completos, corretos, exatos e íntegros.

Vamos ver aqui como validar estas inserções de dados feita pelo usuário através da interface usando formulários reativos, e como exibir mensagens de validação úteis.

21.1 VALIDAÇÃO NATIVA FUNCIONA

Nos formulários reativos, é importante utilizar os atributos de validação nativa do HTML, pois eles fornecem maiores detalhes de acessibilidade para os usuários. Portanto é importante de qualquer maneira informar estes atributos, como o `required`.

```
<input required minlength="2" formControlName="nome" type="text" id="nome" name="nome">
```

Aqui usamos os atributos nativos `required` e `minlength`, que passam para a instância de `FormGroup` existem campo preenchidos incorretamente, portanto o formulário está inválido.

Propriedades de validação do `FormGroup`

É possível obter este status do formulário acessando suas propriedades `valid` ou `invalid`, que retornam um booleano, possibilitando fazermos operações lógicas com estes estados, **por exemplo:**

```
cadastrar(){

  console.log('é valido?', this.formCadastro.valid);

  if(this.formCadastro.invalid) {
    console.error('Form inválido! ->', this.formCadastro.invalid);
    return;
  }

  console.log(this.formCadastro.value);
}
```

Neste exemplo, caso o formulário estiver inválido, exibimos uma mensagem de erro no `console` e fazemos um `return` que impede a execução do restante do código dentro do método.

Outra lógica que podemos fazer, é no template, desabilitando o botão do enviar do formulário, caso o mesmo esteja inválido, usando o atributo `disabled` nativo do HTML:

```
<button [disabled]="formCadastro.invalid" >Enviar</button>
```

Aqui envolvemos o atributo nativo do HTML com colchetes, pois ele receberá seu valor booleano acessando um objeto, no caso, `formCadastro`, que por sua vez tem a propriedade `invalid` que guarda `true` ou `false`, atendendo o funcionamento do atributo.

Para conhecer todos os atributos e métodos de `FormGroup`, consulte os detalhes da documentação [2].

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

21.2 FORM VALIDATORS

Podemos e devemos usar os atributos do HTML para a acessibilidade, porém não temos detalhes dos erros de validação para exibir mensagens apropriadas com os detalhes dos erros para o usuário.

Para isso devemos também informar as validações de cada controle do formulário em sua instância de `FormControl`. O segundo parâmetro de `FormControl` é onde passamos as validações síncronas. As validações de um campo podem ser referenciadas através da classe `Validators`, que tem métodos estáticos, ou seja, não há necessidade de instânciar nem injetar a classe `Validators`.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'cmail-cadastro',
  templateUrl: './cadastro.component.html'
})
```

```

export class CadastroComponent {

  formCadastro = new FormGroup({
    nome: new FormControl('', Validators.required),
    email: new FormControl('', Validators.required),
    senha: new FormControl('', Validators.required)
  });

  //...
}

```

Propriedades de validação de um FormControl

O `FormControl` é uma extensão de `AbstractControl`, portanto as principais propriedades que usamos para obter status do controle (do campo) é uma herança de `AbstractControl`, **todas são apenas para consulta** (read-only), não podendo ter seus valores sobreescritos.

Abaixo, listamos algumas propriedades que causam dúvida sobre suas diferenças e que podem ser de grande ajuda na hora de exibir mensagens de validação:

- **`status : string`**

É o `status` de validação daquele controle. Existem **quatro** valores possíveis de status, são eles:

- `VALID` : O controle passou por todas validações;
- `INVALID` : O controle falhou em pelo menos uma validação;
- `PENDING` : O controle está no meio de uma validação (usado principalmente para validações assíncronas que podem demorar mais tempo para retornar um status);
- `DISABLED` : O control é isento de validações;

Todos os valores são mutualmente exclusivos, portanto um controle não pode ser `VALID` e `INVALID`, ou `INVALID` e `DISABLED` ;

- **`valid : boolean`**

Um controle é válido quando seu `status` é `VALID` .

- **`invalid : boolean`**

Um controle é inválido quando seu `status` é `INVALID` .

- **`pending : boolean`**

Um controle é inválido quando seu `status` é `PENDING` .

- **`pristine : boolean`**

Um controle é `pristine` se o usuário ainda **não mudou o seu valor** através da interface (UI).

- **`dirty : boolean`**

Um controle é `dirty` se o usuário **mudou o seu valor** através da interface (UI).

- **`touched : boolean`**

Será verdade `true` se o controle é marcado como `touched` (tocado). Um controle é marcado como `togado` se o usuário disparou um evento `blur` (quando saiu de foco).

Mudando os valores das validações

Como as propriedades que guardam os status de validação são `read-only`, caso seja necessário mudar os valores delas programaticamente, o `AbstractControl` nos trás métodos para realizar esta atividade, por exemplo os métodos: `markAsTouched()`, `markAsUntouched()`, `markAsDirty()`, `markAsPristine()`.

Na versão 8 do Angular, foi lançado um método novo na API de `AbstractControl`, o `markAllAsTouched`. Este método é muito útil caso o usuário submeta o formulário com vários campos inválido. Pois para a exibição de mensagens de validação, geralmente combinamos o estado de `touched` com mais outra regra do campo, e este método dispara `markAsTouched()` para todos os campos daquele formulário específico. **Por exemplo:**

```
cadastrar(){

  if(this.formCadastro.invalid) {
    this.formCadastro.markAllAsTouched();
    return;
  }

  console.log(this.formCadastro.value);

}
```

Percorrendo objetos como uma lista

Caso esteja trabalhando em uma versão do Angular inferior a 8, uma maneira de fazer o mesmo efeito do método `markAllAsTouched`, é chamar o tipo ancestral `Object` e seu método `keys()`, que recebe um objeto como parâmetro e retorna um `array`, onde com isto conseguimos percorrer a lista de controles, e a cada iteração disparar o método `markAsTouched()`. **Por exemplo:**

```
Object.keys(form.controls).forEach(field => {
  const control = form.get(field);
  control.markAsTouched({ onlySelf: true });
})
```

É possível verificar todos os métodos e seus detalhes na referência na documentação [6].

21.3 CRIANDO ATRIBUTOS DO COMPONENTE

De acordo com as boas práticas para criação de *web components*, cada componente criado deve ter sua própria tag, ou seja seu próprio elemento HTML, sendo este um ***custom element***. Assim como é possível criar um elemento HTML completamente novo, podemos também criar atributos personalizados deste elemento, possibilitando o envio de valores de um componente para o outro.

Para isto, devemos criar propriedade da classe do tipo `input` (*input property*), para isto devemos decorá-las com `@Input()`, que fará com que aquela propriedade se torne também um atributo do elemento do componente, possibilitando usar este valor para dentro da classe e no template do componente. **Por exemplo:**

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'cmail-form-group',
  templateUrl: './form-group.component.html'
})
export class FormGroupComponent {

  @Input() label = "";

  constructor() { }

}
```

No exemplo acima, criamos a propriedade `label` e graças ao uso do decorator `@Input` a propriedade também pode ser usada como um atributo, **por exemplo**:

```
<cmail-form-group label="Endereço"></cmail-form-group>
```

Enviado um objeto para um atributo

É possível passar um objeto inteiro como valor de uma propriedade `input`.

No exemplo abaixo criamos a propriedade `validation`, e definimos o tipo dela como `AbstractControl`, para que possamos pegar os atributos de validação no `template`.

```
import { Component, Input } from '@angular/core';
import { AbstractControl } from '@angular/forms';

@Component({
  selector: 'cmail-form-group',
  templateUrl: './form-group.component.html'
})
export class FormGroupComponent {

  @Input() label = "";
  @Input() validation: AbstractControl;

  constructor() { }

}
```

Agora quem chamar `FormGroupComponent` vai poder passar um objeto do mesmo tipo por atributo, lembrando de usar o colchetes na volta do atributo para que o *bind* do objeto seja feito:

```
<cmail-form-group [validation]="this.address" label="Endereço"></cmail-form-group>
```

Alias para atributos

Podemos também criar um *alias* (um nome alternativo) para uma propriedade *input* para ajudar no preenchimento no *template*, onde o a propriedade na classe pode ter um nome, e na hora de usar o atributo no elemento do componente, podemos definir outro nome. **Por exemplo:**

```
import { Component, Input } from '@angular/core';
import { AbstractControl } from '@angular/forms';

@Component({
  selector: 'cmail-form-group',
  templateUrl: './form-group.component.html'
})
export class FormGroupComponent {

  @Input() label = "";
  @Input('campo') validation: AbstractControl;

  constructor() { }

}
```

Perceba que no decorador `@Input` da propriedade `validation` passamos um parâmetro '`'campo'`', uma *string* que será seu *alias* na hora de utilizar o atributo no elemento customizado. Veja como ficará o atributo customizado:

```
<cmail-form-group [campo]="address" label="Endereço"></cmail-form-group>
```

Elvis operator ?

Quando queremos utilizar propriedades de um objeto, que não estará disponível ou definido no tempo da execução do código, podemos ter problemas. Por exemplo:

```
let casa;
console.log(casa.endereco);
```

Isto retornará:

```
Uncaught TypeError: Cannot read property 'endereco' of undefined
```

Pois não existe nada dentro da variável `casa` , muito menos a propriedade `endereco` .

No caso de um componente Angular, é possível consultar o valor de uma propriedade da classe que fará referência à um objeto, e que esta propriedade ainda tem seu valor `undefined` , porém sabemos que em algum momento futuro este objeto será passado como valor da propriedade. Porém como evitar que os erros sejam exibidos no `console` do navegador?

Para isto podemos usar o **operador elvis** (*elvis operator*), que nada mais é que o ponto de interrogação `? .`. Este operador é um recurso do TypeScript, e que permite usarmos referências que **estão declaradas** porém não tem seus valores definidos, sem que erros sejam exibidos do console. Veja o *template* do **exemplo abaixo**:

```
import { Component, Input } from '@angular/core';
import { AbstractControl } from '@angular/forms';
```

```

@Component({
  selector: 'cmail-form-group',
  template: `
    <p>
      O campo está válido: {{validation?.invalid}}
    </p>
    <p>
      O campo foi tocado: {{validation?.touched}}
    </p>
  `
})
export class FormGroupComponent {

  @Input() label = "";
  @Input('campo') validation: AbstractControl;

  constructor() { }

}

```

Este componente contém uma propriedade `validation` que não tem um valor definido na hora da sua declaração, apenas seu tipo, favorecendo o *autocomplete* na hora de usar ela no `template` do componente.

No `template`, usamos a expressão de *bind* do Angular para exibir um valor de uma propriedade de `validation`. Se usarmos a expressão sem o interrogação no `validation`, vamos obter erros no console dizendo que as propriedades não existem.

Referências

1. <https://angular.io/guide/reactive-forms>
2. <https://angular.io/api/forms/FormGroup>
3. <https://angular.io/guide/form-validation>
4. <https://angular.io/api/forms/FormControl>
5. <https://angular.io/api/forms/AbstractControl>
6. <https://angular.io/api/forms/AbstractControl#methods>
7. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys

EXERCÍCIO: CADASTRO DE CONTAS - MELHORANDO O FEEDBACK DE ERROS

22.1 OBJETIVO

Fizemos a configuração básica dos campos, porém ainda não informamos quais campos queremos validar. Vamos informar isto e mostrar as mensagens de validação de cada campo ao usuário.

The screenshot shows a user registration form with the following fields and validation messages:

- Nome:** The input field is empty. Below it, the error message "Informar um nome é obrigatório!" is displayed in red.
- Username:** The input field is empty. Below it, the error message "Informar um nome de usuário é obrigatório!" is displayed in red.
- Senha:** The input field is empty. Below it, the error message "Informar uma senha é obrigatório!" is displayed in red.
- Avatar:** This field is currently empty.

A red button labeled "CADASTRAR" is located at the bottom left of the form area.

Figura 22.1: Mensagens de validação no cadastro

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

22.2 PASSO A PASSO COM O CÓDIGO

1. Para cada campo que precisa de validação, passamos na construção de **FormControl**, um valor inicial para o campo (string vazia), e depois o **Validators**, que é um objeto do pacote de formulários com uma série de métodos de validação de formulários pronto, e chamamos seu método **required**:

`#./src/app/modules/cadastro/cadastro.component.ts`

```
formCadastro = new FormGroup({  
  nome: new FormControl('', [Validators.required]),  
  username: new FormControl('', [Validators.required]),  
  senha: new FormControl('', [Validators.required]),  
  avatar: new FormControl(),  
})
```

2. Para que as mensagens apareçam, precisamos colocar uma classe `is-invalid` na `<div>` que envolve o `<input>` e o `` que carrega o texto da mensagem. Porém esta classe precisa de uma condição para ser inserida, a qual é: apenas se aquele campo específico do template for inválido. Portanto o status do campo está em **CadastroComponent**, que faz instâncias de **CmailFormGroupComponent**, então o dado da validação que está na classe, terá que ser passado como um parâmetro para as instâncias, só que desta vez passaremos um dado de um componente à outro via template.

Adicione em **CmailFormGroupComponent** uma propriedade **campo**, que será em um primeiro momento uma instância de `FormControl`, e que depois vai ser sobreescrita. Esta vai receber um *decorator* `@Input()` no qual transformará esta propriedade da classe em um atributo da tag `<cmail-form-group>` podendo receber valores, neste caso o objeto do campo em si:

`#./src/app/components/cmail-form-group/cmail-form-group.component.ts`

```
//código anterior omitido ...
export class CmailFormGroupComponent implements OnInit {

  textoDaLabel = '';
  idCampo = '';
  @Input() campo = new FormControl();

//código posterior omitido ...
```

3. No template de **CmailFormGroupComponent** vamos fazer a condição para inserção da classe com o `[ngClass]` :

`#./src/app/components/cmail-form-group/cmail-form-group.component.html`

```
<div class="mdl-textfield mdl-textfield--floating-label" [ngClass]="{'is-invalid': campo.invalid &
& campo.touched }">
```

4. Em **CadastroComponent**, cada instância de `<cmail-form-group>` vai ter o atributo campo, que deverá passar o objeto que representa aquele campo no **formCadastro**:

`#./src/app/modules/cadastro/cadastro.component.html`

```
<cmail-form-group [campo]="formCadastro.get('nome')">
```

Faça para todos os campos:

username:

```
<cmail-form-group [campo]="formCadastro.get('username')">
```

senha:

```
<cmail-form-group [campo]="formCadastro.get('senha')">
```

avatar:

```
<cmail-form-group [campo]="formCadastro.get('avatar')">
```

Com isto já conseguimos ver as mensagens exibidas caso passarmos pelos campos obrigatórios sem os preencher.

5. Agora vamos fazer uma lógica para caso o formulário seja submetido e os campos estejam inválidos, todas as mensagens apareçam de uma vez. Na classe de **CadastroComponent** criamos o método `validarTodosOsCamposDoFormulario` que receberá o objeto do formulário como parâmetro, e então percorrerá todos os campos e os marcará como **touched** fazendo a validação com base neste status disparar:

`#./src/app/modules/cadastro/cadastro.component.ts`

```
validarTodosOsCamposDoFormulario(form: FormGroup) {

  Object.keys(form.controls).forEach(field => {
    const control = form.get(field);
    control.markAsTouched({ onlySelf: true });
```

```
})
}
```

6. Vamos chamar o método no **else** de **handleCadastrarUsuario** :

```
#./src/app/modules/cadastro/cadastro.component.ts
```

```
handleCadastrarUsuario() {
  if(this.formCadastro.valid){
    console.log(this.formCadastro.value);
    this.formCadastro.reset();
  }
  else {
    this.validarTodosOsCamposDoFormulario(this.formCadastro);
  }
}
```

Pronto! Agora ao submeter o formulário vazio, recebemos as mensagens de validação.

MÚLTIPLAS REGRAS DE VALIDAÇÃO

Um campo de formulário pode exigir mais regras de validação além do preenchimento obrigatório ou o tipo do campo. Como por exemplo, em um campo de senha que é necessário preencher com pelo menos oito caracteres, ou com um determinado padrão usando expressão regular.

23.1 COMBINANDO FUNÇÕES DE VALIDAÇÕES

No objeto `FormControl` de cada campo é possível fazer uma lista de validadores, ao invés de informar apenas um único método de `Validators`. Para isto, usamos um `array` e a cada posição desta lista, temos uma regra de validação, através dos métodos estáticos da classe `Validators`. **Por exemplo:**

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'cmail-cadastro',
  templateUrl: './cadastro.component.html'
})
export class CadastroComponent {

  formCadastro = new FormGroup({
    nome: new FormControl('', [
      Validators.required,
      Validators.minLength(2)
    ]),
    email: new FormControl('', [
      Validators.required,
      Validators.email
    ]),
    senha: new FormControl('', [
      Validators.required,
      Validators.minLength(8),
      Validators.pattern('^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!#$%&? "]).*$')
    ])
  });
}

//...
```

Aqui o controle `nome` além de ser um campo obrigatório deve ser preenchido com no mínimo 2 caracteres. O controle `email`, é obrigatório e precisa atender o padrão de um email (com `@` e um domínio). E a senha, é obrigatória, com mínimo de oito caracteres e precisa atender a expressão regular

informada no método `pattern`.

Composição de validadores

Uma proposta de melhorar a legibilidade deste código é declarar cada validador como uma propriedade da classe usando o método `compose`, por exemplo:

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'cmail-cadastro',
  templateUrl: './cadastro.component.html'
})
export class CadastroComponent {

  nomeValidators = Validators.compose([
    Validators.required
    ,Validators.minLength(2)
  ]);

  emailValidators = Validators.compose([
    Validators.required
    ,Validators.email
  ]);

  senhaValidators = Validators.compose([
    Validators.required
    ,Validators.minLength(8)
    ,Validators.pattern('^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!#$%&? "]).*$')
  ]);

  formCadastro = new FormGroup({
    nome = new FormControl('', this.nomeValidators),
    email = new FormControl('', this.emailValidators),
    senha = new FormControl('', this.senhaValidators)
  });

  //...
}
```

Métodos para validação

Perceba que os métodos estáticos da classe `Validators` são uma API para as mesmas regras nativas dos atributos de validação elementos de formulário do HTML. Precisamos que estas regras sejam adicionadas nas instâncias de `FormControl`, para que seja possível fazer testes de unidade no código, além de os mesmos atributos estarem presentes no HTML para que seja proporcionada a devida acessibilidade. Conheça como funcionam estes métodos:

- `min` : Validador que requer que o **valor** do controle seja **maior ou igual** ao número fornecido. O validador existe apenas como uma função e não como uma diretiva.
- `max` : Validador que exige que o **valor** do controle seja **menor ou igual** ao número fornecido. O validador existe apenas como uma função e não como uma diretiva.

- `required` : O validador que requer o controle tem um valor não vazio.
- `requiredTrue` : Validador que requer que o valor do controle seja verdadeiro. Este validador é comumente usado para campos do tipo `checkbox` que são obrigatórios.
- `email` : O validador que requer o valor do controle passa por um teste de validação de email. Existe uma diretiva `email` para o elemento HTML.
- `minLength` : Validador que requer que o **comprimento do valor** do controle seja **maior ou igual ao comprimento mínimo** fornecido. Este validador também é fornecido por padrão se você usar o atributo `minlength` do HTML.
-
- `maxLength` : Validador que requer que o **comprimento do valor** do controle seja **menor ou igual ao comprimento máximo** fornecido. Esse validador também é fornecido por padrão se você usar o atributo `maxlength` do HTML.
- `pattern` : Validador que requer que o valor do controle corresponda a um padrão de expressão regular. Este validador também é fornecido por padrão se você usar o atributo `pattern` do HTML.
- `compose` : Componha vários validadores em uma única função que retorna a união dos mapas de erro individuais para o controle fornecido.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

23.2 TRABALHANDO AS MENSAGENS DE ERRO NO TEMPLATE

No *template* é necessário apresentar uma mensagem descrevendo cada tipo de erro ao usuário, para que o mesmo possa preencher o campo corretamente. Para isto podemos usar métodos de

`FormControl` para nos retornar detalhes de cada erro.

Acessando um objeto `FormControl` com `get`

Primeiro temos que saber como acessar o objeto de um `FormControl` de um `FormGroup` pelo *template* do componente. Para isto, usamos o método `get` e passamos como parâmetro o nome do controle informado pela diretiva `formControlName` e também na instância de `FormGroup`, **por exemplo:**

```
formCadastro.get('senha');
```

Como isto é um objeto, não conseguimos ver seus detalhes no HTML, porém sabemos que necessitamos obter os detalhes dos erros deste controle, portanto vamos descobrir os métodos deste objeto que nos fornecem estes detalhes.

`hasError()`

O método `hasError` espera um parâmetro que é o nome do erro em *string*, e retorna se determinado erro está presente através de um valor booleano, permitindo fazer alguma lógica de exibição de elementos no template. **Por exemplo:**

```
{{ formCadastro.get('senha').hasError('required') }}  
{{ formCadastro.get('senha').hasError('pattern') }}  
{{ formCadastro.get('senha').hasError('minlength') }}
```

Observação: aqui o `minlength` é todo em minúsculo pois é acessado como uma propriedade.

`getError()`

O método `getError` espera um parâmetro que é o nome do erro em *string*, e retorna um objeto com detalhes do erro. **Por exemplo:**

```
{{formCadastro.get('senha').getError('required') | json }}  
{{formCadastro.get('senha').getError('pattern') | json }}  
{{formCadastro.get('senha').getError('minlength') | json}}
```

Abaixo como se apresentam os objetos retornados pelos erros `minlength` e `pattern`:

```
{ "requiredPattern": "^.(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!#$%& \"]).*$", "actualValue": "abc"  
}  
  
{ "requiredLength": 8, "actualLength": 4 }
```

Com estes detalhes conseguimos escrever mensagens de erro mais precisas para o usuário.

O pipe `json`

No *template*, podemos usar dentro das expressões de `bind` `{}{}` o carácter pipe `|` seguido da palavra `json`. Esta funcionalidade serve para transformar a apresentação dos dados no *template* de

maneira simples. No caso do pipe json, transformamos um objeto javascript em uma texto JSON para que possamos ver os detalhes deste objeto. **Por exemplo:**

```
 {{formCadastro.get('senha').getError('minlength') | json}}
```

Vai nos exibir no documento algo como:

```
{ "requiredLength": 8, "actualLength": 4 }
```

Observação: se dentro do objeto anotado pelo pipe json contiver uma estrutura circular (muito níveis de objetos, ou uma auto-referência dentro de uma propriedade) o pipe não funciona e uma mensagem de erro é apresentada do console do navegador.

23.3 CONDIÇÕES NO TEMPLATE COM *NGIF

Muitas vezes, a aplicação que estamos desenvolvendo precisa exibir um *template* ou parte de um *template* apenas em circunstâncias específicas.

A diretiva Angular `ngIf` deve ser usada em um elemento HTML. Ela **insere ou remove um elemento** baseado em uma condição verdadeira/ falsa. Sua notação começa com asterisco pois também é uma diretiva estrutural, ou seja, ela modifica os elementos HTML do *template*. **Por exemplo:**

```
<p *ngIf="formCadastro.get('senha').hasError('minlength')"  
    A senha deve conter pelo menos  
    {{formCadastro.get('senha').getError('minlength').requiredLength}}  
    caracteres  
</p>
```

Caso a expressão `formCadastro.get('senha').hasError('minlength')` seja verdadeira, no caso aqui, se o erro existir, então o elemento é inserido no HTML pelo Angular e o usuário poderá ver a mensagem de erro referente aquele problema do formulário. Caso a expressão retorne falso, então o elemento não estará presente do documento HTML.

