UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAFAEL KLYNGER DA SILVA DANTAS

GERADOR DE CÓDIGO BASE PARA APLICAÕES BACK-END QUE USAM MVC

CAMPINA GRANDE - PB

2019

RAFAEL KLYNGER DA SILVA DANTAS

GERADOR DE CÓDIGO BASE PARA APLICAÕES BACK-END QUE USAM MVC

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Adalberto Cajueiro de Farias.

CAMPINA GRANDE - PB

2019

D192g Dantas, Rafael Klynger da Silva.
Gerador de código base para aplicações back-end que usam MVC. / Rafael Klynger da Silva Dantas. – 2019.

8 f.

Orientador: Prof. Dr. Adalberto Cajueiro de Farias.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Desenvolvimento web. 2. Aplicações back-end. 3. Gerador de código base. 4. Desenvolvimento de software. 5. Verbos REST. 6. Representational State Transfer. 7. Código de controller. I. Farias, Adalberto Cajueiro de. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

RAFAEL KLYNGER DA SILVA DANTAS

GERADOR DE CÓDIGO BASE PARA APLICAÕES BACK-END QUE USAM MVC

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Adalberto Cajueiro de Farias
Orientador – UASC/CEEI/UFCG

Professor Dr. Eanes Torres Pereira
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Disciplina TCC– UASC/CEEI/UFCG

Trabalho aprovado em: 25 de novembro 2019.

CAMPINA GRANDE - PB

Gerador de código base para aplicações Back-end que usam MVC

Rafael Klynger da Silva Dantas
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
rafael.klynger@gmail.com

Resumo

Desenvolver aplicações backend com operações como criar, editar, deletar e recuperar dados (CRUD) é algo muito comum. O desenvolvimento inicial dessas aplicações se resume a definir rotas, entidades e relações o que faz com que o desenvolvedor crie arquivos com códigos bastante semelhantes. Esse desenvolvimento inicial pode ser automatizado pela nossa ferramenta, que ao receber uma especificação em JSON de quais entidades existem e como elas se relacionam consegue criar uma aplicação servidor na linguagem selecionada que possui uma API REST com os verbos básicos (GET, PUT, POST E DELETE). Assim, acelerando o desenvolvimento inicial da aplicação.

Keywords

Desenvolvimento Web, ferramenta, automatização, CRUD. back-end.

1. INTRODUÇÃO

Ao iniciar o desenvolvimento do back-end MVC [1] (Model-View-Controller) de uma aplicação Web, o desenvolvedor captura os requisitos e então inicia as decisões arquiteturais onde será definido quais entidades a aplicação terá e como será as suas relações. Após esse passo o desenvolvimento o fluxo segue com a utilização dos artefatos gerados pelas fases anteriores. Esse desenvolvimento inicial gera um código sem muita regra de negócio onde há várias entidades que se relacionam e são expostas na API pelos verbos da REST. Ver imagens 1.1 e 1.2. REST (*Representational State Transfer*) é um modelo arquitetural para interfaces de aplicações no qual seu conceito gira em torno da definição de recursos. Por exemplo, em uma aplicação de um blog podemos dizer que posts, comentários e usuários são recursos da aplicação que podem ser obtidos, criados, modificados, removidos através da utilização da interface [2].