O valor de `ngIf` aceita várias expressões de templates, onde podemos consultar com mais detalhes na documentação oficial [6].

Referências

1. <https://angular.io/api/forms/Validators>
2. <https://angular.io/api/forms/AbstractControl>
3. <https://angular.io/api/common/JsonPipe>
4. <https://angular.io/guide/displaying-data#conditional-display-with-ngif>
5. <https://angular.io/api/common/NgIf>
6. <https://angular.io/guide/template-syntax#template-expressions>

EXERCÍCIO: CADASTRO DE CONTAS - VALIDAÇÕES MÚLTIPLAS

24.1 OBJETIVO

Muito bom! Agora vamos usar mais validadores, inclusive, combiná-los!

The screenshot shows a user registration form with the following fields and validation messages:

- Name:** The input "Du" is shown with a red error message below it: "Você preencheu 2 caracteres de 3".
- Username:** The input "Usernam" is shown with a red error message below it: "Informar um nome de usuário é obrigatório!".
- Password:** The input "qwertuyuiop" is shown with a red error message below it: "Informar uma senha é obrigatório!".
- Phone:** The input "987654321" is shown with a red error message below it: "O telefone deve conter 8 ou 9 números."
- Avatar:** This field has no visible input or error message.

A red button labeled "CADASTRAR" is at the bottom left of the form area.

Figura 24.1: Mensagens de validação no formulário

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

24.2 PASSO A PASSO COM O CÓDIGO

1. No controle do campo `nome` passamos mais um item na lista de validadores, o **minLength**:

```
#./src/app/modules/cadastro/cadastro.component.ts
```

```
formCadastro = new FormGroup({
  nome: new FormControl('', [Validators.required, Validators.minLength(3)]),
  username: new FormControl('', [Validators.required]),
  senha: new FormControl('', [Validators.required]),
  avatar: new FormControl(),
})
```

2. No template podemos fazer uma mensagem para cada tipo de erro, e no momento que temos uma lista de validadores, teremos uma lista de erros. Então a primeira condição é verificar se **errors** existem neste controle, depois entramos na verificação dos erros individualmente e suas respectivas mensagens. Fique de olho na condição de cada ***ngIf**:

```
#./src/app/modules/cadastro/cadastro.component.html
```

```
<cmail-form-group [campo]="formCadastro.get('nome')">
<input formControlName="nome" autofocus required type="text" name="nome" cmailFormField>

<!-- se tiver erros, entao... -->
<div *ngIf="formCadastro.get('nome').errors">
  <!-- errors.required... -->
  <span class="mdl-textfield_error" *ngIf="formCadastro.get('nome').errors.required">
    Informar um nome é obrigatório!
  </span>

  <!-- errors minlength... -->
  <span class="mdl-textfield_error" *ngIf="formCadastro.get('nome').errors.minlength">
    Você preencheu {{ formCadastro.get('nome').errors.minlength.actualLength }} caracteres
    de {{ formCadastro.get('nome').errors.minlength.requiredLength }}
  </span>
```

```
</div>
</cmail-form-group>
```

3. Vamos adicionar mais um campo no formulário, o para o número de telefone, onde adicionaremos uma validação usando o *pattern* de uma **RegEx**:

Adicione o campo no template de **CadastroComponent**:

```
#./src/app/modules/cadastro/cadastro.component.html
```

```
<cmail-form-group [campo]="formCadastro.get('telefone')">
  <input formControlName="telefone" required type="text" name="telefone" cmailFormField>
  <div *ngIf="formCadastro.get('telefone').errors">
    <span class="mdl-textfield_error" *ngIf="formCadastro.get('telefone').errors.required">
      Informar um telefone é obrigatório!
    </span>
    <span class="mdl-textfield_error" *ngIf="formCadastro.get('telefone').errors.pattern">
      O telefone deve conter 8 ou 9 números.
      <!-- {{formCadastro.get('telefone').errors.pattern.requiredPattern}} -->
    </span>
  </div>
</cmail-form-group>
```

4. Crie mais um controle de formulário e use o `Validators.pattern` :

```
#./src/app/modules/cadastro/cadastro.component.ts
```

```
formCadastro = new FormGroup({
  nome: new FormControl('', [Validators.required, Validators.minLength(3)]),
  username: new FormControl('', [Validators.required]),
  senha: new FormControl('', [Validators.required]),
  telefone: new FormControl('', [Validators.required, Validators.pattern('[0-9]{4}-?[0-9]{4}[0-9]?')]),
  avatar: new FormControl(),
})
```

5. Desafio extra: que tal criar um componente para lidar com as mensagens de erro?

Pronto! agora temos validações reativas funcionando com regras e mensagens específicas para cada tipo de campo.

REQUISIÇÕES HTTP NO ANGULAR

Devido a atual arquitetura de sistemas utilizada por aplicações que trocam dados pela Internet, que permite um modelo mais amplo de clientes que consomem um único serviço de dados, o uso de operações HTTP feitas por clientes Web através do JavaScript se tornou muito popular, devido as possibilidades de fazer aplicações avançadas na Web com conteúdo dinâmico em páginas feitas com apenas HTML, CSS e JavaScript, assim proporcionando uma ótima experiência de uso.

Basicamente em toda linguagem de programação para Web é disponibilizada uma classe que constrói um objeto capaz de realizar requisições HTTP pelo código. No JavaScript, o ato de fazer requisições HTTP é popularmente chamado de AJAX (*Asynchronous Javascript and XML*), pois na época em que a classe `XMLHttpRequest` do JavaScript foi desenvolvida o formato de troca de dados existente era o XML, caracterizando o consumo de APIs no padrão SOAP.

Atualmente, por mais que o formato para troca de dados mais popular seja o JSON (JavaScript Object Notation) fornecido por APIs REST, algumas pessoas costumam ainda este antigo termo AJAX para se referir à ação de disparar uma requisição HTTP através do código JavaScript.

Hoje com JavaScript puro é ainda possível realizar requisições HTTP por `XMLHttpRequest` (XHR). Porém uma maneira mais simples e prática foi lançada na atualização de 2015 da linguagem, através da função `fetch()`, disponibilizada no escopo global.

25.1 HTTPCLIENT

No Angular, temos uma implementação mais robusta deste recurso nativo do JavaScript, onde algumas facilidades e melhorias foram realizadas para facilitar o dia a dia do desenvolvimento fornecido pela classe `HttpClient` do pacote `@angular/common/http`.

O `HttpClient` deve ser usado por injeção de dependência em uma classe, assim não sendo necessário entender os detalhes de como construir uma instância deste tipo. Porém para que o framework construa corretamente `HttpClient` é necessário importar `HttpClientModule` no módulo do componente que necessita fazer a requisição HTTP, pois este módulo que provê ao sistema de injeção de dependência do Angular todas as instruções para que a instância seja feita corretamente.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

25.2 MÉTODOS DE HTTPCLIENT

A API `HttpClient` contém vários métodos para solicitar recursos externos, como também para enviar. Ao usarmos o *autocomplete* do Visual Studio Code, percebemos que estes métodos da classe tem praticamente os mesmos nomes dos métodos do protocolo HTTP, como GET, POST, DELETE etc, e mais alguns outros que nos auxiliam em cenários específicos.

Praticamente todos os métodos de `HttpClient` tem sobrecargas (*overload*), exceto o método `jsonp`, assim cobrindo um vasto de cenário de operações HTTP. Portanto o retorno padrão dos métodos de `HttpClient` é uma `Observable<Object>`, podendo ser modificado de acordo com a sobrecarga ou implementação do método.

```
constructor(private httpClient: HttpClient) {  
    this.httpClient.  
}  
    delete  
    get  
    head  
    jsonp  
    options  
    patch  
    post  
    put  
    request
```

Figura 25.1: Métodos de `HttpClient`

25.3 CRIANDO UMA REQUISIÇÃO HTTP COM ANGULAR: HTTPCLIENT

Após a importação de `HttpClientModule` e a injeção de `HttpClient`, podemos realizar uma requisição (ou *request*) HTTP.

Para realizar uma requisição para obter recursos, usamos o método `get` que tem apenas um parâmetro obrigatório, que é a url que localiza o recurso, por exemplo o endereço de uma API, o segundo parâmetro é opcional (denotado pelo "*elvis operator*"), e pode ser informado através de um objeto para adicionarmos configurações como Headers do HTTP etc.

O método `get` recebe como primeiro parâmetro (e de preenchimento obrigatório) uma *string* com o endereço da API que desejamos consumir. Este método faz a requisição HTTP, captura a resposta da API, e interpreta o corpo da resposta HTTP, o `body`, como um JSON.

O retorno de um método no TypeScript é caracterizado pelo : (dois pontos) após os parâmetros do método. Podemos ver na *tooltip* que o retorno do `get` é uma `Observable<Object>`. Ou seja, por fim o `get` retorna o `body` da resposta HTTP da API, como um `Object` num formato de **fluxo**, através do tipo `Observable<Object>`.

```
constructor(private httpClient: HttpClient) {}

ngOnInit(){
  this.httpClient
    .get('https://api.cmail.com.br/users')
}
```

Desta forma a requisição ainda não é disparada. É necessário utilizarmos o método `subscribe` de `Observable<T>` - que é o tipo do retorno do método `get` - para que a requisição seja realizada pelo Angular.

25.4 DISPARANDO A REQUISIÇÃO HTTP

No método `subscribe` opcionalmente podemos trabalhar com as respostas de retorno da requisição, que pode ser dados ou um erro HTTP. No primeiro parâmetro de `subscribe`, construímos uma função de *callback* para acessar a resposta da requisição caso ela tenha sido efetuada com sucesso. Este *callback* tem disponível em seu parâmetro o objeto que está dentro da `Observable<T>` retornada pelo método de `HttpClient`. Por padrão o objeto é o *response body* da requisição, ou seja, os dados já transformados de JSON para um objeto JavaScript.

```
constructor(private httpClient: HttpClient) {}

ngOnInit(){
  this.httpClient
    .get('https://api.cmail.com.br/users')
    .subscribe(
      function (response) {
        console.log(response);
      }
)
```

```
}
```

Arrow functions e escopo léxico

Veja que você consegue enxergar uma flecha (no inglês, *arrow*) abaixo:

```
=>
```

Agora que você já sabe porquê este nome *Arrow Functions*, vamos entender melhor como elas funcionam.

Uma **arrow function** é uma **função anônima** que possui uma sintaxe mais curta, quando comparada com a **function expression**. Porém, o seu diferencial não é apenas a sintaxe enxuta: **toda arrow function compartilha o mesmo this léxico de seu escopo pai**.

Qual é o `this` do método `constructor`? A instância da sua classe. Usando a *arrow function*, qual o `this` de `subscribe`? O de `constructor`. Sendo assim, o `this` de `subscribe` será o da instância da classe, ou seja, o mesmo `this` do método que usa `subscribe`.

```
listaUsers = [];

constructor(private httpClient: HttpClient) {}

ngOnInit(){
  this.httpClient
    .get('https://api.cmail.com.br/users')
    .subscribe(
      (response) => {
        console.log(response);
        this.listaUsers = response;
      }
    )
}
```

Antes com *function expression*, não seria possível acessar `this.listaUsers`, pois o `this` seria o método que invocou a função de *callback* e não classe.

É possível enxugar ainda mais o código, omitindo o bloco da *arrow function* e os parênteses (possível apenas se hover um único parâmetro), e deixar tudo em apenas 1 linha:

```
listaUsers = [];

constructor(private httpClient: HttpClient) {}

ngOnInit(){
  this.httpClient
    .get('https://api.cmail.com.br/users')
    .subscribe(
      response => this.listaUsers = response
    )
}
```

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

25.5 TRABALHANDO ERROS DE HTTP

Caso ocorra um erro na requisição HTTP, ou seja retornado um erro pelo servidor, o segundo parâmetro de `subscribe` é executado, e para tratar este cenário, também passamos um `callback` que terá em seu parâmetro um objeto do tipo `HttpErrorResponse` [4] com os detalhes do erro.

```
listaUsers = [];

constructor(private httpClient: HttpClient) {}

ngOnInit(){
  this.httpClient
    .get('https://api.cmail.com.br/users')
    .subscribe(
      response => this.listaUsers = response
      ,(erro: HttpErrorResponse) => console.error(erro)
    )
}
```

O terceiro parâmetro de `subscribe` é um `callback` que representa o "*complete*", ou seja, quando a resposta já foi trabalhada com sucesso, e mais alguma outra tarefa precisa ser executada. Esta não recebe nenhum objeto como parâmetro.

```
listaUsers = [];

constructor(private httpClient: HttpClient) {}

ngOnInit(){
  this.httpClient
    .get('https://api.cmail.com.br/users')
    .subscribe(
      response => this.listaUsers = response
    )
}
```

```

        ,(erro: HttpErrorResponse) => console.error(erro)
        ,() => console.log('complete!')
    )
}

```

25.6 OPÇÕES PARA AS REQUISIÇÕES

Uma das opções do último parâmetro dos métodos de `HttpClient` é o `observe`. Por padrão seu valor é `'body'`, ou seja, a resposta da requisição HTTP é o `body`, porém caso eu queira ter acesso a alguma informação no `Headers` da resposta, posso alterar o valor para `'response'`, e assim vamos ter um objeto do tipo `HttpResponseBase` [5] com detalhes da resposta retornada.

```

this.httpClient
.get(
  'https://api.cmail.com.br/users'
  , { observe: 'response' }
)
.subscribe(
  (response: HttpResponseBase) => {
    console.log(response);
  }
  ,(erro: HttpErrorResponse) => {
    console.error(erro);
  }
)

```

É possível trabalhar com os detalhes do HTTP `Headers` e `body` construindo uma interface para o tipo do retorno e iterando sob o objeto do tipo para obter os dados, conforme os detalhes disponível no guia [6].

Requisição do tipo `head`

O método HEAD do HTTP solicita uma resposta de forma idêntica ao método GET, porém sem conter o corpo da resposta. Portanto, quando você só precisa de acesso aos detalhes da requisição e resposta e não ao conteúdo, usamos o método `head`, que é tem um custo menor de transferência de dados. Como por exemplo para verificar se um recuso na Web está disponível:

```

validaImagem(campo: FormControl) {
  return this
    .httpClient
    .head(
      campo.value
      ,{ observe: 'response'}
    )
}

```

25.7 UM POUCO DE RXJS

A biblioteca Reactive X fornece uma API para programação assíncrona através de *observable streams* (fluxos observáveis). Esta biblioteca é disponibilizada para as linguagens mais populares de

mercado, e sua versão pra JavaScript é a ***Reactive Extensions for JavaScript (RxJS)***.

A RxJS é formada por um conjunto de bibliotecas para compor programas assíncronos e baseados em eventos, usando coleções observáveis. Na prática, interagimos com um fluxo observável (***observable stream***) da RxJS através dos seus métodos, permitindo modificar o seu conteúdo de forma assíncrona através dos operadores (***operators***) desta biblioteca.

Este **fluxo**, ou também conhecido como *stream* é proporcionado pelo uso do tipo `Observable<T>` , que proporciona métodos que abrem um canal de transformados dos dados que passarão por ali.

Referências

1. <https://angular.io/guide/http>
2. <https://angular.io/api/common/http/HttpClient>
3. <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods>
4. <https://angular.io/api/common/http/HttpErrorResponse>
5. <https://angular.io/api/common/http/HttpResponseBase>
6. <https://angular.io/guide/http#reading-the-full-response>
7. <http://reactivex.io/>

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

VALIDAÇÕES ASSÍNCRONAS

Um dos grandes diferenciais dos formulários reativos do Angular é poder criar funções de validação customizadas, possibilitando a implementação de regras específicas do domínio da aplicação, porém mais que isso, é possível também que estas funções consultem de serviços externos para que com isto seja feita uma lógica de validação.

Em um formulário de cadastro, onde a pessoa precisa criar um nome de usuário, para garantir que os nomes sejam únicos, é possível antecipar a submissão do formulário verificando na API se o nome de usuário preenchido já foi utilizado ou não no momento em que o campo é preenchido.

As validações assíncronas possibilitam verificações que dependem de APIs de terceiros, através de requisições HTTP, como uma consulta do código postal informado, ou autorização de um exame pelo convênio de saúde.

26.1 VALIDADORES ASSÍNCRONOS

A classe `Validators` não tem nenhuma função pronta deste tipo, pois entende-se que uma validação assíncrona terá uma regra de negócio muito específica. Portanto é necessária implementar a sua própria, que receberá como parâmetro o objeto do controle, e esta deverá retornar uma `Observable<T>` ou `Promise<T>`.

Construindo um validador assíncrono

A maneira mais simples de fazer uma função de validação assíncrona é criando um método na própria classe do componente. Para fazer ela funcionar minimamente, declaramos o seu parâmetro o objeto do controle que tem o tipo `AbstractControl`, e esta deverá retornar uma `Observable`. Aqui usamos como exemplo o método `from` da RxJS que cria uma `Observable` a partir do parâmetro passado. Criamos uma `Observable` com um objeto com detalhes do erro que poderá nos ajudar no feedback para o usuário.

```
validaCEP(control: AbstractControl){  
  return from([{cepInvalido: true}]);  
}
```

Utilizando o validador assíncrono

Uma função de validação assíncrona é o terceiro parâmetro opcional na construção de um `FormControl`, e é disparada logo após a validação síncrona apenas se esta for executada sem erros. É possível informar apenas uma função, ou uma lista ou uma composição de validadores assíncronos.

No exemplo abaixo colocamos `validaCEP` para o controle `cep`:

```
formEndereco = new FormGroup({
  cep: new FormControl('', [
    Validators.required,
    Validators.minLength(8),
    Validators.maxLength(8),
    Validators.pattern('[0-9]{8}')
  ],
  this.validaCEP.bind(this)
),
logradouro: new FormControl({value: '', disabled: true}),
complemento: new FormControl({value: '', disabled: true}),
bairro: new FormControl({value: '', disabled: true}),
localidade: new FormControl({value: '', disabled: true}),
uf: new FormControl({value: '', disabled: true})
})
```

Aqui temos que fazer o `.bind(this)` no método para que o contexto de execução da função seja o correto, no caso o componente em questão, caso contrário, `this` ficará `undefined` impedindo de acessarmos propriedades e métodos da classe do componente. Isto teve que acontecer pois estamos criando o método de validação no corpo da classe do componente. Pois se o método de validação tiver sua própria classe implementando a interface `AsyncValidator` isto não é necessário, mais detalhes na referência [3] deste capítulo.

Com isto podemos implementar a lógica da função assíncrona do método, usando o `HttpClient` para fazer uma requisição apenas do headers do HTTP, e com base na `response` da requisição, decidimos se estará válido ou não o valor informado no campo CEP.

Para que a requisição seja realizada é necessário colocar o `return` de `validaCEP` diretamente chamada de `httpClient`. Usamos a API do site ViaCep, onde fazemos uma interpolação do endereço que vamos fazer a `request` com o valor de `control`, ou seja, o valor digitado no campo CEP.

```
validaCEP(control: AbstractControl){
  return this
    .httpClient
    .get(`https://viacep.com.br/ws/\${control.value}/json/`\)
```

Caso seja necessário modificar ou apenas fazer uma verificação do conteúdo (retornado na resposta HTTP, e antes de finalizar a operação com o `subscribe()`, usamos o método `pipe()` também do tipo

```

Observable<T> .

validaCEP(control: AbstractControl){
  return this
    .httpClient
    .get(`https://viacep.com.br/ws/${control.value}/json/`)
    .pipe()
}

}

```

"Pipeable" Operators da RxJS

O método `pipe()` abre um canal para acessarmos o conteúdo dentro da `Observable<T>`, permitindo você compor lógicas complexas com este dado retornado. Para entrar neste conteúdo e trabalharmos nas modificações dos dados, usamos uma série de "**operadores encadeados**" oriundos da biblioteca RxJS. Como o retorno do método `get` de `httpClient` é uma `Observable<T>`, usamos o método `pipe()` em conjunto com **operadores**.

```
import { map, catchError } from "rxjs/operators";
```

Os operadores são funções puras (*pure functions*) disponibilizadas na biblioteca, portanto são importadas ao código sob demanda. As funções puras favorecem o empacotamento do projeto (*build*), pois utiliza o *tree shaking* ("sacudir a árvore"), técnica que elimina do empacotamento as dependências que foram importadas no projeto porém que não estão sendo realmente utilizadas. Assim reduzindo de forma significativa o tamanho do pacote da aplicação entregue à produção.

O operador map

Caso a requisição tenha sido feita com sucesso (caracterizada pelo intervalo de números 200 do HTTP status), usamos o operador `map` para entrar dentro do objeto de `response` retornado pelo `get`.

O `map` recebe como parâmetro uma função, que contém disponível em seu parâmetro o objeto de `response` do HTTP na forma de um objeto.

```

validaCEP(control: AbstractControl){
  return this
    .httpClient
    .get(`https://viacep.com.br/ws/${control.value}/json/`)
    .pipe(
      map((response: any) =>{
        console.log(response)
        return null
      })
    )
}

```

Para esta API em específico, verificamos se o objeto retornado tem uma propriedade `erro`, pois nela, mesmo que o número do CEP não exista, é feita a requisição e é retornado um objeto com uma

única propriedade `erro`. Caso esta propriedade `erro` esteja presente, retornamos um objeto com os detalhes do erro, portanto criamos um objeto com a propriedade `cepInvalido`, que armazena um valor booleano. E caso o objeto `response` não tenha a propriedade `erro` é sinal que o CEP está válido e contém as informações completas do endereço do código postal, portanto deveremos retornar um `null`, ou seja, esta requisição não contém erros de validação.

```
validaCEP(control: AbstractControl){  
  
    return this  
        .httpClient  
        .get(`https://viacep.com.br/ws/${control.value}/json/`)  
        .pipe(  
            map((response: any) =>{  
                if(response.erro) {  
                    return {cepInvalido: response.erro}  
                }  
                return null  
            })  
        )  
    }  
}
```

O operador catchError

O método `pipe` da RxJS permite usarmos vários operadores para trabalhar com a `Observable<T>`, cada operador deverá ser um parâmetro de `pipe`. Para caso a requisição HTTP falhe, podemos tratar o objeto de erro com o operador `catchError`, que também recebe uma função como parâmetro e que tem disponível o objeto com os erros do HTTP Headers e que tem o tipo `HttpErrorResponse`.

```
validaCEP(control: AbstractControl){  
  
    return this  
        .httpClient  
        .get(`https://viacep.com.br/ws/${control.value}/json/`)  
        .pipe(  
            map((response: any) =>{  
                if(response.erro) {  
                    return {cepInvalido: response.erro}  
                }  
                return null  
            })  
            ,catchError(  
                (response: HttpErrorResponse) => [{cepInvalido: !response.ok}]  
            )  
        )  
    }  
}
```

Com isto podemos pegar detalhes deste erro para forma o objeto de erro e validação que a nossa função de validação precisa retornar. Um ponto de atenção é que o `catchError` precisa retornar o objeto dentro de um `array`. E para ficar coerente continuamos retornando um objeto com a propriedade `cepInvalido` que armazena um valor booleano, para usarmos na lógica de feedback para o usuário.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

26.2 FEEDBACK DE ERROS

Quando uma validação assíncrona começa, o controle entra no estado pendente (*PENDING*), no qual consultamos pela propriedade `status`. Esta propriedade pode ser utilizada para um feedback visual enquanto a validação está sendo realizada. Um exemplo comum para esta indicação visual é usar um *loader/spinner* que pode ser exibido de acordo com a condição do `status`.