54 lines (47 sloc) | 1.36 KB

```ts
import { PostService } from './post.service';
import { UserService } from '../user/user.service';
import { CreatePostDto } from '../shared/models/post/create-post.dto';
import { UpdatePostDto } from '../shared/models/post/update-post.dto';
import {
  Get,
  Put,
  Post,
  Body,
  Param,
  Query,
  Delete,
  HttpCode,
  Controller,
  HttpStatus,
  BadRequestException,
} from '@nestjs/common';

@Controller('/post')
export class PostController {
  constructor(private readonly postService: PostService, private readonly userService: UserService) {}

  @Get(':id')
  public getPost(@Param('id') id: string) {
    return this.postService.getPost(id);
  }

  @Get()
  public getPosts(@Query() query: QueryType) {
    const { userId }: { userId?: string } = query;
    if (userId === undefined) {
      throw new BadRequestException('You must pass the query \'userId\'');
    }
    const user = this.userService.getUser(userId);
    return this.postService.getPosts(user.posts);
  }

  @Post()
  public createPost(@Body() postDto: CreatePostDto) {
    return this.postService.addPost(postDto);
  }

  @Put(':id')
  public updatePost(@Body() postDto: UpdatePostDto, @Param('id') id: string) {
    return this.postService.updatePost(postDto, id);
  }

  @Delete(':id')
  @HttpCode(HttpStatus.NO_CONTENT)
  public deletePost(@Param('id') id: string) {
    this.postService.deletePost(id);
  }
}
```

Imagem 1.1 - Código de Controller de uma entidade chamada Post.

```
41 lines (35 sloc)  934 Bytes
import { UserService } from './user.service';
import { CreateUserDto } from '../shared/models/user/create-user.dto';
import { UpdateUserDto } from '../shared/models/user/update-user.dto';
import {
  Put,
  Get,
  Post,
  Body,
  Param,
  Delete,
  HttpCode,
  HttpStatus,
  Controller,
} from '@nestjs/common';

@Controller('user')
export class UserController {
  constructor(private readonly userService: UserService) {}

  @Get(':id')
  getUser(@Param('id') id: string) {
    return this.userService.getUser(id);
  }

  @Post()
  createUser(@Body() createUserDto: CreateUserDto) {
    return this.userService.createUser(createUserDto);
  }

  @Put(':id')
  updateUser(@Body() updateUserDto: UpdateUserDto, @Param('id') id: string) {
    return this.userService.updateUser(updateUserDto, id);
  }

  @Delete(':id')
  @HttpCode(HttpStatus.NO_CONTENT)
  deleteUser(@Param('id') id: string) {
    this.userService.deleteUser(id);
  }
}
```

1.2 - Código de Controller de uma entidade chamada User.

Essas aplicações têm uma estrutura bem definida de camadas onde cada entidade tem uma implementação de cada camada que a aplicação decide implementar.

Controller	Camada responsável por implementar cada recurso e verbo que será exposto para o cliente, é nela também onde é feita uma validação inicial para checagem de campos que estão faltando por exemplo.
Service	Responsável por implementar a lógica de negócio da aplicação e por se comunicar com outros serviços
Repository	Responsável pela persistência dos dados
Model	Model não é uma camada da aplicação, mas é necessário em vários projeto para definição de tipos.
Camadas adicionais	Alguns geradores possuem outras camadas além das citadas acima, como exemplo temos a camada de módulo que o framework *NestJS* faz uso.

Essa solução possui apenas um gerador que gera código typescript com *API REST* que utiliza *NestJS* como framework, mas graças a sua arquitetura é muito simples criar um gerador para outro framework e até outra linguagem, como Java + *SpringBoot* por exemplo.

As soluções atuais para esse problema se resumem a CLIs de frameworks específicos que geram apenas o seu código base sem atributos ou métodos em seus arquivos. CLIs são programas de linha de comando que aceitam texto com entrada e executam funções do sistema [3].

2. SOLUÇÃO

2.1 Descrição

O objetivo deste trabalho é desenvolver uma aplicação que aceite uma definição em JSON diratamente na API sobre a aplicação e dê como resposta um projeto com o código gerado de cada camada que foi decidido implementar para cada entidade. Além dessa opção para criar um novo projeto é possível utilizar o frontend da aplicação possui um formulário para criação da entrada esperada pela API.

O código gerado pela aplicação irá funcionar caso tente rodar apenas o código gerado se o usuário escolher por gerar todas as camadas, caso contrário, a ferramenta irá gerar o projeto, mas talvez não seja possível rodá-lo sem ter que alterar algo no código.

Para que o código gerado funcione, é necessário também que toda entidade possua um atributo chamado id e que o nome do projeto e das entidades estejam em *Pascal Case* [4].

A representação de atributos que referenciam outras entidades é através do seu *id*, com isso se uma entidade 'A' possui uma referência para entidade 'B' que é representada por um atributo 'c', o tipo de 'c' é do mesmo tipo dos *ids* da aplicação.

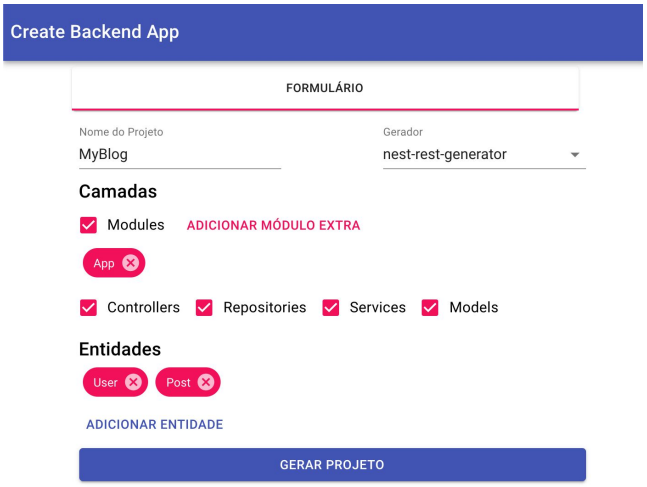

Imagem 2.1 - Front-end da aplicação.

2.2 Arquitetura

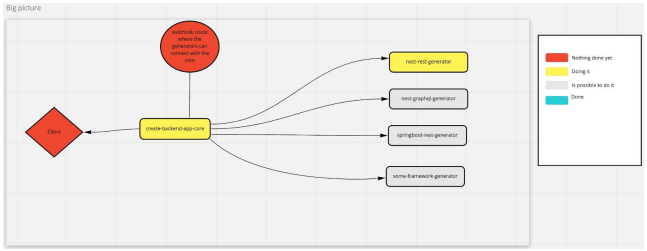

Imagem 2.2 - Macro arquitetura [5]

A arquitetura é definida da seguinte forma, há 3 aplicações, a principal delas é o *create-backend-app-core [6]*, essa é aplicação central responsável por gerar o projeto como um todo e é com ela que a outra app, o cliente, irá conversar. Por trás do core existem os geradores, que geram o código de fato, esses geradores obedecem uma especificação de API na qual o core irá conversar.

Para o core saber da existência de um gerador, esse gerador se comunica com uma rota específica do core dizendo o seu nome, quais camadas ele consegue gerar e qual o tipo de API (REST ou GraphQL).

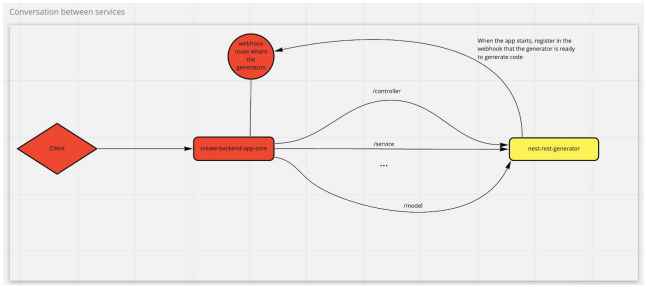

Imagem 2.3 - Interação entre os serviços

2.2.1 Tecnologias do backend

No backend foi utilizado *NodeJS* com o framework *NestJS*, como não há nenhum tipo de persistência de dados não houve a utilização de nenhum banco de dados. Há uma grande utilização da biblioteca *Ramda* que é uma biblioteca com várias funções utilitárias do paradigma funcional.