Veja os exemplos na tag `span` e `button`:

```
<!-- código anterior omitido -->
<label for="cep">CEP: </label>
<input formControlName="cep" placeholder="12345678" id="cep" type="search" required>

<!-- Feedback para validação assíncrona -->
<span *ngIf="formEndereco.get('cep').status === 'PENDING'" class="loader"></span>

<button [ngClass]="{{'valid':formEndereco.get('cep').status === 'VALID'}}">
  Consultar
</button>
<!-- código posterior omitido -->
```

E para utilizarmos o objeto que criamos com detalhes do erro da validação assíncrona, podemos fazer da seguinte maneira:

```
<span *ngIf="formEndereco.get('cep').hasError('cepInvalido')" class="erro">
  CEP {{formEndereco.get('cep').value}} inválido
</span>
```

O código completo e funcionando deste exemplo está disponível nas referências do capítulo [5].

26.3 CONTROLANDO O DISPARO DAS REQUISIÇÕES

Por padrão toda a vez que o validador síncrono está sem erros e associado com o evento `change`, a

validação assíncrona é disparada. Para melhorar a performance da aplicação podemos mudar a estratégia de disparo da validação para apenas quando o usuário saí do campo (o que sinaliza que ela já terminou de preencher).

No construtor de `FormGroup` [6], em seu segundo parâmetro passamos um objeto do tipo `AbstractControlOptions` [7], onde através da propriedade `updateOn` podemos mudar o padrão '`change`' para o valor '`blur`', caracterizando o evento que desfoco do campo. É possível também mudar para '`submit`'. **Por exemplo:**

```
formEndereco = new FormGroup({
  cep: new FormControl('',
    [
      Validators.required
      ,Validators.minLength(8)
      ,Validators.maxLength(8)
      ,Validators.pattern('[0-9]{8}')
    ]
    ,this.validaCEP.bind(this)
  )
  ,{updateOn: 'blur'}
})
```

Referências

1. <https://angular.io/guide/form-validation#custom-validators>
2. <https://angular.io/guide/form-validation#async-validation>
3. <https://angular.io/guide/form-validation#implementing-custom-async-validator>
4. <https://viacep.com.br/>
5. <https://stackblitz.com/edit/postalcode-validation>
6. [https://angular.io/api/forms/FormGroup#constructor\(\)](https://angular.io/api/forms/FormGroup#constructor)
7. <https://angular.io/api/forms/AbstractControlOptions>

EXERCÍCIO: CADASTRO DE CONTAS - VALIDAÇÕES ASSÍNCRONAS E REQUISIÇÕES HTTP

27.1 OBJETIVO

No campo avatar vamos esperar que seja informado a URL de uma imagem já hospedada em outro servidor, para facilitar a manutenção da própria atualização dessa imagem pelo usuário e também para não termos que lidar com a hospedagem de arquivos no servidor da nossa aplicação. Porém como garantir que o endereço da imagem informada está válido e disponível? Vamos ter que fazer uma validação deste endereço informado, ou seja, uma validação assíncrona.

Título da página atual

Nome
Bob

Username
bobesponja

Senha

Telefone
123456789

Avatar
http://www.naoexiste.erro

Erro na URL informada :(

CADASTRAR

Figura 27.1: Validação assíncrona no campo avatar

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

27.2 PASSO A PASSO COM CÓDIGO

1. Vamos criar nosso próprio validador, e como ele fará uma requisição para verificar a URL, será um **validador assíncrono**, para isto basta ter um método da classe que recebe o campo como parâmetro e retorna uma *promise* ou *observable*, vamos chama-lo de **validaImagem**. Adicionaremos este método como terceiro parâmetro no campo `avatar` na propriedade `formCadastro` de **CadastroComponent**:

`#./src/app/modules/cadastro/cadastro.component.ts`

```
formCadastro = new FormGroup({
  nome: new FormControl('', [Validators.required, Validators.minLength(3)]),
  username: new FormControl('', [Validators.required]),
  senha: new FormControl('', [Validators.required]),
  telefone: new FormControl('', [Validators.required, Validators.pattern('[0-9]{4}-?[0-9]{4}[0-9]?'}}},
  avatar: new FormControl('',[Validators.required], this.validaImagem.bind(this))
})
```

Aqui o método `bind()` está passando o campo como parâmetro através do `this` do contexto de execução deste código.

2. Crie o método no corpo da classe **CadastroComponent** apenas com requisitos básicos, logo mais vamos implementá-la com a regra que precisamos:

`#./src/app/modules/cadastro/cadastro.component.ts`

```
validaImagem(campoDoFormulario: FormControl){
  return new Promise(resolve => resolve());
}
```

3. Para verificar se a imagem está disponível, temos que fazer uma requisição para o endereço informado. Para isto usaremos o **HttpClient**, biblioteca do Angular responsável para este tipo de tarefa.

Como construir uma instância de **HttpClient** é muito difícil, vamos ter injetá-la na classe, onde teremos acesso à uma instância global fornecida por seu módulo.

Pacote de referência:

```
#./src/app/modules/cadastro/cadastro.component.ts
```

```
import { HttpClient } from '@angular/common/http';
```

Injeção de HttpClient:

```
#./src/app/modules/cadastro/cadastro.component.ts
```

```
constructor(private httpClient: HttpClient){}
```

4. A instância pronta de **HttpClient** para injeção é provida por **HttpClientModule**, portanto devemos importá-lo para que o código anterior funcione. Caso contrário vamos ter o seguinte erro no console do navegador: [No provider for HttpClient!]

```
#./src/app/app.module.ts:26
```

```
imports: [
  BrowserModule,
  FormsModule,
  ModuloRoteamento,
  ReactiveFormsModule,
  HttpClientModule // Importamos HttpClientModule!
],
```

5. Agora podemos trabalhar na lógica do método `validaImagem` usando o **HttpClient** - no lugar da Promisse - para fazer a requisição à imagem informada no campo `avatar`:

```
#./src/app/modules/cadastro/cadastro.component.ts
```

```
validaImagem(campoDoFormulario: FormControl) {
  return this.httpClient
    .head(campoDoFormulario.value, {
      observe: 'response'
    })
    .pipe(
      map((response: HttpResponseBase) => {
        return response.ok ? null : { urlInvalida: true }
      }),
      catchError((error) => {
        return [{ urlInvalida: true }]
      })
    )
}
```

Os métodos `map` e `catchError` são específicos da biblioteca **RxJS**, utilizada no **HttpClient** para retornar **observables streams** (fluxos observáveis), que é uma nova abordagem para trabalhar com **programação assíncrona**. Para utilizá-los precisamos importar suas referências:

```
import { map, catchError } from "rxjs/operators";
```

6. Por último, adicionamos no template de **CadastroComponent** as mensagens para validação do campo avatar:

```
#./src/app/modules/cadastro/cadastro.component.html
```

```
<cmail-form-group [campo]="formCadastro.get('avatar')">
  <input formControlName="avatar" type="text" name="avatar" cmailFormField>
  <span class="mdl-textfield_error" *ngIf="formCadastro.get('avatar').hasError('required')">
    Informar uma url com um avatar é obrigatório!
  </span>
  <span class="mdl-textfield_error mdl-textfield_checking" *ngIf="formCadastro.get('avatar').status === 'PENDING'">
    Validação pendente....
  </span>
  <span class="mdl-textfield_error mdl-textfield_valid" *ngIf="formCadastro.get('avatar').status === 'VALID'">
    URL Válida ☺
  </span>
  <span class="mdl-textfield_error" *ngIf="formCadastro.get('avatar').hasError('urlInvalida')">
    Erro na URL informada ☹
  </span>
</cmail-form-group>
```

Observação para as classes de CSS **mdl-textfield_valid** e **mdl-textfield_checking** nas mensagens de URL válida e pendente, caso contrário elas não serão exibidas.

ENVIADO DADOS DO CADASTRO PARA A API

28.1 PERSISTÊNCIA DE DADOS COM JS E A NECESSIDADE DE UMA API BACKEND

Com JavaScript não podemos nos conectar diretamente em uma base de dados segura e persistente, como um Postgre, Mysql, Oracle, MongoDB etc. Apenas conseguimos acessar as APIs de armazenamento local que o navegador fornece, como LocalStorage[1] ou IndexDB [2], no qual têm uma persistência temporária.

Porém através do JavaScript conseguimos fazer requisições HTTP para aplicações backend que acessam um banco de dados, fazendo um intermediário para o frontend e a camada de dados, estas são as APIs - *Application Programming Interface*.

Portanto é necessário o suporte de uma aplicação backend para uma aplicação Angular (frontend) apresentar e manipular dados que persistem, além de outras tarefas, como gerar um token de autenticação etc.

Para o Angular, não importa em qual linguagem o backend é desenvolvido, que pode ser em Java, Python, Ruby, Node, C# etc, o importante é que seja possível fazer requisições dos dados num formato fácil de integrar, como JSON ou até mesmo o antigo XML.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

28.2 CMAIL BACK API

Para o CMail, criamos uma aplicação com NodeJS e um banco de dados de documentos, o SQLite3 [3], onde é possível obter tudo que precisamos para a aplicação num único pacote.

O backend do CMail tem seu código fonte e documentação disponível no Github pelo endereço: <https://github.com/caelum/cmail-back>.

Faça uma cópia do código fonte em seu computador, na pasta do `cmail-back` instale as dependências do projeto com o comando `npm install`, e depois inicie a aplicação com `npm start`.

Um servidor web local será disponibilizado no endereço `http://localhost:3200` em seu computador.

28.3 ENVIANDO DADOS PELO HTTP

Com JavaScript coletamos e armazenamos em uma variável os dados de um formulário no HTML. O formato em que os dados devem ser enviados depende de como a API espera receber estes dados (JSON ou XML). O método POST do HTTP possibilita o envio de um `body`, ou seja, o corpo de uma requisição, e nesta estrutura que colocamos os dados coletados do formulário. Lembrando que no HTTP só é possível trafegar texto, portanto estes dados devem estar no formato em que API espera e em texto (*string*).

Para realizar esta operação com Angular, o `HttpClient` contém um método `post`. Este método tem dois parâmetros obrigatórios, a `url` e o `body`, e o terceiro opcional onde podemos configurar os `headers` do HTTP.

```
constructor(private http: HttpClient){}

handleCadastro() {
  this.http
    .post('http://localhost:3200/users', this.formCadastro.value)
}
```

O método `post` retorna uma `Observable<Object>`, portanto para que a requisição realmente seja disparada é necessário se inscrever no retorno de `post` com o método `subscribe`, que possibilita também realizarmos funções caso a resposta HTTP seja positiva ou com erros:

```
constructor(private http: HttpClient){}

handleCadastro() {
  this.http
    .post('http://localhost:3200/users', this.formCadastro.value)
    .subscribe(
      () => {
        console.log(`Cadastrado com sucesso`);
        this.formCadastro.reset()
    }
}
```

```

        ,erro => console.error(erro)
    )
}

```

28.4 DATA TRANSFER OBJECT: CONCEITO E CRIAÇÃO;

Algumas APIs foram desenvolvidas em um modelo de dados diferente do modelo da aplicação frontend. No caso do CMail, os objetos dos campos do formulário de cadastro foram registrados em português, porém a API foi desenvolvida em inglês e espera que os dados sejam enviados em inglês, caso contrário irá responder a requisição com um erro dizendo que alguns dados estão em falta.

Para resolver este problema de "tradução" de dados de uma aplicação para outra, podemos utilizar um objeto de transferência de dados (DTO) [4], que é um padrão de projeto de software amplamente utilizado. Com DTOs é possível criar um intermediador que conhece tanto o modelo de dados frontend quanto o do backend.

Existem algumas maneiras de criar um DTO, a mais simples através de um objeto literal que tem os valores de suas propriedades preenchidas com os valores das propriedades do objeto de origem, assim transferindo os dados de um objeto com o modelo de dados em português, para um objeto com o modelo de dados em inglês:

```

cadastroDTO = {
  name: this.formCadastro.get('nome').value,
  username: this.formCadastro.get('usuario').value,
  password: this.formCadastro.get('senha').value,
  phone: this.formCadastro.get('telefone').value
}

```

Ou de maneira mais elaborada e reaproveitável, criando uma classe com TypeScript, com propriedades em inglês, e que poderá receber em sua construção um objeto desestruturado com as propriedades em português, assim retornando uma instância de um modelo em inglês com base em um modelo em português:

```

export class UserInputDTO {
  name = '';
  username = '';
  password = '';
  phone = '';

  constructor({nome, usuario, senha, telefone}) {
    this.name = nome;
    this.username = usuario;
    this.password = senha;
    this.phone = telefone;
  }
}

```

No Angular, as classes de modelos DTOs podem ficar armazenadas em `src/app/models/dto` e o nome do arquivo pode ser `user-input.ts`.

Existem DTOs para **input** e para **output**, ou seja, o para **input** fará a transferência de dados da aplicação atual (origem) para o idioma ou modelo da aplicação de destino. E o DTO para **output** fará a transferência de dados da aplicação externa para a aplicação atual (origem).

Usando a classe DTO

```
constructor(private http: HttpClient){}

handleCadastro() {

  const userDTO = new UserInputDTO(this.formCadastro.value);

  this.http
    .post('http://localhost:3200/users', userDTO)
    .subscribe(
      () => {
        console.log(`Cadastrado com sucesso`);
        this.formCadastro.reset()
      }
      ,erro => console.error(erro)
    )
}
```

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

28.5 REDIRECIONANDO ROTAS

Após realizar o cadastro com sucesso, é possível fazer um redirecionamento para a página de login. O roteamento do Angular é o responsável por ter todas as informações das rotas e também métodos para acessar elas programaticamente. Para isto, trazemos para a atual instância de `Router`, já disponível pelo `RouterModule` quando o mesmo foi importando em `AppModule`, através da injeção de dependência, assim possibilitando o uso dos métodos do roteamento.

```
constructor(private http: HttpClient
            ,private router: Router){}

handleCadastro() {
```

```
const userDTO = new UserInputDTO(this.formCadastro.value);

this.http
  .post('http://localhost:3200/users', userDTO)
  .subscribe(
    () => this.router.navigate(['login'])
    , erro => console.error(erro)
  )
}
```

Método `navigate`

O método `navigate` [5] de `Router` espera receber um *array* como parâmetro, assim como o `routerLink`, onde cada posição do *array* é um fragmento da rota, que pode conter *strings* ou variáveis que retornam *string*, possibilitando a composição de uma rota bem detalhada.

Referências

1. <https://developer.mozilla.org/pt-BR/docs/Web/API/Window/Window.localStorage>
2. <https://developer.mozilla.org/pt-BR/docs/Web/API/WindowOrWorkerGlobalScope/indexedDB>
3. <https://www.npmjs.com/package/sqlite3>
4. https://pt.wikipedia.org/wiki/Objeto_de_Transfer%C3%A3ncia_de_Dados
5. <https://angular.io/guide/router#navigating-back-to-the-list-component>

EXERCÍCIO: ENVIANDO DADOS DO FORMULÁRIO COM HTTPCLIENT

29.1 OBJETIVO

Agora que o formulário está completo, basta coletarmos os dados e usarmos o `HttpClient` que postará os dados em um *backend* (servidor web, ou API), possibilitando que estes dados sejam persistentes. Para isto, vamos subir uma API local que será a responsável por receber estas requisições por parte do cliente, neste caso o CMail.

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

29.2 PASSO A PASSO COM CÓDIGO

- Vamos copiar do Github da Caelum o repositório da API, disponível no endereço:
<https://github.com/caelum/cmail-back>

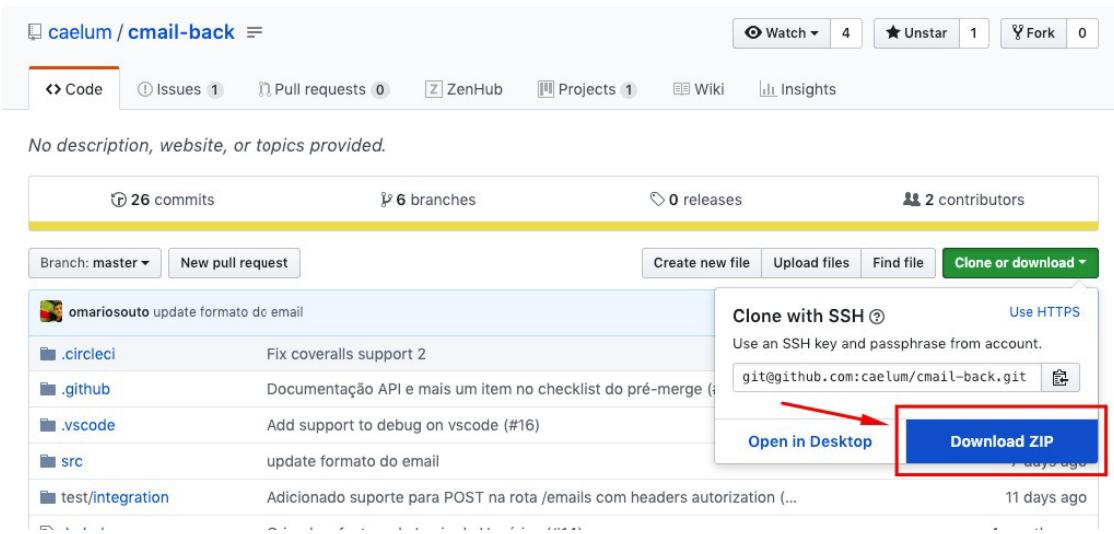


Figura 29.1: Download no Github

Após o download, extraia o .zip e coloque em um diretório de fácil acesso, como a área de trabalho ou a pasta do usuário do computador.

2. A API foi construída em **Nodejs** portanto devemos acessar sua pasta pelo terminal e instalar as dependências com o comando `npm install`:

```
cd cmail-back-master
npm install
```

3. Para iniciar o servidor: `npm start`:

```
> cross-env NODE_ENV='production' nodemon ./server.js --exec babel-node
[nodemon] 1.18.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: ***!
[nodemon] starting `babel-node ./server.js`
consign v0.1.6 Initialized in /Users/Vanessa/Desktop/cmail-back-master
+ ./src/routes/emails.js
+ ./src/routes/login.js
+ ./src/routes/users.js
Servidor subiu na porta 3200
Acesse em: http://localhost:3200
Para derrubar use CTRL + C
```

Figura 29.2: API para o CMail iniciada

Agora já podemos voltar ao projeto Angular para terminar o envio dos dados!

4. Na classe de **CadastroComponent** no método `handleCadastrarUsuario`, terminamos de implementar o envio de dados. Com o `HttpClient` usamos o método `post` que recebe a URL da API e em seguida o objeto com os dados do formulário:

```
#./src/app/modules/cadastro/cadastro.component.ts

handleCadastrarUsuario() {
  if (this.formCadastro.valid) {

    const userData = new User(this.formCadastro.value);

    this.httpClient
      .post('http://localhost:3200/users', userData)
      .subscribe(
        () => {
          console.log(`Cadastrado com sucesso`);
          this.formCadastro.reset()
        }
      ,erro => console.error(erro)
    )
  }
  else {
    this.validarTodosOsCamposDoFormulario(this.formCadastro);
  }
}
```

Perceba que criamos uma classe **User** para podemos fazer o modelo esperado pelo backend, ou seja, com os atributos do objeto em inglês.

- Vamos criar uma classe **User** em um novo diretório `src/app/models/user.ts`. Esta classe receberá um objeto, que será o formulário, e vamos pegar seus atributos e deixá-lo como a API precisa receber:

```
#./src/app/models/user.ts
```

```
export class User {

  name = '';
  username = '';
  password = '';
  phone = '';
  avatar = '';

  constructor({nome, username, senha, telefone, avatar}){
    this.name = nome;
    this.username = username;
    this.password = senha;
    this.phone = telefone;
    this.avatar = avatar;
  }
}
```

Pronto! Agora temos nosso cadastro funcionando... Porém o que fazer após os dados serem enviados?

- Vamos redirecionar o usuário para a tela de login após o cadastro. Para isto vamos chamar o **Router**, pois ele é o que tem os poderes para nos enviar até as rotas. Para usá-lo temos que injetar sua instância no construtor:

```
#./src/app/modules/cadastro/cadastro.component.ts
```

```
constructor(private httpClient: HttpClient  
,private roteador: Router){}
```

7. Por último podemos chamar seu método **navigate**, após o sucesso de cadastro do usuário, que pode nos levar até a rota desejada:

```
#./src/app/modules/cadastro/cadastro.component.ts
```

```
//código anterior omitido  
this.httpClient  
.post('http://localhost:3200/users',userData)  
.subscribe(  
() => {  
  console.log(`Cadastrado com sucesso`);  
  this.formCadastro.reset()  
  
  //após 1 segundo, redireciona para a rota de login  
  setTimeout(() => {  
    this.roteador.navigate(['']);  
  }, 1000);  
}  
,erro => console.error(erro)  
)  
//código posterior omitido
```

8. **Extra:** e se der erro na API? Vamos tratar o erro e mostrar as mensagens na tela para o usuário no callback de erro do método `subscribe`.

Declare uma propriedade na classe **CadastroComponent** chamada **mensagensErro**.

9. No callback de erro da API, preencha a propriedade **mensagensErro**:

```
#./src/app/modules/cadastro/cadastro.component.ts
```

```
this.httpClient  
.post('http://localhost:3200/users',userData)  
.subscribe(  
  (response) => {  
    console.log(`Cadastrado com sucesso`);  
    this.formCadastro.reset()  
  
    setTimeout(() => {  
      this.roteador.navigate(['']);  
    }, 1000);  
  }  
,(responseError: HttpErrorResponse) => {  
  //resposta caso existam erros!  
  this.mensagensErro = responseError.error.body  
})
```

10. No template, faça uma verificação se o erro existe, caso positivo exiba as mensagens:

```
#./src/app/modules/cadastro/cadastro.component.html
```

```
<ul *ngIf="mensagensErro" class="mdl-textfield is-invalid">  
  <li *ngFor="let erro of mensagensErro" class="mdl-textfield__error">
```

```
    {{erro.message}}: {{erro.value}}
  </li>
</ul>
```

MÓDULOS NO ANGULAR

Uma das vantagens do Angular é o seu sistema próprio de módulos, onde podemos organizar páginas e features da aplicação de maneira isolada, ou seja, separando um conjunto de componentes, diretivas, pipes, serviços etc que pertencem à um módulo específico. Porém isto não significa uma organização de arquivos em uma pasta, e sim, carregar estes recursos do módulo específico apenas quando o usuário acessá-lo através de sua página, ou seja, um carregamento de recursos sob demanda, possibilitando uma experiência inicial mais rápida com a aplicação, em conjunto uma arquitetura de projeto organizada e contextualizada.

30.1 NGMODULE

Os módulos do Angular [1] são classes TypeScript anotadas com o *decorator* `@NgModule`, e não são a mesma coisa que módulos do JavaScript. Quando um novo projeto Angular é criando, é necessário haver um módulo `root` (raiz) que irá fazer a configuração de *bootstrapping* (inicialização) da aplicação, no caso o `AppModule`.

Através do *decorator* `@NgModule`, podemos passar como parâmetro um objeto com propriedades que configuram os recursos de um módulo, sendo as mais comuns:

- `declarations` : recebe uma lista (*array*) de componentes, diretivas e pipes que fazem parte deste módulo;
- `exports` : define o conjunto de declarações (componentes, pipes etc) que ficam disponíveis (públicos) para outros módulos;
- `imports` : importa uma lista (*array*) de outros módulos que serão utilizados pelos componentes declarados neste módulo;
- `providers` : provê uma lista de serviços criados por nós que dependem de injeção de dependência. Caso seja o `AppModule`, os serviços ficam disponíveis para toda a aplicação.

Em `AppModule` também há a propriedade `bootstrap` que só é utilizada para este modulo, onde é informado os componentes de "boot" ou inicio que serão exibidos por primeiro na aplicação. Abaixo como é o `AppModule` básico de uma aplicação:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
```

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

30.2 CARREGAMENTO PREGUIÇOSO

No momento em que a Web começou a ser utilizada em dispositivos móveis e portanto a consumir dados móveis, várias técnicas de performance para páginas tiveram que ser elaboradas para que um usuário não terminasse com seu pacote de dados ao acessar um site ou aplicativo.

Uma destas técnicas que logo se tornou um padrão de projeto foi a **lazy-load** [2], ou seja, carregamento preguiçoso. No caso de uma aplicação SPA, isto se viabiliza por meio de carregar arquivos JavaScript sob demanda, já que uma aplicação SPA tem como entregável apenas 1 arquivo HTML, 1 CSS e 1 ou pouco mais que 1 arquivo JS.

Em uma SPA que não contém *lazy-load*, após o empacotamento, todas as "n" páginas ficarão em um único arquivo JavaScript, este que potencialmente conterá um tamanho consideravelmente grande para *download*. Fazendo com que a boa experiência do usuário que acessa pela primeira vez a aplicação seja comprometida devida a demora para abrir a página.

Os módulos do Angular viabilizam a aplicação da técnica de *lazy-load*, onde cada página da

aplicação terá seu próprio módulo, que será carregado pelo `RouterModule` apenas se a rota foi acessada, evitando assim o carregamento desnecessário de código.

30.3 CRIAÇÃO DE MÓDULOS COM A CLI

O primeiro módulo da aplicação não foi feito manualmente por nós, e sim automaticamente pela CLI do Angular, portanto ela sabe como construir um módulo, nos poupando este trabalho repetitivo, pois a estrutura inicial de um módulo é sempre a mesma, apenas muda seu nome.

Com o comando `generate` conseguimos gerar módulos desta forma:

```
ng generate module nome-do-modulo
```

Caso já tenha uma pasta criada com o nome do módulo, basta fazer:

```
ng generate module modules/cadastro
```

Irá criar a pasta `modules` caso não existe, e dentro irá criar uma pasta `cadastro` com o arquivo `cadastro.module.ts`, que terá a estrutura abaixo:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  declarations: [],
  imports: [
    CommonModule,
  ]
})
export class CadastroModule { }
```

CommonModule

Quando criamos um módulo pela CLI, ela por padrão já importa o `CommonModule`, porém para que ele serve?

O `CommonModule` [3] é importante pois ele disponibiliza para os componentes do módulo onde ele foi inserido todas as diretivas e pipes comuns, como `ngClass`, `ngFor`, `ngIf` etc...

Referências

1. <https://angular.io/guide/ngmodules>
2. https://en.wikipedia.org/wiki/Lazy_loading
3. <https://angular.io/api/common/CommonModule>

EXERCÍCIO: TRABALHANDO COM MÓDULOS

31.1 OBJETIVO

Agora que a página de cadastro de usuários está completa, se paramos para analisar, quantas vezes abrimos a página de cadastro de um site? Pouquíssimas vezes, e a aplicação da maneira que está agora, carrega o código-fonte de todas as páginas o tempo todo. É então que entra em cena a necessidade do real uso do sistema de módulos do Angular, que não é apenas poder organizar o código em pastas ou abstrair em classes, e sim fazer carregamento destes códigos por demanda, ou seja, usar a muito comentada abordagem *lazy-load* (carregamento preguiçoso). Neste primeiro momento vamos apenas preparar o terreno para implementar esta abordagem.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

31.2 PASSO A PASSO COM CÓDIGO

1. Vamos criar um módulo do Angular para cada página da aplicação. **Cadastro, Login e Caixa de Entrada:**

Cadastro

```
#./src/app/modules/cadastro/cadastro.module.ts
```

```
import { NgModule } from '@angular/core';

@NgModule({
  declarations: [],
  imports: []
})
export class CadastroModule {}
```

Login

#./src/app/modules/login/login.module.ts

```
import { NgModule } from '@angular/core';

@NgModule({
  declarations: [],
  imports: []
})
export class LoginModule {}
```

Caixa de entrada

#./src/app/modules/caixa-de-entrada/caixa-de-entrada.module.ts

```
import { NgModule } from '@angular/core';

@NgModule({
  declarations: [],
  imports: []
})
export class CaixaDeEntradaModule {}
```

2. Agora vamos **declarar** e **importar** tudo relativo à cada módulo. Deverá ficar parecido com os códigos abaixo:

Cadastro

#./src/app/modules/cadastro/cadastro.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ReactiveFormsModule, FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';
import { CadastroComponent } from './cadastro.component';
import { CmailFormGroupComponent } from 'src/app/components/cmail-form-group/cmail-form-group.component';
import { CmailFormFieldDirective } from 'src/app/components/cmail-form-group/cmail-form-field.directive';

@NgModule({
  declarations: [
    CadastroComponent,
    CmailFormGroupComponent,
    CmailFormFieldDirective
  ],
  imports: [
```

```

    CommonModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule
],
})
export class CadastroModule { }

```

Login

`#./src/app/modules/login/login.module.ts`

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { LoginComponent } from './login.component';

@NgModule({
  declarations: [LoginComponent],
  imports: [CommonModule]
})
export class LoginModule { }

```

Caixa de entrada

`#./src/app/modules/caixa-de-entrada/caixa-de-entrada.module.ts`

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { CaixaDeEntradaComponent } from './caixa-de-entrada.component';

@NgModule({
  declarations: [ CaixaDeEntradaComponent ],
  imports: [
    CommonModule,
    FormsModule,
  ]
})
export class CaixaDeEntradaModule { }

```

3. Em `AppModule`, **retirar** todas as declarações e importações e **manter apenas**: a declaração de `AppComponent` e `BrowserModule`:

`#./src/app/app.module.ts`

```

//... imports omitidos
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Ao fazer esta primeira refatoração, temos um problema, **CadastroComponent** e **CaixaDeEntradaComponent** passam a desconhecer **HeaderComponent**, pois mesmo se **HeaderComponent** estivesse declarado em **AppModule**, ambos módulos estariam declarados em um nível abaixo de **HeaderComponent**, portanto não conseguiram reconhecê-lo. Para isto vamos criar um módulo que exporta os componentes da pasta **components** e importar em cada módulo específico.

4. Em `src/app/components` crie o módulo **SharedComponentsModule**, que exportará o **HeaderComponent**:

```
#./src/app/components/shared-components.module.ts
```

```
import { NgModule } from '@angular/core';
import { HeaderComponent } from './header/header.component';
import { CommonModule } from '@angular/common';

@NgModule({
  declarations: [HeaderComponent],
  imports: [CommonModule],
  exports: [HeaderComponent]
})
export class SharedComponentModule{}
```

5. Importe **SharedComponentModule** dentro de **CadastroModule** e **CaixaDeEntradaModule**.
6. **Opcional** : separe também **CmailFormGroupComponent** e **CmailFormFieldDirective** e um módulo **CmailFormModule**; Obs: neste, não esqueça de importar o **CommonModule** .

SUBROTEAMENTO: ROTAS FILHAS

O sistema de roteamento do Angular permite lógicas mais complexas do que simplesmente carregar componentes para uma rota. Podemos criar um esquema de subroteamento por módulo da aplicação, que viabilizará o *lazy-load*.

Basicamente, cada módulo de uma página terá seu próprio módulo de roteamento, e este trará a atual instância de `RouterModule` que será modificada com a adição de uma nova lista de rotas pertencentes à este módulo/página.

Com isto, devemos modificar o arquivo `app.routes.ts`, que ao invés de apenas exportar rotas, agora se conterá o `AppRoutingModule`, que irá importar as rotas "raiz" (*root*) da aplicação. Também **renomeamos** o nome do arquivo para algo mais coerente com a atual função `app-routing.module.ts`.

```
const rotasApp:Routes = [
  {
    path: 'cadastro'
  },
  {
    path: 'login'
  },
  {
    path: 'inbox'
  },
  {
    path: ''
  },
  {
    path: '***'
  }
];

@NgModule({
  imports: [
    //Aqui importamos as rotas
    RouterModule.forRoot(rotasApp)
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

Perceba que retiramos as propriedades `component` que havia em cada rota, pois logo mais iremos

carregar um módulo ao invés de um componente por rota.

Em AppModule devemos importar AppRoutingModule e retirar a antiga maneira de fazer as rotas, ficando apenas com o mínimo necessário:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule //importou!
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

32.1 MÓDULOS POR ROTAS

Para carregar o módulo da página daquela rota, usamos loadChildren no lugar de component . A partir da versão 8 do Angular, a forma de importar um módulo mudou bastante, ficou como no exemplo abaixo, onde usamos a notação de importação dos arquivos JavaScript, ou seja, uma *arrow function* importando de maneira assíncrona o módulo, e este que logo está sendo então acessado dentro da função de *callback* do then :

```
const rotasApp: Routes = [
  {
    path: 'cadastro',
    loadChildren: () => import('./modules/cadastro/cadastro.module').then(m => m.CadastroModule)
  },
  {
    path: 'login',
    loadChildren: () => import('./modules/login/login.module').then(m => m.LoginModule)
  },
  {
    path: 'inbox',
    loadChildren: () => import('./modules/caixa-de-entrada/caixa-de-entrada.module').then(m => m.CaixaDeEntradaModule)
  },
  {
    path: '',
    redirectTo: 'inbox',
    pathMatch: 'full'
  },
  {
    path: '**',
    component: PaginaNaoEncontradaComponent
  }
];
```

Em versões anteriores à 8 do Angular, o valor de `loadChildren` era como no exemplo abaixo:

```
{ path: 'cadastro',
  loadChildren: 'src/app/modules/cadastro/cadastro.module#CadastroModule'
}
```

patchMatch

Uma rota que fará um redirecionamento requer a propriedade `patchMatch` para informar ao `router` como corresponder uma URL ao caminho de uma rota. O `router` gera um erro se você não o fizer. No exemplo acima, o `router` deve selecionar a rota para `CaixaDeEntradaModule` apenas quando a URL inteira corresponder a '' , portanto, defina o valor `patchMatch` como 'full' .

Tecnicamente, `patchMatch: 'full'` resulta em uma rota atingida quando os segmentos restantes e incomparáveis do URL correspondem '' . Neste exemplo, o redirecionamento está em uma rota de nível superior, portanto a URL restante e a URL inteira são a mesma coisa.

O outro valor possível do `patchMatch` é 'prefix' , que informa ao roteador para corresponder à rota de redirecionamento quando a URL restante começa com o caminho do prefixo da rota de redirecionamento.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

32.2 ADICIONADO UM SUBROTEAMENTO

Agora cada, página tem um módulo, e também deverá ter seu próprio módulo de roteamento. Por exemplo o `LoginRoutingModule` , que deve conter sua própria lista de rotas, lembrando que esta lista, serão segmentos de URL adicionados após '/login'. Portanto na "raiz" de login, queremos carregar o `LoginComponent` .

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { LoginComponent } from './login.component';
```

```

const routes: Routes = [
  {
    path: '',
    component: LoginComponent,
    pathMatch: 'full'
  }
];

@NgModule({
  exports: [RouterModule]
})
export class LoginRoutingModule { }

```

E para que a instância do `router` leia estas rotas, devemos usar o método `forChild` que receberá a lista de rotas deste módulo, e seu retorno deverá ser importado no *decorador* `NgModule`:

```

import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { LoginComponent } from './login.component';

const routes: Routes = [
  {
    path: '',
    component: LoginComponent,
    pathMatch: 'full'
  }
];

@NgModule({
  //Importou rotas filhas! ↴
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class LoginRoutingModule { }

```

Agora basta importar o `LoginRoutingModule` no módulo da página, no caso em `LoginModule`:

```

@NgModule({
  declarations: [LoginComponent],
  exports: [LoginComponent],
  imports: [
    CommonModule,
    //Importou subroteamento ↴
    LoginRoutingModule
  ]
})
export class LoginModule {}

```

32.3 ÁRVORE DE ROTEAMENTO

A imagem abaixo ilustra em forma de um diagrama de como os arquivos estão conectados fazendo este fluxo de importação sob demanda de arquivos:

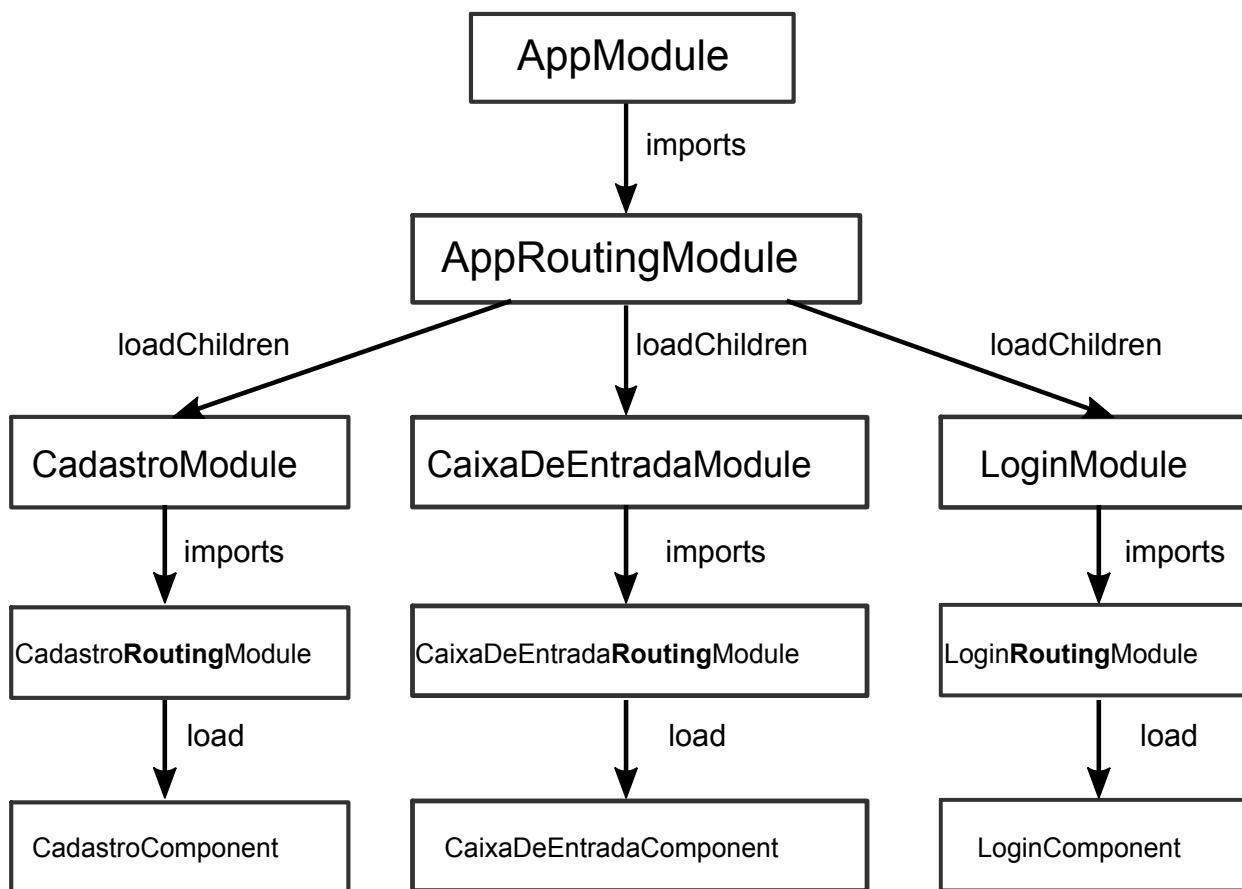


Figura 32.1: Diagrama da ordem de importação dos arquivos

Referências

- <https://angular.io/guide/router#set-up-redirects>
- <https://vsavkin.tumblr.com/post/146722301646/angular-router-empty-paths-componentless-routes>
- <https://angular.io/guide/lazy-loading-ngmodules>

EXERCÍCIO: SUBROTEAMENTO COM ROTAS FILHAS

33.1 OBJETIVO

Com os módulos para cada página devidamente criados, vamos criar o módulo de roteamento principal da aplicação, no qual indicaremos agora rotas que carregam módulos ao invés de componentes. Para isto também vamos ter que criar subroteamentos, ou seja, um módulo de roteamento por página.

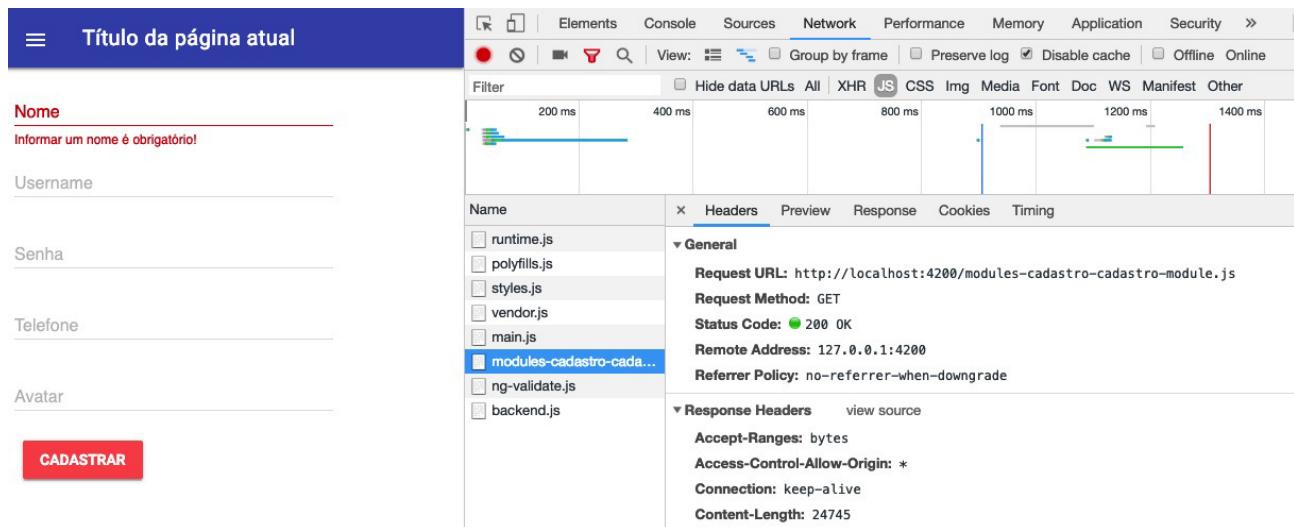


Figura 33.1: Módulo modules-cadastro-cadastro-module.js sendo carregado depois

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

33.2 PASSO A PASSO COM CÓDIGO

1. Vamos criar um novo arquivo, que será o módulo de roteamento da nossa app. Será o `app-routing.module.ts` que deverá ficar na raiz da pasta `app`.
2. Em `app-routing.module.ts` vamos criar uma classe que será o módulo de roteamento principal da aplicação. A classe se chamará **AppRoutingModule**, e vamos decorar ela com `@NgModule`:

`#./src/app/app-routing.module.ts`

```
`ts import { NgModule } from '@angular/core'; @NgModule() export class AppRoutingModule{}`
```

3. Em **AppModule**, vamos importar **AppRoutingModule** e remover o antigo **ModuloRoteamento**. Também vamos remover **CadastroModule** pois quem irá carregar este módulo será o **AppRoutingModule**:

`#./src/app/app.module.ts`

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule //importou!
  ],
  providers: []
})
```

```

    bootstrap: [AppComponent]
})
export class AppModule { }

```

Obs: remova os imports que não estão sendo mais usados!

- Para que o **AppRoutingModule** saiba carregar o módulo de uma rota, vamos declarar nele, uma lista de rotas, similar ao que fizemos no jeito antigo das rotas. Porém ao invés de indicar um componente vamos carregar um módulo "filho", usando a propriedade **loadChildren**. O valor dessa propriedade, desde a versão 8 do Angular, é uma função que importa o módulo.

`#./src/app/app-routing.module.ts`

```

import { NgModule } from '@angular/core';
import { Routes } from '@angular/router';

const rotas: Routes = [
{
  path: 'cadastro',
  loadChildren: () => import('./modules/cadastro/cadastro.module').then(m => m.CadastroModule)
},
{
  path: '**',
  redirectTo: 'cadastro',
  pathMatch: 'full'
}
]

@NgModule()
export class AppRoutingModule{}

```

- Após fazer a lista de rotas, continuamos informar a lista para **RouterModule**, porém, podemos chamar o método **forRoot** direto na diretiva **@NgModule()**:

`#./src/app/app-routing.module.ts`

```

import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const rotas: Routes = [
{
  path: 'cadastro',
  loadChildren: () => import('./modules/cadastro/cadastro.module').then(m => m.CadastroModule)
},
{
  path: '**',
  redirectTo: 'cadastro',
  pathMatch: 'full'
}
]

@NgModule({
  imports: [
    //Importou e leu rotas ↴
    RouterModule.forRoot(rotas)
  ],
  //Exportou módulo configurado
})

```

```
    exports: [RouterModule]
})
export class AppRoutingModule{}
```

6. Vamos precisar criar agora um sub sistema de roteamento para o módulo de cadastro, iniciando pela sua classe **CadastroRoutingModule**:

```
#./src/app/modules/cadastro/cadastro-routing.module.ts
```

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { CadastroComponent } from './cadastro.component';

const rotasCadastro: Routes = [
  {path: '', component: CadastroComponent}
]

@NgModule({
  imports: [
    RouterModule.forChild(rotasCadastro)
  ],
  exports:[
    RouterModule
  ]
})
export class CadastroRoutingModule {}
```

7. E então, o recém criado submódulo de roteamento `CadastroRoutingModule`, deve ser importado em **CadastroModule**:

```
#./src/app/modules/cadastro/cadastro.module.ts
```

```
imports: [
  CommonModule,
  ReactiveFormsModule,
  SharedComponentModule,
  //importou CadastroRoutingModule
  CadastroRoutingModule
],
```

8. Por fim, faça o sub roteamento os módulos de **Login** e **Caixa de Entrada**.

SOLICITANDO AUTENTICAÇÃO

Para desenvolver a funcionalidade de acessar uma conta em um sistema, é necessário primeiro poder criar contas de usuários (e ter usuários). Para isto, usamos o auxílio de uma aplicação *server side* (lado servidor), no nosso caso o CMail Back, que é capaz de receber requisições do CMail expondo uma API, e assim obter os dados de um usuário informados por um formulário e para registrar eles no banco de dados do backend.

Agora, para acessar esta conta cadastrada, a documentação da API deve nos informar quais dados são necessários e através de um formulário de login coletamos os dados solicitados, que devem ser enviados através de uma requisição HTTP para a API. A API então irá responder, no caso do CMail Back, a resposta é um objeto com os dados do usuário como, nome, email, avatar e também um **token**. Caso contrário um erro de informações inválidas serão retornados.