2.2.2 Funcionamento do gerador de nest com API REST

Para geração de código de fato foi utilizado interpolação de strings [7]. E pelo fato de tratar o código como string e usar bastante concatenação de strings, o paradigma funcional foi bastante usado no desenvolvimento, por isso o uso da biblioteca *ramda*.

```
export function generateDelete(entityName: string, tabSize: number, layerBellow?: Layer) {
  const actionLine = getDefaultMethodActionContent(Layer.controller, entityName, Verb.DELETE, 'id', layerBellow, false);
  const outerIdentationSpaces = getIdentation(tabSize, DEFAULT_INNER_CLASS_TABS);
  const innerIdentationSpaces = getIdentation(tabSize, DEFAULT_INNER_CLASS_TABS + 1);

  return `${outerIdentationSpaces}@Delete(':id')
${outerIdentationSpaces}@HttpCode(HttpStatus.NO_CONTENT)
${outerIdentationSpaces}delete${entityName}(@Param('id') id: string) {
${innerIdentationSpaces}${actionLine}
${outerIdentationSpaces}}`;
}

export function generatePut(entityName: string, tabSize: number, layerBellow?: Layer) {
  const returnLine = getReturnLine(Layer.controller, entityName, Verb.PUT, `id, update${entityName}Dto`, layerBellow);
  const outerIdentationSpaces = getIdentation(tabSize, DEFAULT_INNER_CLASS_TABS);
  const innerIdentationSpaces = getIdentation(tabSize, DEFAULT_INNER_CLASS_TABS + 1);

  return `${outerIdentationSpaces}@Put(':id')
${outerIdentationSpaces}update${entityName}(@Body() update${entityName}Dto: Update${entityName}Dto, @Param('id') id: string) {
${innerIdentationSpaces}${returnLine}
${outerIdentationSpaces}}`;
}

export function generateGet(entityName: string, tabSize: number, layerBellow?: Layer) {
  const returnLine = getReturnLine(Layer.controller, entityName, Verb.GET, 'id', layerBellow);
  const outerIdentationSpaces = getIdentation(tabSize, DEFAULT_INNER_CLASS_TABS);
  const innerIdentationSpaces = getIdentation(tabSize, DEFAULT_INNER_CLASS_TABS + 1);

  return `${outerIdentationSpaces}@Get(':id')
${outerIdentationSpaces}get${entityName}(@Param('id') id: string) {
${innerIdentationSpaces}${returnLine}
${outerIdentationSpaces}}`;
}
```

Imagem 2.4 - Utilização de interpolação de strings para geração dos métodos DELETE, PUT e GET de um Controller

Esta técnica de interpolação de strings é bastante semelhante com a utilização de arquivos templates para geração de código.

2.2.3 Criando o seu gerador de código

Por ter uma estrutura bem definida, para criar o seu gerador de código basta apenas seguir as especificações do gerador de nest com API REST [8], ou até mesmo fazer um fork do projeto e alterar apenas as funções que geram o código, uma vez que é possível enxergar uma especificação de uma aplicação backend MVC como um caminho em um grafo de árvore finito onde cada folha representa uma especificação diferente.

2.3 Tecnologias do front-end

No front-end foi utilizado três grandes dependências. *React* como biblioteca de renderização, *MaterialUI* como biblioteca de componentes de layout e *Formik* para gerenciamento do estado do formulário.

React [9] é uma biblioteca para desenvolvimento de interface de usuário. Essa biblioteca torna muito fácil o

desenvolvimento por ser declarativa e baseada em componentes, o que torna o reuso de código bastante simples.

MaterialUI [10] é uma biblioteca de componentes *React* que implementa o padrão de design *Material Design* [11] o que facilita o desenvolvimento da interface.

Formik [12] é uma biblioteca para gerenciamento do estado de formulários em *React.* Ela é responsável por controlar a mudança dos campos e por gerenciar os campos com valores errados.