O token retornado pela API está no padrão JWT (JSON Web Token) [1], hoje muito utilizado para autenticação e troca de dados entre duas aplicações. Esta será a chave de acesso às páginas do CMail que precisarão das informações do usuário, como por exemplo a caixa de entrada.

34.1 AUTENTICAÇÃO E RESPONSABILIDADE DO LADO CLIENTE

A autenticação em um sistema sempre é dada pela aplicação server side, ela é a garantia que o usuário existe e é verdadeiro.

Porém, a aplicação client side precisa armazenar sua chave de autenticação entregue pela API, para que seja usada em todo local restrito à usuários autenticados. Esta chave é geralmente representada por um token.

A informação de todos os tokens ativos e quando cada um vai expirar é gerenciada pelo pela aplicação server side.

```
{
  "email": "bob@cmail.com.br",
  "name": "Sponge Bob",
  "avatarUrl": "http://localhost:3000/public/bob.jpg", "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InNkQGNtYWlsLmNvbS5iciIsImhdCI6MTU2NzgwNTQ5MywiZXhwIjoxNTY4NDEwMjkzfQ.fA8TvkLp-n-Me6GMrDyBf2JIr0_DuUlGVDCNIOXqGEo"
}
```

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

34.2 ARMAZENAMENTO LOCAL

Quando a resposta da API acontecer, e a mesma contiver o token, podemos armazenar ele em um banco de dados temporário que o navegador oferece por meio de uma API JavaScript, o **localStorage** [2].

Isto irá nos ajudar para quando necessitarmos acessar páginas restritas à usuários autenticados. No exemplo abaixo, como definir uma chave no localStorage para armazenar o valor do token retornado pelo API:

```
this.httpClient
  .post('http://localhost:3200/login', this.login)
  .subscribe(
    (response: any) => {
      localStorage.setItem('cmail-token', response.token);
    },
    (error) => console.error(error)
  )
```

Para acessar uma chave no localStorage:

```
localStorage.getItem('cmail-token');
```

Para remover uma chave no localStorage:

```
localStorage.removeItem('cmail-token');
```

Referências

1. <https://tools.ietf.org/html/rfc7519>
2. <https://developer.mozilla.org/pt-BR/docs/Web/API/Window/Window.localStorage>

EXERCÍCIO: COMEÇANDO A TELA DE LOGIN

35.1 OBJETIVO

Feito o cadastro de usuários, agora precisamos nos autenticar na API do CMail. Vamos então construir o formulário de login do client do CMail em **LoginComponent**. Com ele, vamos coletar os dados de um usuário já cadastrado, enviar para na API e armazenar o token de autenticação no localStorage do navegador. Também trataremos os erros do formulário e de requisição HTTP.

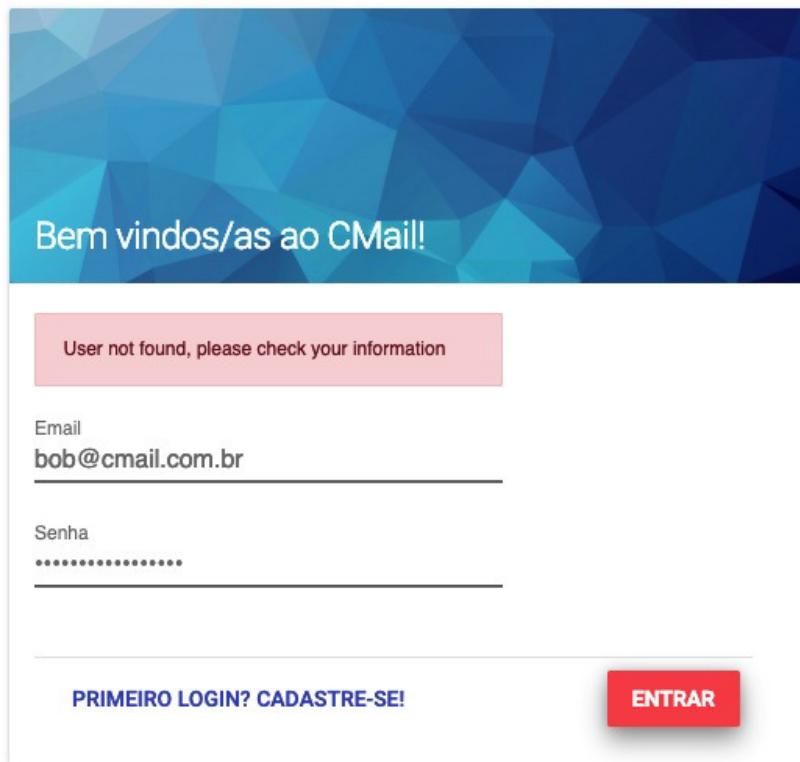


Figura 35.1: Form de login com mensagem de validação

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

35.2 PASSO A PASSO COM CÓDIGO

Para que não seja necessário digitar o código dos arquivos de HTML ou CSS, baixe-os a partir do repositório dos arquivos do curso, pasta **15**:

1. Copie os arquivos do curso, pasta 15, onde teremos o template (HTML) e CSS de **LoginComponent**:
<https://github.com/caelum/projeto-js45/tree/master/15/src/app/modules/login>
2. Adicione os campos de **email** e **senha** usando o **CmailFormComponent** e a diretiva **cmailFormField**, usando o **ngModel** para fazer o bind de dados e validação, com isto prepare mensagens de validação básicas, como as de campo obrigatório:

`#./src/app/modules/login/login.component.html`

```
<cmail-form-group [campo]="email">
  <input type="email" #email="ngModel" [(ngModel)]="login.email" name="email" cmailFormField required autofocus>
  <span class="mdl-textfield__error" *ngIf="email.invalid">
    Informar um email é obrigatório!
  </span>
</cmail-form-group>

<cmail-form-group [campo]="senha">
  <input type="password" #senha="ngModel" [(ngModel)]="login.password" name="senha" cmailFormField required autofocus>
```

```

<span class="mdl-textfield_error" *ngIf="senha.invalid">
  Informar uma senha é obrigatória!
</span>
</cmail-form-group>

```

Pontos de atenção:

- Temos que declarar a variável de template de cada `input`, para pegarmos seu status de validação passar para o `*ngIf` das mensagens;
 - Na tag `cmail-form-group` não esqueça de passar `[campo]` com a referência a variável de template do respectivo `input`, para que as mensagens de validação apareçam.
3. Na tag `<form>` vamos fazer o bind de `handleLogin` com o evento `ngSubmit`, onde passamos a referência do formulário, através da criação de uma variável de template, como parâmetro para validarmos a submissão:

`#./src/app/modules/login/login.component.html`

```

<form (ngSubmit)="handleLogin(formLogin)" #formLogin="ngForm" autocomplete="off" class="mdl-card__supporting-text">

```

4. Agora vamos usar o **HttpClient** para realizar o POST com os dados do formulário coletados pelo `ngModel`.

`#./src/app/modules/login/login.component.ts`

```

//código anterior omitido
export class LoginComponent implements OnInit {

  login = {
    email: '',
    password: ''
  }

  constructor(private httpClient: HttpClient) { }

  ngOnInit() {}

  handleLogin(formLogin: NgForm){

    if(formLogin.valid){

      this.httpClient
        .post('http://localhost:3200/login', this.login)
        .subscribe(
          (response) => {
            console.log(response);
            console.log('deu certo');
          },
          (error) => {
            console.error(error);
            console.log('deu ruim');
          }
        )
    }
  }
}

```

```
 }
}
```

5. Caso tenhamos um erro, podemos tratá-lo exibindo uma mensagem no template. Vamos criar uma propriedade **mensagemErro** na classe de login, e associar o retorno do callback do erro com ela:

```
#./src/app/modules/login/login.component.ts
```

```
(responseError: HttpErrorResponse) => {
  this.mensagemErro = responseError.error.body
}
```

6. No template podemos fazer um **ngIf** que verifica se **mensagemErro** tem algum valor caso positivo, exibindo os erros:

```
#./src/app/modules/login/login.component.html
```

```
<section *ngIf="mensagemErro" class="mdl-textfield is-invalid">
  <p class="mdl-textfield_error">
    {{mensagemErro.message}}
  </p>
</section>
```

7. Sem erros, vamos armazenar o **token** que retornamos da API, no **localStorage** do navegador, para usar ele para acessar a caixa de entrada.

```
#./src/app/modules/login/login.component.ts
```

```
(response: any) => {
  localStorage.setItem('cmail-token', response.token);
}
```

Ao acessar no console do navegador, conseguimos verificar o token salvo:
`localStorage.getItem('cmail-token')`

SERVIÇOS NO ANGULAR

Um serviço é uma categoria ampla que abrange qualquer valor, função ou recurso que a aplicação necessita. Traduzindo, um serviço geralmente é uma classe com um objetivo estrito e bem definido, na qual deve fazer algo específico e fazê-lo bem.

O Angular distingue componentes de serviços para aumentar modularidade e reutilização. Ao separar a funcionalidade relacionada à exibição de um componente de outros tipos de processamento, podemos tornar as classes de componentes mais enxutas e eficientes.

Idealmente, o trabalho de um componente é permitir a experiência do usuário e nada mais. Um componente deve apresentar propriedades e métodos através associação de dados (*data-binding*), mediando a visualização (renderizada pelo template) e a lógica da aplicação (que geralmente usa parte do modelo/*model*).

Um componente pode delegar determinadas tarefas aos serviços, como buscar dados do servidor, validar a entrada do usuário etc. Ao definir estas tarefas de processamento em uma classe de serviço injetável (*injectable service class*), elas podem ser disponibilizadas para qualquer componente. Assim é possível tornar sua aplicação mais adaptável, pois é possível injetar em um componente diversos provedores (*providers*) conforme apropriado para diferentes circunstâncias.

O Angular não impõe estes princípios de arquitetura de código. Porém, o Angular nos ajuda a seguir esses princípios, facilitando o desenvolvimento da lógica da aplicação através dos serviços e disponibilizando estes aos componentes por meio da injeção de dependência. [1]

36.1 CRIANDO UM SERVIÇO

A injeção de dependência (DI) é conectada à estrutura Angular e usada em todos os lugares para fornecer aos componentes os serviços ou outras coisas que eles precisam. Componentes consomem serviços; isto é, você pode injetar um serviço em um componente, dando acesso ao componente a classe do serviço, não havendo a necessidade de instânciar a classe de um serviço.

Decorator @Injectable()

Para definir uma classe como um serviço no Angular, usamos *decorator* `@Injectable()` para fornecer os metadados que permitem ao Angular injetá-lo em um componente como uma dependência.

Providers e Injectors

O injetor (*injector*) é o mecanismo principal para a injeção de dependência. Apenas o Angular cria um injetor (nós não precisamos criar injetores), e o disponibiliza para toda a aplicação durante o processo de inicialização (*bootstrap*).

Um injetor cria dependências e mantém um "container" de instâncias das dependências para reutilizar, se possível.

Um provedor (*provider*) é um objeto que informa ao injetor como obter ou criar uma dependência.

Para qualquer dependência necessária na aplicação, devemos registrar um provedor (*provider*), para que o **injetor** possa usar o **provedor** para criar novas instâncias. No caso de um serviço, o provedor normalmente é a própria classe de serviço.

Uma dependência não precisa ser um serviço. Ela pode ser uma função ou um valor, por exemplo.

Quando o Angular cria uma nova instância de um componente, ele determina quais serviços ou outras dependências esse componente precisa, observando os tipos de parâmetros do construtor. Por exemplo, o construtor de `LoginComponent` precisa de `LoginService`.

```
constructor(private loginService: LoginService) {}
```

Quando o Angular descobre que um componente depende de um serviço, ele primeiro verifica se o injetor possui alguma instância existente desse serviço. Se uma instância de serviço solicitada ainda não existir, o injetor fará uma usando um provedor registrado e a adicionará ao injetor antes de retornar o serviço ao Angular.

Quando todos os serviços solicitados foram resolvidos e retornados, o Angular pode chamar o construtor do componente com esses serviços como argumentos.

Provendo um serviço injetável

Um serviço deve ser registrado em pelo menos um provedor (*provider*). O provedor pode ser informado diretamente nos metadados do serviço (no *decorator* `@Injectable()`), disponibilizando esse serviço em qualquer lugar. Ou podemos registrá-lo nos metadados dos módulos (`@NgModule()`) ou em componentes específicos (`@Component()`).

Por padrão, o comando Angular CLI `ng generate service` já registra um provedor com no injetor raiz do serviço recém criado, incluindo os metadados do provedor no decorador `@Injectable()`. Por exemplo:

```
@Injectable({  
  providedIn: 'root',  
})
```

Quando você fornece o serviço no nível **root**, o Angular cria uma instância única (Singleton) [2] e compartilhada deste serviço e o injeta em qualquer classe que o solicitar. O registro do provedor nos metadados de `@Injectable()` também permite que o Angular otimize a aplicação, removendo o serviço na aplicação compilado, caso este serviço não for usado.

Quando registramos um serviço em um provedor de um `NgModule` específico, a mesma instância deste serviço está disponível apenas para todos os componentes deste `NgModule`. Para se registrar nesse nível, use a propriedade `provider` do decorador `@NgModule()`:

```
@NgModule({  
  providers: [  
    BackendService,  
    Logger  
  ],  
  ...  
})
```

Ao registrar um provedor ao nível do componente, você obtém uma nova instância do serviço com cada nova instância desse componente. No nível do componente, registre um provedor de serviços na propriedade `providers` dos metadados de `@Component()`.

```
@Component({  
  selector: 'cmail-login-component',  
  templateUrl: './login.component.html',  
  providers: [ LoginService ]  
})
```

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

36.2 TIPOS CUSTOMIZADOS

Os tipos são nada mais nada menos que classes que criamos. Podemos criar classes para estruturas de dados que vamos usar na aplicação, facilitando o desenvolvimento com um bom suporte de tipos.

Por exemplo, podemos criar uma classe para a estrutura de dados necessária para realizar o *login* da aplicação:

```
export class Login {  
  email = '';  
  password = '';  
}
```

Caso queremos definir um tipo de um objeto como `Login`, e este objeto estiver em outro formato ou idioma, podemos criar no construtor uma maneira de receber um objeto neste outro formato:

```
export class Login {  
  email = '';  
  password = '';  
  
  constructor({ email, senha }) {  
    this.email = email;  
    this.password = senha;  
  }  
  
}
```

Neste exemplo usamos o recurso de desestruturação de objetos do JavaScript (*object destructuring*), onde o construtor espera receber um objeto com as propriedades `email` e `senha`.

36.3 ASSINATURA DE MÉTODOS NO TYPESCRIPT

Com TypeScript, podemos melhorar os métodos das nossas classes definindo os tipos dos parâmetros e o tipo do retorno. Facilitando o uso do código e também garantindo um bom padrão de código.

Para declarar os tipos nos parâmetros de um método, basta usarmos os `:` logo após a definição do nome do parâmetro, e então trazemos o tipo deste parâmetro, **por exemplo**:

```
autenticar(dadosLogin: Login) {  
  return this.http.post(this.api, dadosLogin)  
}
```

Também é possível definir um tipo de um objeto desestruturado. Por exemplo no construtor da classe `Login`, após a declaração do objeto, criamos o objeto de tipo onde após o nome da propriedade definimos o tipo dela:

```
export class Login {  
  email = '';  
  password = '';  
  
  constructor(  
    email: string,  
    password: string  
  ) {  
    this.email = email;  
    this.password = password;  
  }  
}
```

```
{ email, senha }: {email:string, senha:string}  
) {  
  this.email = email;  
  this.password = senha;  
}  
}
```

O retorno de um método pode ser definido em sua assinatura, logo após a declaração dos parâmetros, também usando os `:`, **por exemplo**:

```
autenticar(dadosLogin: Login): Observable<Object> {  
  return this.http.post(this.api, dadosLogin)  
}
```

Referências

1. <https://angular.io/guide/architecture-services>
2. <https://angular.io/guide/singleton-services>
3. <https://angular.io/guide/styleguide#providing-a-service>

EXERCÍCIO: MELHORANDO A QUALIDADE DO CÓDIGO, CRIANDO UM SERVIÇO

37.1 OBJETIVO

Até então, todo o momento que tivemos uma lógica ou regra do CMail acabamos por criar ela diretamente atrelada ao componente. Em uma aplicação simples, no dia a dia isso pode até parecer mais rápido para "ter as coisas prontas". Porém o ponto negativo, é que o projeto como um todo se tornará de difícil manutenção e escalabilidade. É importante deixar as regras de negócio o mais separado o possível da camada de aplicação. Com isso, **LoginComponent** não deverá ter a dependencia do **HttpClient**, nem saber que deve armazenar um token. O responsável pelas regras será o **serviço de login**.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

37.2 PASSO A PASSO COM CÓDIGO

- Vamos fazer uma classe **LoginService**, que se tornará um serviço para o **LoginComponent**, para isto devemos decorá-la com `Injectable()`. Esta classe, neste momento também conterá a propriedade `api`, com o endereço, a injeção de `HttpClient` no construtor e o método `logar`:

```
#./src/app/services/login.service.ts
```

```

import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable()
export class LoginService {

  api = 'http://localhost:3200/login'

  constructor(private http: HttpClient){}

  logar(dadosLogin){
    return this.http.post(this.api, dadosLogin)
  }
}

```

2. Vamos importar o **LoginService** em **LoginModule** numa nova propriedade de **NgModule** , a **providers** :

`#./src/app/modules/login/login.module.ts`

```

,providers: [
  LoginService
]

```

3. Agora podemos utilizar **LoginService** no lugar de **HttpClient** em **LoginComponent**. Aproveitando, já vamos fazer com que sejamos redirecionados para a rota `inbox` , chamando o **Router** , logo após que tivermos a resposta positiva do serviço de login:

`#./src/app/modules/login/login.component.ts`

```

import { Component, OnInit } from '@angular/core';
import { HttpErrorResponse } from '@angular/common/http';
import { NgForm } from '@angular/forms';
import { LoginService } from 'src/app/services/login.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

  mensagemErro;
  login = { email: '', password: ''};

  constructor(private loginService: LoginService
             ,private roteador: Router ) { }

  ngOnInit() {}

  handleLogin(formLogin: NgForm){
    if(formLogin.valid) {
      this.loginService
        .logar(this.login)
        .subscribe(
          () => this.roteador.navigate(['/inbox'])
        ,(responseError: HttpErrorResponse) => this.mensagemErro = responseError.error
    }
  }
}

```

```
        )
    }
}
```

4. Para finalizar, vamos fazer com que **LoginService** grave o **token** no `LocalStorage`, e devolva a resposta da API para os inscritos no serviço:

`#./src/app/services/login.service.ts`

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { map } from 'rxjs/operators';

@Injectable()
export class LoginService {

  api = 'http://localhost:3200/login';

  constructor(private http: HttpClient){}

  logar(dadosLogin){
    return this.http
      .post(this.api, dadosLogin)
      .pipe(
        map( response => {
          localStorage.setItem('TOKEN', response.token);
          return response;
        })
      )
  }
}
```

GUARDIÕES DE ROTAS

Até aqui, qualquer usuário pode navegar aonde quiser na aplicação a qualquer momento. E nem sempre isto é o certo a se fazer.

- Talvez o usuário não esteja autorizado a navegar para um determinado componente.
- Talvez o usuário precise fazer login (autenticar) primeiro.
- Talvez você deva buscar alguns dados antes de exibir o componente de destino.
- Você pode querer salvar as alterações pendentes antes de sair de um componente.
- Você pode querer perguntar ao usuário se não há problema em descartar alterações pendentes em vez de salvá-las.

Para isto, é possível adicionar proteções à configuração da rota para lidar com esses cenários, esta proteção chamamos de "*guard*".

38.1 O QUE É UM GUARD?

Na prática, um *guard* é uma classe usada através da injeção de dependência pelo `RouterModule`. Esta classe implementa uma das possibilidades de interface, que irá controlar o acesso do usuário em uma determinada rota.

O valor de retorno de um *guard* que controla o comportamento do *router*:

- Se retornar `true`, o processo de navegação continuará.
- Se retornar `false`, o processo de navegação será interrompido e o usuário permanecerá parado.
- Se ele retornar uma `UrlTree`, a navegação atual será cancelada e uma nova navegação será iniciada no `UrlTree` retornado.

Nota: O *guard* também pode dizer ao *router* para navegar para outro lugar, cancelando efetivamente a navegação atual. Ao fazer isso dentro de um *guard*, o mesmo deve retornar falso;

O *guard* pode retornar sua resposta booleana de forma síncrona. Mas em muitos casos, o *guard* não pode produzir uma resposta de forma síncrona. O *guard* poderia fazer uma pergunta ao usuário, salvar as

alterações no servidor ou buscar novos dados. Todas essas são operações assíncronas.

Assim, um *guard* de roteamento pode retornar uma `Observable<boolean>` ou uma `Promise<boolean>` e o roteador aguardará a *observable* resolver como verdadeiro ou falso.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

38.2 INTERFACES PARA O GUARD

O roteador suporta várias interfaces de guarda:

- `CanActivate` para mediar a navegação em uma rota.
- `CanActivateChild` para mediar a navegação em uma rota filha.
- `CanDeactivate` para mediar a navegação para longe da rota atual.
- `Resolve` para executar a recuperação de dados da rota antes da ativação da rota.
- `CanLoad` para mediar a navegação em um *feature module* carregado de forma assíncrona.

Você pode ter várias proteções em todos os níveis de uma hierarquia de roteamento. O roteador verifica os *guards* `CanDeactivate` e `CanActivateChild` primeiro, da rota filho mais profunda para o topo. Em seguida, ele verifica os *guards* do `CanActivate` de cima para baixo até a rota filho mais profunda. Se o *feature module* for carregado de forma assíncrona, a proteção `CanLoad` será verificada antes do carregamento do módulo. Se qualquer *guard* retornar falso, os *guards* pendentes que não foram concluídas serão canceladas e a navegação inteira será cancelada.

CanActivate : exigindo autenticação

As aplicações geralmente restringem o acesso a uma área de recursos com base em quem é o usuário. Podemos permitir acesso apenas a usuários autenticados ou a usuários com uma função específica. Podemos também bloquear ou limitar o acesso até a conta do usuário ser ativada.

O *guard* do tipo `CanActivate` é uma ferramenta para gerenciar essas regras de negócios de

navegação.

Abaixo, segue um exemplo de como fazer um *guard* implementando a `CanActivate`:

```
import { CanActivate, Router } from '@angular/router';
import { Injectable } from '@angular/core';
import { EmailService } from '../services/email.service';
import { map, catchError } from 'rxjs/operators';

@Injectable()
export class AuthGuard implements CanActivate {

  constructor(private roteador: Router,
    private servico: EmailService) {}

  canActivate() {
    return this.servico
      .validaToken()
      .pipe(
        map(response => response.ok),
        catchError(() => {
          localStorage.removeItem('cmail-token');
          this.roteador.navigate(['login']);
          return [false];
        })
      )
  }
}
```

O método do `EmailService` seria algo como:

```
validaToken(){
  return this.http
    .head(
      this.url
      ,{headers: this.cabecalho
      ,observe: 'response'}
    )
}
```

Propriedade `canActivate`

Agora para utilizar este *guard*, devemos usá-lo na definição da rota através da propriedade `canActivate`. No exemplo abaixo, estamos usando o *guard* na rota `inbox`:

```
const listaDeRotas: Routes = [
  {
    path: 'cadastro',
    loadChildren: () => import('../modules/cadastro/cadastro.module').then(m => m.CadastroModule)
  },
  {
    path: 'login',
    loadChildren: () => import('../modules/login/login.module').then(m => m.LoginModule)
  },
  {
    path: 'inbox',
    loadChildren: () => import('../modules/caixa-de-entrada/caixa-de-entrada.module').then(m => m.Caix
```

```

aDeEntradaModule)
    ,pathMatch: 'full'
    ,canActivate: [AuthGuard]
},
{
  path: '',
  redirectTo: 'inbox',
  pathMatch: 'full'
},
{
  path: '**',
  component: PaginaNaoEncontradaComponent
}
];
}

@NgModule({
  imports: [
    RouterModule.forRoot(listaDeRotas),
    HttpClientModule
  ],
  exports: [
    RouterModule
  ],
  providers: [
    AuthGuard,
    EmailService
  ]
})
export class AppRoutingModule {}

```

Perceba que o AuthGuard está provido nos metadados de `AppRoutingModule`, e não no `root`, pelos seus próprios metadados de `Injectable`, porém, esta é uma possibilidade.

Referências

1. <https://angular.io/guide/router#milestone-5-route-guards>

EXERCÍCIO: PROTEGENDO AS ROTAS DA APLICAÇÃO COM GUARDS

39.1 OBJETIVO

A listagem de emails só deve ser acessada caso o usuário esteja logado, ou seja, com um token válido salvo em seu **localStorage**. Precisamos adicionar uma verificação nesta rota, ou seja, um guarda de rota, que vai decidir se o poderá prosseguir ou não.



Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

39.2 PASSO A PASSO COM CÓDIGO

1. Vamos criar a classe **AuthGuard** em `guards/auth.guard.ts` . Esta classe é um pouco peculiar pois deve ser provida como um serviço, portanto é decorada com `@Injectable()` , e também deve implementar a interface **CanActivate**, que exige a implementação do método `canActivate` , neste método que deve haver toda a lógica da permissão da rota, e caso atenda os critérios, deverá retornar um valor booleano `true` , caso contrário, deve retornar um `false` .

`#./src/app/guards/auth.guard.ts`

```
import { Router, CanActivate } from '@angular/router';
import { Injectable } from '@angular/core';
```

```

@Injectable()
export class AuthGuard implements CanActivate {

  constructor(private roteador: Router){}

  canActivate(): boolean {

    if(localStorage.getItem('cmail-token')){
      return true;
    }
    else {
      this.roteador.navigate([''])
      return false
    }
  }
}

```

É obrigatório usar a interface **CanActivate**, pois o Angular usa a interface para saber que todos os itens do array que passamos na rota são desse tipo, e assim obrigar todas a ter a função **canActivate**.

- Em **ModuloRoteamento** no arquivo `app-routing.module.ts` , na rota da `inbox` que carrega o **CaixaDeEntradaComponent**, adicionamos uma terceira propriedade, a **canActivate** , que receberá como valor uma lista de classes que devem retornar a condição boleana se a rota pode ou não ser acessada, no caso quem vai fazer a condição para nós é a classe **AuthGuard**.

`#./src/app/app-routing.module.ts`

```

import { NgModule } from "@angular/core";
import { Routes, RouterModule } from "@angular/router";
import { AuthGuard } from './guards/auth.guard';

const rotas: Routes = [
{
  path: 'inbox',
  //Guard aqui!
  canActivate: [AuthGuard],
  loadChildren: () => import('./modules/caixa-de-entrada/caixa-de-entrada.module').then( m => m.CaixaDeEntradaModule)
},
{
  path: 'login',
  loadChildren: () => import('./modules/login/login.module').then(m => m.LoginModule)
},
{
  path: 'cadastro',
  loadChildren: () => import('./modules/cadastro/cadastro.module').then(m => m.CadastroModule)
}
,{
  path: '**',
  redirectTo: '/login',
  pathMatch: 'full'
}
]

@NgModule({
  imports: [RouterModule.forRoot(rotas)],
  exports: [ RouterModule ],
  providers: [AuthGuard] //Guard aqui!
})

```

```
export class AppRoutingModule {}
```

Detalhe: como AuthGuard é um serviço, deve ser informado no providers de **ModuloRoteamento**.

3. **Desafio** : como fazer o Guard verificar se o Token existe na API?

HTTP HEADERS, VARIÁVEIS DE AMBIENTE E CASTING

Quando conversamos com outras aplicações, estas tem seus protocolos para que possamos fazer operações nela, como enviar dados ou recuperar dados. Algumas APIs exigem que alguns dos seus *endpoints* recebam determinadas informações na requisição HTTP através do HTTP Headers, como por exemplo o formato do dado enviado e o formato que a aplicação requisitante quer a resposta, assim como pedir uma chave de acesso para autorizar ou não o acesso aos dados daquele *endpoint*, para isto devemos saber como enviar uma requisição com configurações no `headers`.

40.1 CONFIGURANDO HTTPHEADERS NO HTTPCLIENT

Em todos os métodos de `HttpClient` temos o último parâmetro que é um objeto com opções para configurar a requisição, e uma delas é a propriedade `headers` que deverá receber como valor uma instância de `HttpHeaders` como valor.

```
import { HttpHeaders } from '@angular/common/http';

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json',
    'Authorization': 'my-auth-token'
  })
};
```

Por padrão o `Content-Type` do `body` da requisição no `HttpClient` já é `application/json`, porém é possível sobrescrever este valor informando um novo no `HttpHeaders`.

Header de autorização

O header `Authorization` é usado para armazenar a chave de acesso à algum *endpoint* específico, esta chave de acesso deve ter sido dada à aplicação Angular (cliente da API) em algum momento anterior através de algum *login*. A API vai verificar se a chave enviada pela requisição HTTP existe e caso positivo irá responder ao cliente com os dados solicitados, caso contrário um erro de permissão negada é retornado ao cliente.

No caso do CMail, podemos informar o valor da chave acessando diretamente o item do

```
localStorage :  
  
import { HttpHeaders } from '@angular/common/http';  
  
const httpOptions = {  
  headers: new HttpHeaders({  
    'Content-Type': 'application/json',  
    'Authorization': localStorage.getItem('cmail-token')  
  })  
};
```

Depois basta usarmos `httpOptions` como as opções do método de `HttpClient`, por exemplo:

```
enviar(email: Email): Observable<Object> {  
  return this.http  
    .post(  
      'http://localhost:3200/emails/'  
      ,email  
      ,httpOptions //Aqui httpOptions!  
    )  
}
```

Tipos de HTTP Headers

Existem várias possibilidades de uso e configurações do HTTP Headers, portanto sempre consulte a documentação da API que irá utilizar e também conheça a referência completa de possibilidades no MDN [2]

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

40.2 VARIÁVEIS DE AMBIENTE

O Angular já vem com uma configuração inicial para variáveis de ambiente. Um exemplo para o uso destas variáveis é para o endereço da API. Geralmente quando desenvolvemos as APIs também tem ambientes de desenvolvimento, homologação, produção etc. Portanto nossa aplicação deve estar preparada para ser entregue nestes distintos ambientes. É impraticável e perigoso fazer estas alterações

manualmente a cada tipo de ambiente de destino, por isto que usamos as variáveis de ambiente.

Em um projeto existe a pasta `src/environments`, com dois arquivos, o `environment.ts` e `environment.prod.ts`, onde o arquivo `environment.ts` provê a configuração padrão. Podemos sobreescrever-lo ou a partir dele criar outros arquivos para cada tipo de ambiente. [3]

```
└─myProject/src/environments/
    └─environment.ts
    └─environment.prod.ts
    └─environment.stage.ts
```

O arquivo base `environment.ts`, contém uma configuração básica inicial:

```
export const environment = {
  production: false
};
```

O comando `ng build` vai usar este arquivo para definir o *target* (alvo) quando nenhum ambiente é especificado junto com o comando. Você pode adicionar outras variáveis, como propriedades adicionais no objeto do ambiente ou como objetos separados. Por exemplo aqui, adicionamos uma propriedade para padronizar a chamada da url da API que estamos usando:

```
#./src/environments/environment.ts

export const environment = {
  production: false,
  apiUrl: 'http://localhost:3200/'
};
```

Temos que adicionar a mesma propriedade nos outros arquivos de ambientes específicos do destino, como `environment.prod.ts`. A seguir como deve ficar o arquivo de variáveis de ambientes para produção:

```
#./src/environments/environment.prod.ts

export const environment = {
  production: true,
  apiUrl: 'http://localhost:3200/'
};
```

Usando as variáveis criadas

Nos arquivos que for necessário o uso das variáveis declaradas no arquivo `environment.ts`, basta importá-lo e trazer a referência de do objeto `environment` e portanto usar suas propriedades, como no exemplo:

```
import { HttpHeaders } from '@angular/common/http';
import { environment } from 'src/environments/environment';

// url construída com a variável de ambiente
const url = `${environment.apiUrl}/emails`;

const httpOptions = {
```

```

headers: new HttpHeaders({
  'Content-Type': 'application/json',
  'Authorization': localStorage.getItem('cmail-token')
})
};

```

40.3 CASTING NO RETORNO DO MÉTODO POST

Quando fazemos uma requisição algum método do `HttpClient`, o retorno padrão destes métodos é uma `Observable<Object>`, ou seja, uma representação genérica com `Object` da resposta da API. Porém, nesta resposta da API, muitas vezes recebemos um objeto com propriedade que precisamos acessar para seguir a lógica da aplicação.

No caso do serviço de emails, quando queremos listar os emails, a API nos retorna uma lista com de objetos com `id` (identificador) e a data de envio do email gerado pelo backend. E estes são dados muito importantes para a visualização destes emails. Portanto para acessar estes dados, o retorno não pode ser um `Object`, e sim um tipo específico que precisamos criar.

No exemplo abaixo, criamos um tipo `EmailOutputDTO`, que receberá um objeto qualquer com propriedades em inglês, e construirá um objeto do tipo `EmailOutputDTO` que tem as propriedades em português:

```

export class EmailOutputDTO {

  conteudo = '';
  dataEnvio = '';
  remetente = '';
  id = '';
  assunto = '';
  destinatario = '';

  constructor({content, created_at, from, id, subject, to}){
    this.conteudo = content;
    this.dataEnvio = created_at;
    this.remetente = from;
    this.id = id;
    this.assunto = subject;
    this.destinatario = to;
  }

}

```

Agora, vamos fazer com que o retorno do método listar seja uma lista de `EmailOutputDTO` fazendo o *casting* de `Object` na própria chamada do método `get` com a notação do "diamante" `<>`. Então a chamada do get fica: `get<EmailInputDTO[]>`, e isto é fazer o *casting*.

```

listar(): Observable<EmailOutputDTO[]> {
  return this.http
    .get<EmailInputDTO[]>(url, httpOptions)
    .pipe(
      map(listaEmailsApi => {
        return listaEmailsApi
      })
    )
}

```

```
        .map(emailIngles => new EmailOutputDTO(emailIngles))
    })
}
```

Referências

1. <https://angular.io/guide/http#http-headers>
2. https://developer.mozilla.org/pt_BR/docs/Web/HTTP/Headers
3. <https://angular.io/guide/build#configure-environment-specific-defaults>

EXERCÍCIO: SERVIÇO PARA ENVIAR EMAILS

41.1 OBJETIVO

Agora que temos acesso a um identificador de usuário (o token), podemos ajustar `handleNewEmail` para também gravar os dados do email na API. Porém, o formulário de email está em português e a API espera os dados em inglês. Como lidar com estas conversões? Vamos construir um modelo para o domínio da nossa aplicação que recebe um objeto qualquer, e a partir destes dados, constrói um objeto do tipo `Email` em português. Com isso fazer um serviço que fará a conversão dos dados em inglês, enviará o objeto para a API, e nos retornará um novo Email, confirmando o sucesso do envio.

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

41.2 PASSO A PASSO COM CÓDIGO

1. Vamos criar classe `Emails`, que será de modelo para os emails da aplicação, hoje apenas com as propriedades de email que estamos utilizando:

`#./src/app/models/email.ts`

```

export class Email {

  destinatario = '';
  assunto = '';
  conteudo = '';

  constructor(
    { destinatario, assunto, conteudo}:
      { destinatario: string, assunto: string, conteudo: string }
  ){
    this.destinatario = destinatario;
    this.assunto = assunto;
    this.conteudo = conteudo;
  }
}

```

2. Agora vamos criar **EmailService**. Por enquanto faremos apenas o método `enviar`, que deverá receber os dados do formulário (que estão em português) e converter ele para o padrão esperado da API, inglês. Após o retorno de **HttpClient**, recebemos os dados do email recém cadastrado em inglês, devemos convertê-lo para português usando o modelo **Email** e retornar isto; vamos usar os métodos `pipe` e `map` da RxJs para nos ajudar nesta tarefa:

`#./src/app/services/email.service.ts`

```

import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Email } from '../models/email';
import { map } from 'rxjs/operators';

@Injectable({
  providedIn: 'root'
})
export class EmailService {

  api = 'http://localhost:3200/emails';
  cabecalho = new HttpHeaders({ 'Authorization': localStorage.getItem('TOKEN') });

  constructor(private http: HttpClient){}

  enviar({destinatario, assunto, conteudo}){

    const emailParaApi = {
      to: destinatario,
      subject: assunto,
      content: conteudo
    }

    return this.http
      .post(this.api, emailParaApi, { headers: this.cabecalho })
      .pipe<Email>(
        map(
          (emailApi: any) => {
            return new Email({
              destinatario: emailApi.to,
              assunto: emailApi.subject,
              conteudo: emailApi.content
            })
          }
        )
      )
  }
}

```

```
        )
    }
}
```

Podemos usar o próprio *decorator* `@Injectable()` informando a propriedade `providedIn` para definir o escopo de injeção deste serviço. Aqui vamos deixar no `root` da aplicação.

Para conseguir enviar um email, devemos mandar no Header HTTP um **Authorization**, passando o token que está guardado no **localStorage**.

3. Está na hora de por nosso código para testar! Vamos chamar o **EmailService** em **CaixaDeEntradaComponent** e enviar um email para a API. Este deverá ser exibido em nossa humilde lista de emails:

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.ts
```

```
//código anterior omitido
export class CaixaDeEntradaComponent {

    private _isNewEmailFormOpen = false;
    emailList = [];
    email = { destinatario: '', assunto: '', conteudo: '' }

    //Injetar EmailService
    constructor(private emailService: EmailService){}

    get isNewEmailFormOpen() {
        return this._isNewEmailFormOpen;
    }

    toggleNewEmailForm() {
        this._isNewEmailFormOpen = !this.isNewEmailFormOpen
    }

    handleNewEmail(formEmail: NgForm) {
        if (formEmail.invalid) return;

        this.emailService
            .enviar(this.email)
            .subscribe(
                emailApi => {
                    //Fazemos todas as outras operações após o OK da API
                    this.emailList.push(emailApi)
                    this.email = {destinatario: '', assunto: '', conteudo: ''}
                    formEmail.reset();
                },
                erro => console.error(erro)
            )
    }
}
```

4. **Extra:** exiba uma mensagem de erro caso aconteça algum problema com a API.
5. **Extra:** substitua o endereço da API por uma variável de ambiente usando o arquivo `environment.ts`.

6. **Extra:** crie um serviço que faz uma interface para o armazenamento e recuperação do token.

EXERCÍCIO: LISTANDO COMPONENTES PARA OS EMAILS

42.1 OBJETIVO

Agora que estamos criando nosso email de verdade, podemos melhorar sua apresentação da lista de emails, deixando uma caixa de entrada mais bonita e que pode ter mais funcionalidades depois.

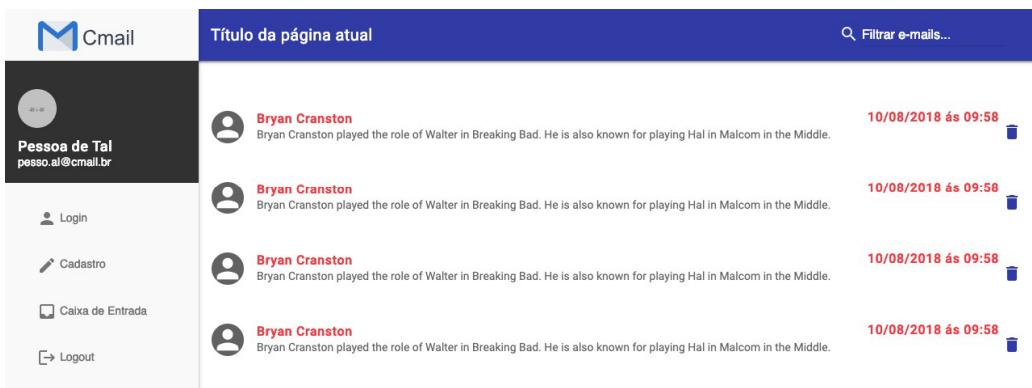


Figura 42.1: CmailListComponent na caixa de entrada

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

42.2 PASSO A PASSO COM CÓDIGO

Para que não seja necessário digitar o código dos arquivos de HTML ou CSS, baixe-os a partir do repositório dos arquivos do curso, pasta **19**:

1. Vamos criar um novo componente para receber os dados dos emails e listar eles na caixa de entrada. Será o **CmailListItemComponent**. Utilize a CLI ou a extensão Angular Schematics para gerar este componente. Por exemplo:

```
ng g c components/cmailListItem
```

1. Perceba que a CLI já percebe qual o módulo do diretório onde foi criado o novo componente, e declara o componente neste módulo, porém ainda temos que fazer o trabalho de informar que ele deve ser exportado. Informe **CmailListItemComponent** em exports de **SharedComponentsModule**:

```
#./src/app/components/shared-components.module.ts
```

```
@NgModule({
  declarations: [ HeaderComponent, CmailListItemComponent], //CLI já declarou
  imports: [ CommonModule ],
  exports: [ HeaderComponent, CmailListItemComponent ] //Exportar CmailListItemComponent
})
```

2. Dos arquivos do curso, pegue o arquivo de estilos CSS, e de template (HTML) para o **CmailListItemComponent**:

- i. 19/src/app/components/cmail-list-item/cmail-list-item.component.css
- ii. 19/src/app/components/cmail-list-item/cmail-list-item.component.html

3. No template de **CaixaDeEntradaComponent**, apenas teste se **CmailListItemComponent** está sendo exibido corretamente.

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.html
```

```
<cmail-list-item></cmail-list-item>
<cmail-list-item></cmail-list-item>
<cmail-list-item></cmail-list-item>
```

No próximo capítulo vamos fazer **CmailListItemComponent** receber dados de um email e ser listado.

EXERCÍCIO: APRIMORANDO O COMPONENTE CMAIL-LIST-ITEM

43.1 OBJETIVO

Vamos exibir os emails que enviamos usando o CmailList Item, e também adicionar mais dois dados, o de data de envio e introdução do conteúdo.



Figura 43.1: cmail-list-item preenchido

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

43.2 PASSO A PASSO COM CÓDIGO

1. Crie na classe de **CmailListItemComponent**, as propriedades "input" que precisamos exibir no template:

```
#./src/app/components/cmail-list-item/cmail-list-item.component.ts
```

```
//código anterior omitido
export class CmailListItemComponent implements OnInit {

  @Input() destinatario = '';
  @Input() assunto = '';
  @Input() introducaoDoConteudo = '';
  @Input() dataDeEnvio = '';

  constructor() {}

  ngOnInit() {}
}
```

2. Faça bind dessas propriedades no template de **CmailListItemComponent**:

```
#./src/app/components/cmail-list-item/cmail-list-item.component.html
```

```
<!-- Email Item -->
<li class="emailList__item mdl-list__item mdl-list__item--three-line" tabindex="0">
  <span class="mdl-list__item-primary-content">
    <a class="emailList__itemLink">
      <i class="material-icons mdl-list__item-avatar">person</i>
      <span>
        {{destinatario}}
      </span>
      <span class="emailList__itemText mdl-list__item-text-body">
        {{assunto}}: {{introducaoDoConteudo}}
      </span>
    </a>
  </span>
  <span class="mdl-list__item-secondary-content">
    <a class="emailList__itemLink" tabindex="-1">
      <span class="emailList__itemDate">
        {{dataDeEnvio}}
      </span>
    </a>
  </span>

  <span class="emailList__itemMenu">
    <button class="mdl-button mdl-js-button mdl-button--icon mdl-button--colored">
      <i class="material-icons">delete</i>
    </button>
  </span>
</li><!-- ./Email Item -->
```

3. No template de **CaixaDeEntradaComponent**, liste `<cmail-list-item>` dentro das ``'s, preenchendo os 4 atributos atributos esperados:

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.html
```

```
<div class="mdl-grid">
  <ul>
    <li *ngFor="let email of emailList">
```

```

<cmail-list-item
  [destinatario]="email.destinatario"
  [assunto]="email.assunto"
  [introducaoDoConteudo]="email.introducaoDoConteudo"
  [dataDeEnvio]="email.dataDeEnvio"
></cmail-list-item>
</li>
</ul>
</div>

```

Porém ao fazer isto, percebemos um problema, que não temos os dados de **data de envio** e **introdução do conteúdo** no modelo de Email, nos impedindo de exibir estas informações. Ao verificar os dados retornados da API, é possível verificar a seguinte estrutura:

```
{
  "id": "6ca85c2a-1dd9-45dd-8348-b0499e031931",
  "from": "bob@cmail.com",
  "to": "sandy@cmail.com",
  "subject": "Partiu Ibira?",
  "content": "Vai ter uma sessão no planetário, vamos?",
  "created_at": "2019-01-07T03:14:20.507Z",
  "updated_at": "2019-01-07T03:14:20.507Z"
}
```

- Vamos adicionar no modelo de **Email**, uma propriedade chamada **dataDeEnvio**, e outra **introducaoDoConteudo**:

`#./src/app/models/email.ts`

```

export class Email {

  destinatario = '';
  assunto = '';
  conteudo = '';
  dataDeEnvio = '';

  constructor (
    { destinatario, assunto, conteudo, dataDeEnvio}:
      { destinatario: string, assunto: string, conteudo: string, dataDeEnvio: string }
  ){
    this.destinatario = destinatario;
    this.assunto = assunto;
    this.conteudo = conteudo;
    this.dataDeEnvio = dataDeEnvio;
  }

  get introducaoDoConteudo() {
    return this.conteudo.substr(0, 140) + '...'
  }
}

```

- Alterando o construtor da classe modelo de **Email**, quebra imediatamente a instância dela em **EmailService**, que agora espera receber em sua construção o dado da data de envio. Vamos fazer este ajuste:

`#./src/app/services/email.service.ts`

```
//código anterior omitido
(emailApi: any) => {
  return new Email({
    destinatario: emailApi.to,
    assunto: emailApi.subject,
    conteudo: emailApi.content,
    dataDeEnvio: emailApi.created_at
  })
}
//código posterior omitido
```

6. **Extra:** Um pequeno ajuste no estilo da lista, poderá deixar as coisas um pouco mais bonitas ;)

`#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.ts`

```
styles: [`
  ul, li {
    margin: 0;
    padding: 0;
    list-style-type: none;
  }
`]
```

EXERCÍCIO: PEGANDO OS EMAILS DO SERVIDOR

44.1 OBJETIVO

Já conseguimos ver o email que enviamos na lista, porém e todos os outros já testamos e enviamos antes? Vamos carregá-los da API no momento de acessar a caixa de entrada! Para isto precisamos adicionar o método `listar` em `EmailService`.

The screenshot shows a web-based email application interface. On the left, there's a sidebar with a user profile picture, the name "Pessoa de Tal", and the email "pesso.al@cmail.br". Below the profile are links for "Login", "Cadastro", "Caixa de Entrada", and "Logout". The main area has a header "Título da página atual" and a search bar "Filtrar e-mails...". The inbox lists six messages:

From	Subject	Date	Action
bobe@cmail.com.br	Lorem ipsum dolor sit amet consectetur adipisicing elit....	2019-01-16T22:50:54.424Z	
bobe@cmail.com.br	Oba: Lorem ipsum dolor sit amet consectetur adipisicing elit....	2019-01-16T23:05:59.053Z	
bob@cmail.br	Vamos no parque?: asdad...	2019-01-17T00:02:43.210Z	
bob@cmail.br	Vamos no parque?: dasdasda...	2019-01-17T00:04:00.471Z	
patrick@cmail.com.br	Ibirá domingo?: Que tal?...	2019-01-17T14:13:08.893Z	
patrick@cmail.com	Ibirá domingo?: que tal?...	2019-01-17T14:14:23.165Z	

Figura 44.1: Lista de emails da API na caixa de entrada

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

44.2 PASSO A PASSO COM CÓDIGO

1. Implemente o método `listar` em **EmailService**. Uma atenção deve ser considerada, é que só vamos listar os emails da pessoa autenticada, portanto vamos passar o headers de autenticação. Devemos retonar uma lista de Emails:

```
#./src/app/services/email.service.ts
```

```
//codigo anterior omitido
listar(){
    return this.http
        .get(this.api, { headers: this.cabecalho })
        .pipe<Email[]>(
            map(
                (response: any[]) => {
                    return response
                        .map(
                            emailApi => new Email({
                                destinatario: emailApi.to,
                                assunto: emailApi.subject,
                                conteudo: emailApi.content,
                                dataDeEnvio: emailApi.created_at
                            })
                )
            )
        )
}
```

2. Agora vamos chamar o método `listar` assim que **CaixaDeEntradaComponent** for inicializado, portanto implementar a interface `OnInit`, e no método `NgOnInit`, chamamos `listar` de **EmailService**:

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.ts
```

```
ngOnInit(){
    this.emailService
        .listar()
        .subscribe(
            lista => {
                this.emailList = lista;
            }
}
```

3. **Extra:** faça um callback de erro caso aconteça algum problema com a API no momento de listar os emails.
4. **Extra:** tem código repetido em **EmailService**, como podemos melhorar isto?

TRANSFORMANDO DADOS COM PIPES

Toda aplicação começa com o que parece ser uma tarefa simples: obter dados, transformá-los e mostrá-los aos usuários. Obter dados pode ser tão simples quanto criar uma variável local ou tão complexo quanto transmitir dados por um WebSocket.

Quando os dados chegam, podemos enviar seus valores `toString` diretamente para a `view`, mas isso raramente contribui para uma boa experiência do usuário. Por exemplo, na maioria dos casos de uso, os usuários preferem ver uma data em um formato simples, como `15 de abril de 1988`, em vez do formato bruto de um `timestamp` como `Sex Abr 15 1988 00:00:00 GMT-0700 (Horário de verão do Pacífico)`.

Claramente, alguns valores como datas, moedas etc, se beneficiam de um pouco de edição. E percebemos que possivelmente teremos que aplicar estas transformações em vários locais, na mesma forma e repetidamente, dentro e entre muitas aplicações. E podemos dizer que estas transformações nestes valores `string` seriam como estilo de apresentação, e que poderíamos aplicá-las como usamos o CSS no HTML, através de indicações nas tags.

Vamos conhecer então os **pipes** do Angular, que é uma maneira de transformar a exibição dos dados diretamente nos *templates* HTML. [1]

45.1 USANDO PIPES

Um *pipe* recebe como entrada os dados à sua esquerda, e os transforma em uma saída desejada. Abaixo, um exemplo de um *pipe* para transformar a propriedade de `dataEnvio` com uma exibição de uma data humanamente mais compreensível:

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'cmail-email',
  template: `
    <article>
      <p>{{destinatario}}</p>
      <!-- pipe date -->
      <p>{{dataEnvio | date}}</p>
      <p>{{assunto}}</p>
      <p>{{conteudo}}</p>
    </article>
  `
```

```

})
export class EmailComponent {
  @Input() assunto = "";
  @Input() destinatario = "";
  @Input() conteudo = "";
  dataEnvio = new Date(2018, 3, 15); // April 15, 2018
}

```

Focando aqui apenas no uso do *pipe* no template:

```
<p>{{dataEnvio | date}}</p>
```

Dentro da expressão de interpolação `{{}}`, você flui o valor de `dataEnvio` do componente através do operador `pipe (|)` para a função `pipe date` à sua direita. Todos os `pipe` funcionam dessa maneira.



Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

45.2 BUILT-IN PIPES

Angular vem com um "estoque" de pipes prontinhos pra uso! Como `DatePipe` , `UpperCasePipe` , `LowerCasePipe` , `CurrencyPipe` e `PercentPipe` . Estão todos disponíveis para uso em qualquer `template`, pois são disponibilizados pelo `CommonModule` .

Leia mais sobre estes e muitos outros *pipes* "de fábrica" nos tópicos de *pipes* na documentação [2].

Obs: Angular não possui um `FilterPipe` ou um `OrderByPipe` pelos motivos que serão explicados nos próximos capítulos.

45.3 PARAMETRIZANDO UM PIPE: SLICEPIPE

Um *pipe* pode aceitar muitos parâmetros opcionais para ajustar sua transformação final. Para

adicionar parâmetros a um *pipe*, após o nome do *pipe* adicione dois pontos (:) e, em seguida, o valor do parâmetro (como por exemplo a moeda: 'EUR' no caso do `currencyPipe`).

Vamos modificar o template de `dataEnvio` para atribuir ao `DatePipe` um parâmetro de formato para exibir horas e minutos, e dia, mês e ano:

```
<p>{{dataEnvio | date:'HH:mm dd/MM/yyyy'}}</p>
```

O *pipe* aceita vários parâmetros, separe os valores de cada parâmetro com dois pontos (como `slice:1:5`). Exemplo:

```
{{conteudo | slice:0:140 }}
```

Referências

1. <https://angular.io/guide/pipes>
2. <https://angular.io/api?type=pipe>

EXERCÍCIO: TRANSFORMANDO DADOS COM PIPES

46.1 OBJETIVO

Nossa caixa de entrada está linda, exceto pelo fato do formato das datas... Estamos vendo algo como `2019-01-16T22:50:54.424Z`, ou seja, um timestamp... Como poderíamos facilmente tratar este dado no formato que desejarmos? Por exemplo: `10/01/2019` ?

Fácil! Vamos usar os pipes do Angular!



Figura 46.1: Datas no formato brasileiro

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

46.2 PASSO A PASSO COM CÓDIGO

1. Para apresentar a data de maneira mais aceitável podemos passar depois da propriedade `dataDeEnvio`, porém ainda dentro da Angular Expression, o caractere `pipe |` que indica no Angular que você vai usar uma função de pipe, que transforma aquele dado de alguma maneira; e logo em seguida usamos a função `date`, que formada a data pelo nome do mês e dia e ano:

```
#./src/app/components/cmail-list-item/cmail-list-item.component.html
```

```
<span class="mdl-list__item-secondary-content">
  <a class="emailList__itemLink" tabindex="-1">
    <span class="emailList__itemDate">
      {{dataDeEnvio | date}}
    </span>
  </a>
</span>
```

2. Podemos ver a documentação e experimentar outros formatos para a data:

```
https://angular.io/api/common/DatePipe
```

```
#./src/app/components/cmail-list-item/cmail-list-item.component.html
```

```
<span class="mdl-list__item-secondary-content">
  <a class="emailList__itemLink" tabindex="-1">
    <span class="emailList__itemDate">
      {{dataDeEnvio | date: 'dd/MM/yyyy' }}
    </span>
  </a>
</span>
```

EMISSÃO DE EVENTOS

Para um componente "pai" obter valores ou algum "sinal" que alguma coisa aconteceu dentro de um componente "filho", este deve ter um mecanismo de escuta de um evento específico que deve ser emitido pelo componente filho. Para isto, usamos a mesma lógica de eventos do JavaScript, onde podemos criar nossos próprios eventos personalizados para enviar valores de um determinado componente filho para seu escopo superior (componente pai). [1]

47.1 EVENTOS CUSTOMIZADOS

No componente `<cmail-list-item>`, quando clicamos no ícone de lixeira, queremos que apague aquele determinado email. Porém, os dados da lista de email está em `CaixaDeEntradaComponent`. Não seria responsabilidade de cada instância de `ListItem` ter acesso ao `EmailService` e uma cópia de toda a lista de emails. Isto poderia fazer um uso alto da memória do navegador, causando lentidão e outros problemas.

Portanto quando alguém clicar no ícone da lixeira de um determinado `<cmail-list-item>`, este deverá disparar um evento que sinalizará `CaixaDeEntradaComponent` que aquele email deve ser excluído da API.

Portanto, a primeira coisa a se fazer é usar um evento comum do JavaScript para capturar o click do usuário no ícone da lixeira:

```
#./src/app/components/cmail-list-item/cmail-list-item.component.html
```

```
<button (click)="apagarEmail()">
  apagar
</button>
```

Criando um evento com `@Output` e `EventEmitter`

No método `apagarEmail` do componente devemos emitir um evento para que o componente pai possa capturar este momento. Este evento será personalizado, será um evento criado por nós e emitido apenas por este tipo de componente. Será o evento `clicouNaLixeira`.

Para criar um evento devemos criar uma propriedade da classe com este nome e decorado com `@Output`.

Esta propriedade deve receber uma instância da classe `EventEmitter` do pacote `'@angular/core'`

```
#./src/app/components/cmail-list-item/cmail-list-item.component.ts

import { Component, Input, EventEmitter, Output } from '@angular/core';

@Component({
  selector: 'cmail-list-item',
  templateUrl: './cmail-list-item.component.html'
})
export class CmailListItemComponent {

  @Input() destinatario = '';
  @Input() assunto = '';
  @Input() conteudo = '';
  @Input() dataEnvio = '';
  @Output() clicouNaLixeira = new EventEmitter();

  constructor(){}

  apagarEmail(){
    console.log(`clicou na lixeira`);
  }
}
```

Emissão do evento

Agora para que o método `apagarEmail` realmente faça o disparo do evento ao ser executado pelo click no botão da lixeira, devemos usar o método `emit` de `EventEmitter`:

```
apagarEmail(){
  this.clicouNaLixeira.emit();
}
```

Caso seja necessário, é possível enviar dados pelo método `emit`, veja as possibilidades:

```
apagarEmail(){
  this.clicouNaLixeira.emit(`Apaga o email aí!`);
}
```

Ou:

```
apagarEmail(){
  this.clicouNaLixeira.emit({remove: true});
}
```

Ou:

```
apagarEmail(){
  this.clicouNaLixeira.emit(true);
}
```

Escuta do evento

Para o componente pai, agora na instância de `<cmail-list-item>`, temos o evento recém criado que pode ser escutado usando a expressão de eventos (`clicouNaLixeira`) e associado com um

método do componente pai, no exemplo abaixo o `handleRemoveEmail` :

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.html
```

```
<cmail-list-item  
  [destinatario]="email.destinatario"  
  [assunto]="email.assunto"  
  [introducaoDoConteudo]="email.introducaoDoConteudo"  
  [dataDeEnvio]="email.dataDeEnvio"  
  (clicouNaLixeira)="handleRemoveEmail($event, email.id)"  
></cmail-list-item>
```

Através de `$event` conseguimos acessar os dados enviados pelo método `emit` disparado no componente filho.

E finalmente, um exemplo de como seria o método do componente pai:

```
handleRemoveEmail(algumaMensagem, emailId){  
  this.emailService  
    .deletar(emailId)  
    .subscribe(  
      () => this.listarEmails()  
      ,erro => console.log(erro)  
    )  
}
```

Referências

1. <https://angular.io/guide/component-interaction#parent-listens-for-child-event>
2. <https://angular.io/guide/template-syntax#custom-events-with-eventemitter>

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

EXERCÍCIO: APAGANDO EMAILS DA API COM EVENTEMITTER

48.1 OBJETIVO

Uma das funcionalidades que queremos ter no **CmailListItemComponent** é a opção de remover o item da lista, no caso o email que teve o seu ícone de remoção clicado. Vamos utilizar o EventEmitter para passar dados de um elemento filho para o elemento pai. Depois teremos que apagar da API os dados do email clicado, passando o id do email e o token do usuário. Por fim teremos que fazer um tratamento na lista de emails de **CaixaDeEntradaComponent** para remover o email recém apagado na API.

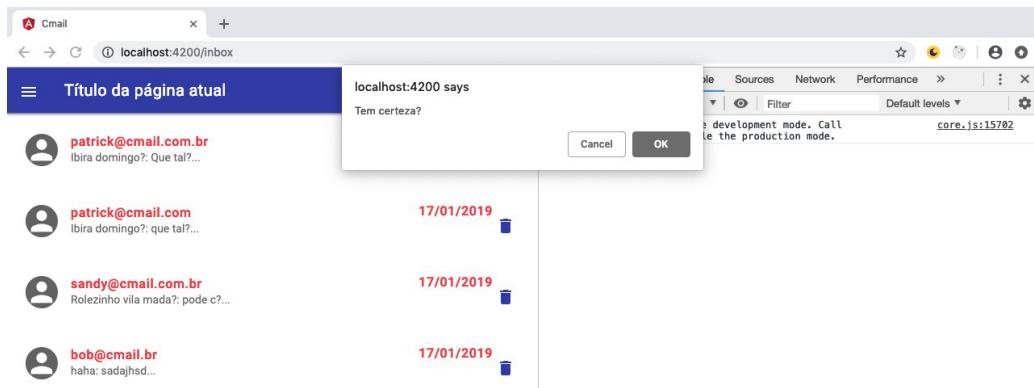


Figura 48.1: Confirma que quer apagar?

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

48.2 PASSO A PASSO COM CÓDIGO

1. Primeiro, vamos adicionar um evento `(click)`, associado a um método `handleRemoveEmail`, para poder remover o email, na instância de `<cmail-list-item>` em **CaixaDeEntradaComponent**:

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.html
```

```
<cmail-list-item  
[destinatario] = "email.destinatario"  
[assunto] = "email.assunto"  
[introducaoDoConteudo] = "email.introducaoDoConteudo"  
[dataDeEnvio] = "email.dataDeEnvio"  
(click) = "handleRemoveEmail($event)"  
></cmail-list-item>
```

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.ts
```

```
handleRemoveEmail(click: Event){  
  console.log('Clicou no item!');  
}
```

Neste momento o código já funciona, porém o evento acontece toda vez que clicamos em qualquer parte do cartão. No Angular as ações sempre fluem do componente maior para os mais internos e não o contrário. Vamos dar um jeito nisso!

2. O componente **CmailListItemComponent**, no botão de remover, também vai precisar ter um evento `(click)` associado a um método da classe, então criaremos o `removeEmail` :

```
#./src/app/components/cmail-list-item/cmail-list-item.component.ts
```

```

removeEmail(click: Event){
  console.log('Clicou no botão remover!')
}

#./src/app/components/cmail-list-item/cmail-list-item.component.html

<span class="emailList__itemMenu">
  <button (click)="removeEmail($event)" class="mdl-button mdl-js-button mdl-button--icon mdl-button--colored">
    <i class="material-icons">delete</i>
  </button>
</span>

```

O problema é que agora não conseguimos notificar a função do *controller*, de **CaixaDeEntradaComponent**, pois ela que conseguirá se comunicar com o **EmailService** e remover da API os dados de um email.

Para que seja possível passar o status de que foi clicado no ícone de remover para o **CaixaDeEntradaComponent**, vamos fazer com que o método `removeEmail` de **CmailListItemComponent** emita uma mensagem para o componente acima, através da **emissão de um evento**, usando o *decorator* `@Output` ;

- Em **CmailListItemComponent** adicione a propriedade `vaiRemover` que fará a saída do evento através do decorador `@Output('eventoVaiRemover')` , para isto esta propriedade deve ser uma instância de **EventEmitter** :

```
#./src/app/components/cmail-list-item/cmail-list-item.component.ts
```

```

//código anterior omitido
export class CmailListItemComponent implements OnInit {

  @Input() destinatario = '';
  @Input() assunto = '';
  @Input() introducaoDoConteudo = '';
  @Input() dataDeEnvio = '';
  //Nova propriedade
  @Output('eventoVaiRemover') vaiRemover = new EventEmitter()

  constructor() {}

  ngOnInit() {}

  removeEmail(click: Event){
    console.log('Clicou no botão remover!')
  }
}

```

- O método `removeEmail` , ainda em **CmailListItemComponent**, agora emitirá um evento após o usuário confirmar:

```
#./src/app/components/cmail-list-item/cmail-list-item.component.ts
```

```

removeEmail(click: Event){
  console.log('Clicou no botão remover!')
  //Emite eventoVaiRemover e ainda manda um status
  if(confirm('Tem certeza?')) {
    this.vaiRemover.emit({ status: 'removing' })
  }
}

```

5. Agora, voltando ao **CaixaDeEntradaComponent**, ao invés de escutarmos o evento click, vamos escutar o evento `eventoVaiRemover`:

`#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.html`

```

<cmail-list-item
  [destinatario]="email.destinatario"
  [assunto]="email.assunto"
  [introducaoDoConteudo]="email.introducaoDoConteudo"
  [dataDeEnvio]="email.dataDeEnvio"
  (eventoVaiRemover)="handleRemoveEmail($event)"
></cmail-list-item>

```

`#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.ts`

```

handleRemoveEmail(eventoVaiRemover, emailId){
  console.log(eventoVaiRemover);
  if (eventoVaiRemover.status === 'removing'){
    //O próximo passo é apagar da API! :)
  }
}

```

6. Para apagar um email na API precisamos passar o **id do email** e o **token do usuário**, portanto vamos alterar o modelo de `Email` para que possamos pegar o dado do ID da API para nossa aplicação:

`#./src/app/models/email.ts`

```

export class Email {

  destinatario = '';
  assunto = '';
  conteudo = '';
  dataDeEnvio = '';
  id = ''; //Propriedade id!

  constructor (
    { destinatario, assunto, conteudo, dataDeEnvio, id}: //id aqui
    { destinatario: string, assunto: string, conteudo: string, dataDeEnvio: string, id: string }
  //tipo do id aqui
  ){
    this.destinatario = destinatario;
    this.assunto = assunto;
    this.conteudo = conteudo;
    this.dataDeEnvio = dataDeEnvio;
    this.id = id; //id aqui
  }

  get introducaoDoConteudo() {
    return this.conteudo.substr(0, 140) + '...'
  }
}

```

```
    }
}
```

7. Em **EmailService** precisamos **mapear o id** vindo da API no **retorno dos métodos enviar e listar**. Os retornos devem ficar parecidos com isso:

```
#./src/app/services/email.service.ts
```

```
return new Email({
  destinatario: emailApi.to,
  assunto: emailApi.subject,
  conteudo: emailApi.content,
  dataDeEnvio: emailApi.created_at,
  id: emailApi.id //mapeando o id vindo da API
})
```

8. Agora vamos implementar o método **deletar** em **EmailService**:

```
#./src/app/services/email.service.ts
```

```
deletar(id){
  return this
    .http
    .delete(` ${this.api}/${id}` , { headers: this.cabecalho})
}
```

9. Por fim, voltando ao **CaixaDeEntradaComponent**, no método **handleRemoveEmail** chamamos o método **deletar** do serviço de email, passando o **id** do email clicado:

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.html
```

```
<cmail-list-item
  [destinatario]="email.destinatario"
  [assunto]="email.assunto"
  [introducaoDoConteudo]="email.introducaoDoConteudo"
  [dataDeEnvio]="email.dataDeEnvio"
  (eventoVaiRemover)="handleRemoveEmail($event, email.id)" <!-- id como segundo parâmetro -->
></cmail-list-item>
```

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.ts
```

```
//recebe emailID no segundo parâmetro
handleRemoveEmail(eventoVaiRemover, emailId){
  console.log(eventoVaiRemover);
  if (eventoVaiRemover.status === 'removing'){
    this.emailService
      .deletar(emailId)
      .subscribe(
        res => {
          console.log(res);

          //remove o email da lista de emails depois dela ser apagada da API
          this.emailList = this.emailList.filter(email => email.id != emailId);
        }
        ,err => console.error(err)
      )
  }
}
```

}

BROADCASTING

Quando precisamos transmitir dados para vários componentes da aplicação, sendo estes não tendo necessariamente uma relação de pai/filho, apenas usar a emissão de eventos não ajuda, e na verdade até deixa mais complexo.

Para resolver este cenário complexo, existe um conceito de transmissão de dados, que é possível de ser implementado no Angular com uma associação do conceito de serviços e com os recursos da biblioteca **RxJS**.

49.1 TRANSMISSOR

A primeira coisa que teremos que fazer é um serviço que é possível fazer a atualização de dados e depois transmitir para todos aqueles que necessitam receber estes dados, ou seja, os assinantes.

Para exemplificar, vamos criar o `PageDataService`, que será um serviço para atualizar dados da página, como o título da aba do navegador e do `HeaderComponent`, utilizado por várias páginas.

O serviço conterá um método `atualizaTitulo`, que servirá para atualizar o título das páginas.

E também conterá uma propriedade `titulo`, que será a fonte do dado assinada por todos os assinantes, e que cada componente assinante poderá realizar a operação que quiser com este dado.

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class PageDataService {

  readonly titulo = new Subject<string>();

  atualizaTitulo(novoTitulo: string){
    document.title = `${novoTitulo} - CMail`;
    this.titulo.next(novoTitulo);
  }
}
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

49.2 SUBJECT

Um `Subject` na RxJS é um tipo especial de `Observable` que permite que os valores sejam convertidos em `multicast` para muitos `Observers` (assinantes). Enquanto `Observable`s comuns são `unicast` (cada `Observer` inscrito possui uma execução independente do `Observable`), os `Subject`s são `multicast`.

49.3 TRANSMITINDO DADOS

Para todos componentes ou diretivas que necessitarão atualizar os dados do título de uma página, estas deverão injetar o serviço como uma dependência, e depois, neste caso, no método `ngOnInit` é acionado o método `atualizaTitulo` de `PageDataService`:

```
constructor(private pageData: PageDataService) { }

ngOnInit() {
  this.pageData.atualizaTitulo('Caixa de Entrada');
}
```

49.4 ASSINANDO DADOS

Para o `HeaderComponent`, também devemos injetar como uma dependência o `PageDataService`, e depois em seu construtor assinamos o `título` usando o método `subscribe`:

```
import { Component } from '@angular/core';
import { PageDataService } from 'src/app/services/page-data.service';

@Component({
  selector: 'cmail-header',
  templateUrl: './header.component.html',
```

```

        styleUrls: ['./header.component.css', './header-search.css']
    })
export class HeaderComponent {

    isMenuOpen = false;
    tituloHeader = '';

    constructor(private pageService: PageDataService){
        this.pageService
            .titulo
            .subscribe(
                (novoTitulo) => {
                    this.tituloHeader = novoTitulo;
                }
            )
    }

    exibeMenu(){
        this.is
        MenuOpen = !this.isMenuOpen;
    }
}

```

Em detalhe:

```

constructor(private pageService: PageDataService){
    this.pageService
        .titulo
        .subscribe(
            (novoTitulo) => {
                this.tituloHeader = novoTitulo;
            }
        )
}

```

Referências

1. <https://rxjs-dev.firebaseio.com/guide/subject>

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e

Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

EXERCÍCIO: BROADCAST DO TITULO DA PÁGINA

50.1 OBJETIVO

Em todas nossas páginas temos que definir um título, aqui no CMail, pelo menos no HeaderComponent e no título da index.html também. Porém como poderíamos definir isto dinamicamente? Vamos criar um serviço para receber os dados que podem mudar de página em página, e estas variáveis que mudam o tempo todo, assinam nosso serviço, aguardando sempre por atualizações. Isto é possível associando o serviço com um **Subject do RxJS!**

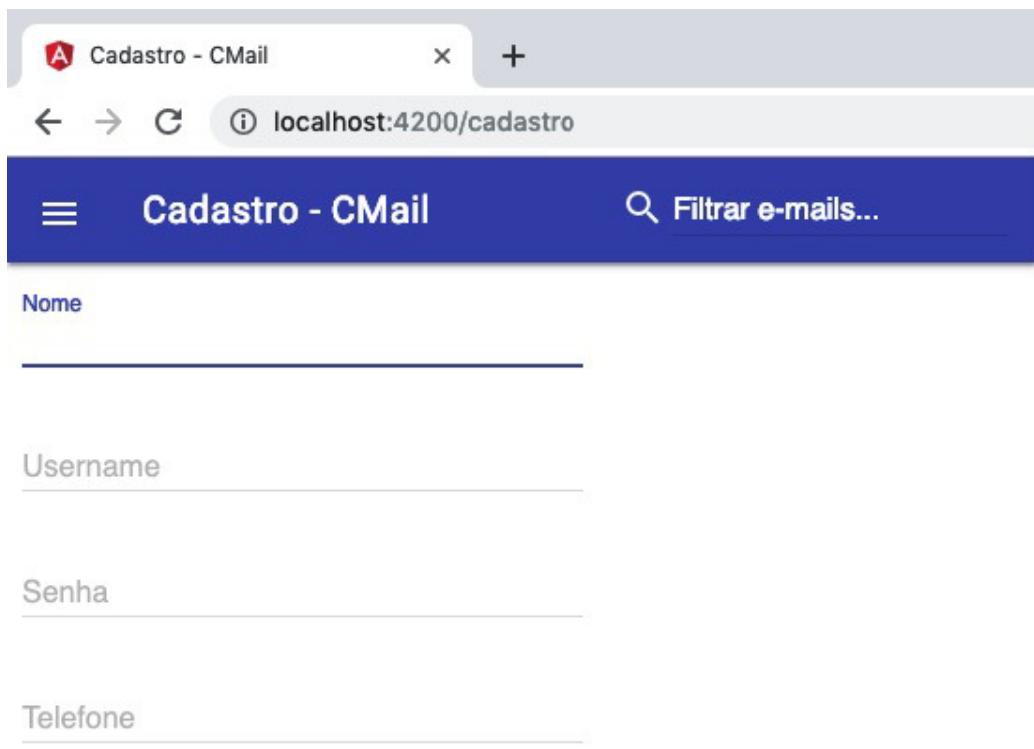


Figura 50.2: Cadastro

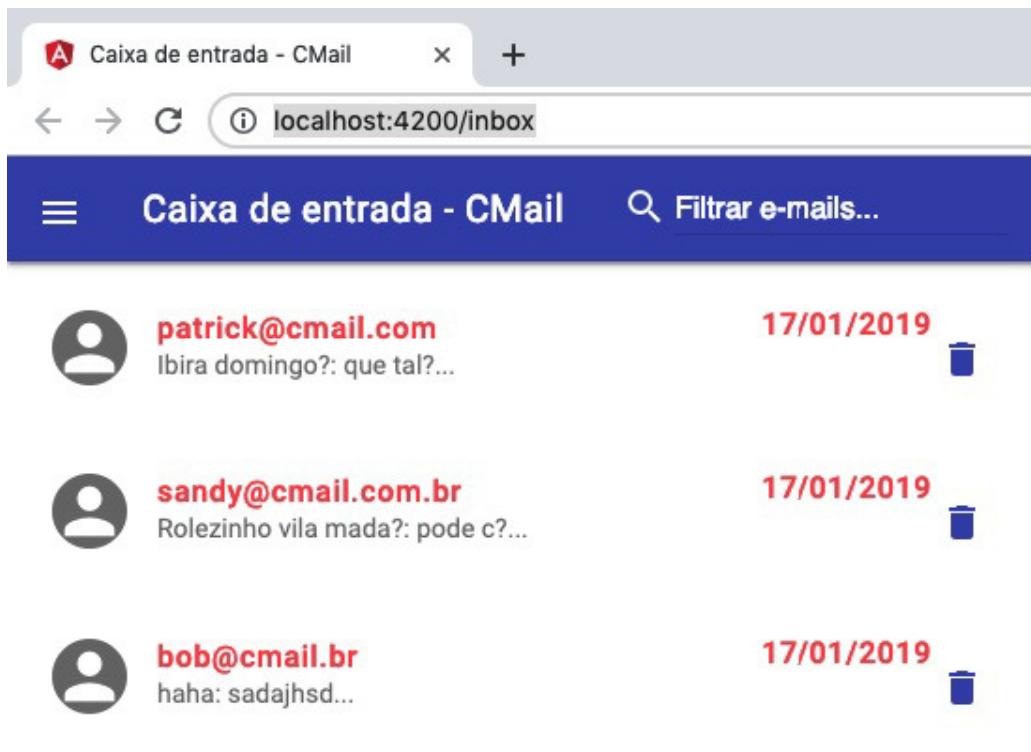


Figura 50.1: Caixa de entrada

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

50.2 PASSO A PASSO COM CÓDIGO

1. Vamos criar a classe **PageDataService** que servirá toda a aplicação, oferecendo métodos para atualizar as variáveis comuns da página, como o título. Faremos nela um atributo `titulo` que será um **Subject** que receberá uma `string` e será o responsável por notificar todos os seus ouvintes sobre a mudança do título da página atual. Também faremos o método `defineTitulo` que receberá uma `string` com um novo valor para o `titulo`, e sempre que for chamado, atualizará o `titulo` da

página HTML e notificará quem se inscrever na propriedade `titulo` avisando sobre a ocorrência da mudança!

`#./src/app/services/page.service.ts`

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class PageDataService {

  titulo = new Subject<string>();

  defineTitulo(novoTitulo: string) {
    document.querySelector('title').textContent = novoTitulo;
    this.titulo.next(novoTitulo);
  }
}
```

2. Em **HeaderComponent**, injetamos no construtor `PageDataService`, e ainda no construtor assinamos o seu atributo `titulo` usando o método `subscribe`, presente no tipo `Subject`. O *callback* de `subscribe` será a ação que vamos tomar neste componente quando a notificação for disparada para o `titulo`, recebendo como parâmetro um valor novo para o `titulo` de **HeaderComponent**. Então, com este novo valor, o passamos para o atributo `tituloDaPagina`.

`#./src/app/components/header/header.component.ts`

```
// Código anterior omitido.
export class HeaderComponent {

  // Restante do código omitido.

  tituloDaPagina = 'CMail'; // Nova propriedade.

  // Injeção de PageDataService.
  constructor(private pageService: PageDataService){

    // Assinando titulo de PageDataService.
    this.pageService
      .titulo
      .subscribe(novoTitulo => this.tituloDaPagina = novoTitulo);
  }

  // Restante do código omitido.

}
```

3. No template de **HeaderComponent**, fazemos bind da propriedade `tituloDaPagina` da classe:

`#./src/app/components/header/header.component.html`

```
<span class="headerGlobal__title mdl-layout-title">{{tituloDaPagina}}</span>
```

4. Agora em **CaixaDeEntradaComponent**, injetamos **PageDataService** no construtor, e no em **ngOnInit** , chamamos o método **defineTitulo** de **pageDataService** , passando a string com o título desta página.

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.ts

// Código anterior omitido.

// Injetamos PageDataService.
constructor(private emailService: EmailService,
            private pageDataService: PageDataService) {}

ngOnInit(){
  this.emailService
    .listar()
    .subscribe(
      (lista) => {
        this.emailList = lista;
      });
}

// Definimos o titulo da página.
this.pageDataService
  .defineTitulo('Caixa de entrada - CMail');
}
// Código posterior omitido.
```

5. **Extra:** faça o mesmo para **LoginComponent** e **CadastroComponent**!

CRIANDO SEU PRÓPRIO *PIPE*

Os *pipes* são uma excelente opção para padronizar transformações de dados a nível de template. Apesar do Angular fornecer um conjunto de *pipes* prontos de "fábrica" para usarmos, nem sempre eles atendem todas as situações, portanto o framework possibilita que possamos extender esta funcionalidade e criamos nossos próprios *pipes*. [1]

51.1 DECORATOR PIPE

Para construirmos um *pipe*, criamos em um arquivo com o sufixo `.pipe` uma classe com o nome do *pipe* [2]. Como exemplo, vamos criar um *pipe* que verifica se o destinatário do email e com base nisso adiciona um marcador.

Para que a classe TypeScript se torne de fato um *pipe*, adicionamos o decorador `Pipe()`. Este receberá como valor um objeto, que e com ele definiremos o nome do *pipe* no template com a propriedade `name` :

```
import { Pipe } from '@angular/core';

@Pipe({
  name: 'marcador'
})
export class MarcadorPipe {
```

Interface `PipeTransform`

Agora para desenvolvemos o a classe corretamente, devemos implementar a interface `PipeTransform` pois ela que nos indica que devemos ter um método `transform` , que é executado quando o *pipe* entra em ação

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'marcador'
})
export class MarcadorPipe implements PipeTransform {
  transform() {
```

Usando método transform

O método `transform` é o que recebe os parâmetros do `pipe`, executa a lógica com os parâmetros recebidos e retorna o valor com a devida transformação. O primeiro parâmetro do `pipe` é aquele no qual o `pipe` é adicionado no `template` HTML, no caso deste exemplo, iremos usar o `pipe` da seguinte maneira:

```
<span>{{destinatario | marcador }}: {{assunto}}</span>
```

Portanto o primeiro parâmetro de marcador é o dado de `destinatario`. Caso haja mais parâmetros no `pipe`, os mesmos seguirão na sequência dos parâmetros do método `transform`.

Caso o método `transform` não tenha retorno, nenhum valor será exibido, é como se `transform` fosse um filtro, e nenhuma condição fosse dada para que o filtro retornasse um valor, portanto tudo estaria "trancado" dentro deste filtro.

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'marcador'
})
export class MarcadorPipe implements PipeTransform{
  transform(destinatario: string){

    return destinatario;

  }
}
```

Agora, vamos fazer uma lógica que irá modificar o valor da `string` de destinatário. Caso a `string` incluir "chefe", modificar ela e adicionar o marcado `URGENT`, caso contiver a palavra "sandy" adicionar o marcador `friends`. Por último caso a `string` não entrar em nenhuma destas condições, devemos retornar a `string` como está para que ela continue sendo exibida.

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'marcador'
})
export class MarcadorPipe implements PipeTransform{
  transform(destinatario: string){

    if(destinatario.includes('chefe')){
      return `[URGENT] ${destinatario}`
    }

    if(destinatario.includes('sandy')){
      return `[friends] ${destinatario}`
    }

    return destinatario

  }
}
```

Declarando para usar

Como componentes e diretivas, os *pipes* também devem ser declarados em um módulo para que possam ser registrados e aptos de uso. Portanto no `declarations` do que faz sentido para este *pipe*, devemos adicionar o `MarcadorPipe`, no nosso caso, ele ficou em `SharedComponentsModule`:

```
//... imports omitidos

@NgModule({
  declarations: [
    HeaderComponent,
    FormGroupComponent,
    FormFieldDirective,
    CmailListItemComponent,
    //aqui marcador pipe!
    MarcadorPipe
  ],
  exports: [
    HeaderComponent,
    FormGroupComponent,
    FormFieldDirective,
    CmailListItemComponent
  ],
  imports: [
    CommonModule,
    RouterModule
  ]
})
export class SharedComponentsModule { }
```

Referências

1. <https://angular.io/guide/pipes#custom-pipes>
2. <https://angular.io/guide/styleguide#pipe-names>

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

EXERCÍCIO: CRIANDO UM PIPE DE FILTRO PARA EMAILS

52.1 OBJETIVO

No header do CMail, temos uma campo para filtrarmos os emails. Vamos implementar esta funcionalidade. E novamente temos que passar dados de um componente para o outro. Aqui o **HeaderComponent** deverá passar o valor do `input` de busca para o **CaixaDeEntradaComponent**. Para isto já sabemos que vamos ter que fazer um serviço, usando o `Subject`, onde a **CaixaDeEntradaComponent** irá assinar a atualização do valor do campo de filtro no header. Será o **HeaderDataService**, com isto vamos implementar um pipe customizado para filtrar os emails.

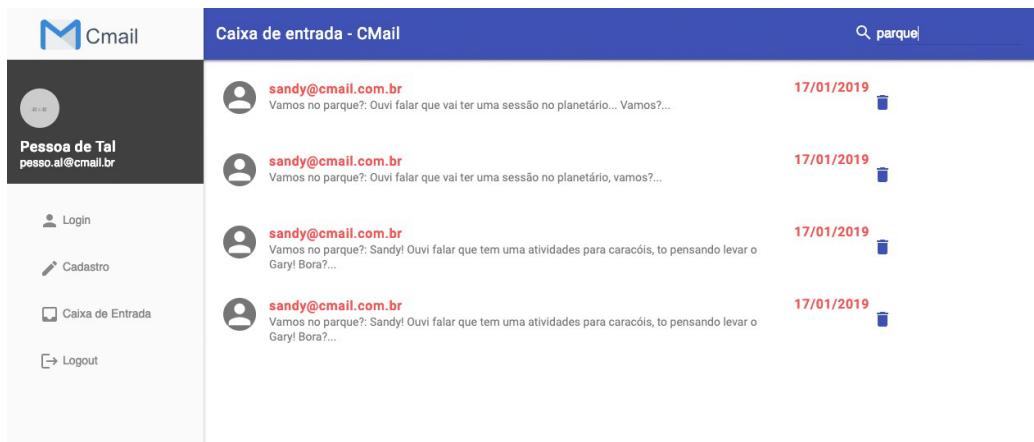


Figura 52.1: Filtro por título

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

52.2 PASSO A PASSO COM CÓDIGO

1. No template de **HeaderComponent** vamos adicionar o evento (`input`) na tag `<input>` que fará bind com um método `handleBuscaChanges` que receberá o evento como parâmetro para acessarmos o valor do input dentro do método:

```
#./src/app/components/header/header.component.html
```

```
<input  
  (input)="handleBuscaChanges($event)"  
  class="headerSearch__input mdl-textfield__input"  
  type="text" name="sample" id="fixed-header-drawer-exp"  
  placeholder="Filtrar e-mails...">
```

2. Na classe de **HeaderComponent** vamos adicionar o método `handleBuscaChanges` :

```
#./src/app/components/header/header.component.ts
```

```
handleBuscaChanges({ target }) {  
  console.log(target.value);  
}
```

3. Agora precisamos criar o **HeaderDataService** que irá mandar os dados do `input` para quem precisar receber estes dados:

```
#./src/app/services/header.service.ts
```

```
import { Injectable } from '@angular/core';  
import { Subject } from 'rxjs';
```

```

@Injectable({
  providedIn: 'root'
})
export class HeaderDataService {
  valorDoFiltro = new Subject<string>();

  constructor(){
    this.atualizarTermoDeFiltro('')
  }

  atualizarTermoDeFiltro(novoValor: string) {
    this.valorDoFiltro.next(novoValor)
  }
}

```

- Voltando ao **HeaderComponent**, vamos injetar este serviço e atualizar o método `handleBuscaChanges` para usar `atualizarTermoDeFiltro` de **HeaderDataService** para atualizar o valor do termo de busca.

`#./src/app/components/header/header.component.ts`

```

handleBuscaChanges({ target }) {
  this.headerService.atualizarTermoDeFiltro(target.value)
}

```

- Na classe de **CaixaDeEntradaComponent**, criamos a propriedade `termoParaFiltro`, que vamos passar para o pipe que iremos criar como condição do filtro. Injetamos o **HeaderDataService**, e no método `ngOnInit` assinamos o serviço para atualizar a propriedade `termoParaFiltro`:

`#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.ts`

```

this.headerService
  .valorDoFiltro
  .subscribe(novoValor => this.termoParaFiltro = novoValor)

```

Podemos verificar se o que digitamos no campo de filtro está sendo passando colocando no template de **CaixaDeEntradaComponent** o bind de `termoParaFiltro`:

`{{termoParaFiltro}}`

- Vamos criar a classe **FiltroPorAssunto**, que será o nosso pipe customizado. Na pasta `caixa-de-entrada` crie o arquivo `filtro-por-assunto.pipe.ts`. A classe **FiltroPorAssunto** que implementa a interface `PipeTransform`, que nos obriga implementar o método `transform`. Também precisamos decorar a classe com `@Pipe`, que recebe um objeto para definirmos o `name` do pipe, ou seja, como vamos chamar ele em um template.

`#./src/app/modules/caixa-de-entrada/filtro-por-assunto.pipe.ts`

```

import { Pipe, PipeTransform } from '@angular/core';
import { Email } from "../../models/email";

@Pipe({
  name: 'filtroPorAssunto'
}

```

```

})
export class FiltroPorAssunto implements PipeTransform {

  transform(listaEmails: Email[], termoFiltro: string){
    return listaEmails
      .filter(
        email =>
          email.assunto.toLowerCase()
          .includes(termoFiltro.toLowerCase()))
  }
}

```

O método `transform` recebe 1 parâmetro automaticamente que é o dado que está à esquerda do caractere `|` quando usamos ele no template. E depois podemos passar outros parâmetros, no caso aqui o termo que usaremos para filtrar a lista de emails.

Depois retornamos a lista de emails filtrada, usando o próprio método `filter` do tipo `array`.

7. Todo que criamos no Angular deve ser informado ao framework de alguma forma. O pipe não é diferente, então, vamos declarar ele em **AppModule**:

`#./src/app/app.module.ts`

```

declarations: [
  AppComponent,
  CaixaDeEntradaComponent,
  FiltroPorAssunto
],

```

8. No template de **CaixaDeEntradaComponent**, no `*ngFor` colocarmos o caractere `|` ao lado de `emailList`, e logo em seguida chamamos o pipe que criamos `filtroPorAssunto`, para passar o segundo parâmetro ao pipe, colocamos `:` e a propriedade da classe que guarda o valor a `termoFiltro`:

`#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.html`

```

<ul>
  <li *ngFor="let email of emailList | filtroPorAssunto:termoParaFiltro">
    <cmail-list-item>
      [destinatario]="email.destinatario"
      [assunto]="email.assunto"
      [introducaoDoConteudo]="email.introducaoDoConteudo"
      [dataDeEnvio]="email.dataDeEnvio"
      (eventoVaiRemover)="handleRemoveEmail($event, email.id)"
    </cmail-list-item>
  </li>
</ul>

```

EXERCÍCIO: FILTRANDO ARRAYS DE ACORDO COM O NOVO ANGULAR

53.1 OBJETIVO

Os pipes são muito legais, e muita gente sentiu falta deste pipe de filtro de uma lista por um termo, oferecido nativamente pelo Angular, como havia no antigo AngularJS. Porém devido da experiência com esta operação, que é muito custosa para o framework/JavaScript/navegador, principalmente quando a lista a ser filtrada é muito grande e com muitos dados, causa lentidão, baixa performance e mal funcionamento, e isto era uma das principais reclamações do AngularJS.

Hoje, de acordo com a própria documentação, o time do Angular não recomenda usarmos pipes para este tipo de funcionalidade, a não ser que seja para lista pequenas e simples.

Com isto, vamos nós mesmos implementar nosso filtro de emails, e ao invés de usar o pipe, usaremos um método da classe que faz exatamente a mesma coisa.

Referências:

<https://angular.io/guide/pipes#appendix-no-filterpipe-or-orderbypipe>

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

53.2 PASSO A PASSO COM CÓDIGO

1. Na classe de **CaixaDeEntradaComponent** vamos criar um método `filtrarEmailsPorAssunto` que faz a mesma coisa que o pipe estava fazendo, a mesma lógica:

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.ts
```

```
filtrarEmailsPorAssunto() {  
  const termoParaFiltroEmMinusculo = this.termoParaFiltro.toLowerCase();  
  
  return this.emailList.filter( email => {  
    const assunto = email.assunto.toLowerCase()  
    return assunto.includes(termoParaFiltroEmMinusculo)  
  })  
}
```

2. No template de **CaixaDeEntradaComponent** no lugar do pipe e de `emailList`, vamos invocar o método `filtrarEmailsPorAssunto` da classe:

```
#./src/app/modules/caixa-de-entrada/caixa-de-entrada.component.html
```

```
<ul>  
  <li *ngFor="let email of filtrarEmailsPorAssunto()">  
    <cmail-list-item  
      [destinatario]="email.destinatario"  
      [assunto]="email.assunto"  
      [introducaoDoConteudo]="email.introducaoDoConteudo"  
      [dataDeEnvio]="email.dataDeEnvio"  
      (eventoVaiRemover)="handleRemoveEmail($event, email.id)"  
    ></cmail-list-item>  
  </li>  
</ul>
```

EXERCÍCIO DESAFIO: DESENVOLVENDO A PÁGINA INTERNA DOS EMAILS

54.1 OBJETIVO

Neste momento, já terminamos todas as funcionalidades principais do nosso CMail. Ao longo do desenvolvimento passamos por diversos pontos onde exploramos juntos desde as bases do Angular, como organizar arquivos até mesmo dicas de arquitetura e de qualidade de código. Esse exercício é um desafio para evoluirmos ainda mais as *skills* com o Angular, onde teremos meio que uma "tarefa" e a ideia é que com os conhecimentos e a estrutura atual do projeto tentar fazer as implementações necessárias. Não se preocupe com o layout, o importante é conseguir fazer as funcionalidades e criar um componente que seja responsável por representar algo que possa ser reutilizado.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

54.2 O QUE DEVE SER IMPLEMENTADO?

- Devemos criar a página interna para visualizarmos um email, essa deve ser um módulo carregado assíncronamente;
 - O id do email deve ser passado por meio da URL e as informações desse email devem ser resgatadas por meio de um método no serviço que faz as operações na parte da API que lida com emails;

- Essa página deve ter um botão possibilitando a exclusão do email e fazendo um redirect para o inbox;
- Essa página deve ter um botão de voltar para a listagem de emails;
- O título da página deve passar a ser o assunto do email;

54.3 DICAS

Esse exercício possui um tópico em específico que não foi passado em aula propositalmente. No dia a dia o lugar mais comum para tirar nossas dúvidas serão ferramentas como o Fórum da Alura, o Stackoverflow entre outros, mas principalmente a documentação do Angular em <https://angular.io/>.

Para dar um ponto final nessa busca, segue aqui como fazer para passar dados dinâmicos para uma rota no Angular: <https://angular.io/guide/router#route-parameters>.