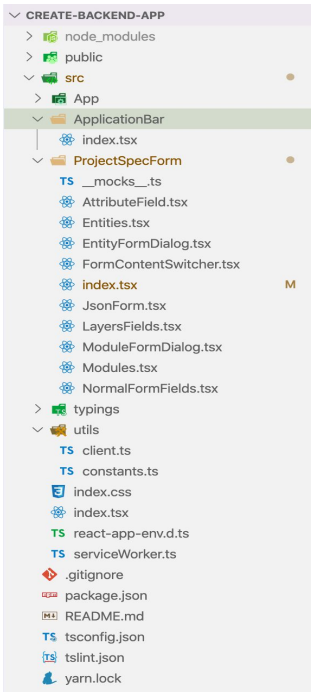

Imagem 2.5 - Estrutura do front-end

3. SISTEMA EM USO

Bons produtos de software são aqueles que deixam o usuário satisfeito ao utilizá-lo tanto no aspecto de experiência de usuário e simplicidade na utilização quanto de conseguir executar as funcionalidades que propõe. Produtos que não possuem uma boa experiência ou conseguem fazer apenas parte do que ele se propõe não conseguem se manter relevante tanto no mercado quanto na comunidade open source.

Por isso é necessário ter um *feedback* dos seus usuários em relação a utilização do sistema. Existem várias técnicas para extrair essa informação, como registro de utilização do sistema, estudos para indicar o estado emocional do usuário no momento de interação [ref], questionários sobre a experiência de interação com o sistema, entre outros.

3.1 Metodologia

Para obter essas métricas de satisfação ao interagir com sistema foi escolhido a utilização de um formulário com seis afirmativas [ref] onde o usuário informa um número de 1 a 7 para cada afirmativa, onde 1 significa que ele concorda completamente e 7 discorda completamente com a afirmativa.

O perfil dos usuários que responderam o questionário são alunos e ex-alunos do curso de computação que trabalham ou têm um conhecimento sobre desenvolvimento de aplicações web, não necessariamente desenvolvimento de aplicações back-end. Para que os usuários utilizassem o sistema foi passado os repositórios necessários para rodar a aplicação, eles baixaram e rodaram os três repositórios e começaram a preencher o formulário. Após baixar o projeto gerado eles instalaram as dependências e rodaram a aplicação para testar e analisar o código gerado. Alguns usuários tiveram problemas em entender algumas sessões do sistema, como a marcação de quais camadas o sistema terá. As pessoas que tiveram essa dúvida não tinham conhecimento sobre o tipo framework no qual elas estavam gerando código, o que torna bastante plausível a indagação, já que usuários que tinham conhecimento sobre a arquitetura de *NestJS*.

3.2 Resultados da pesquisa

Para calcular a satisfação dos usuários foi necessário sumarizar os dados da pesquisa. Com isso temos o resultado na imagem [imagem] que tem a médias aritmética de cada afirmativa.

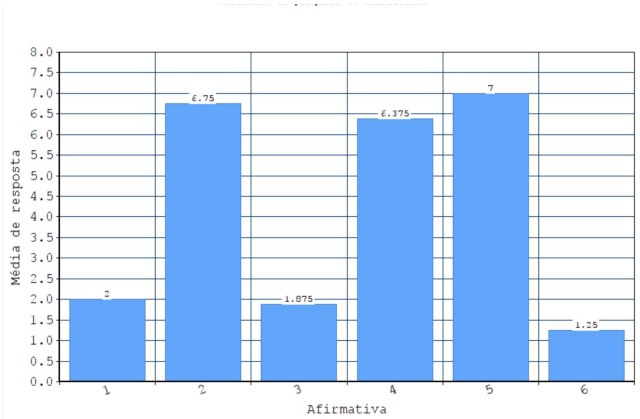

Imagem 3.1 - Gráfico de barras com a resposta média dos usuários para as afirmativas da tabela 3.1

Com esse resultado é possível concluir que a ferramenta possui uma primeira versão utilizável e consegue gerar um código inicial bom, uma vez que os usuários responderam que não precisaram apagar nada ou quase nada do que foi gerado pela ferramenta. O sistema também não aparenta ter inconsistências e é de simples utilização. O sistema falhou bastante em relação às mensagens de erro quando o usuário fez algo de errado.

Número da Afirmativa	Afirmativa
1	Acho que usaria esse sistema com frequência
2	Achei o sistema desnecessariamente complexo

3	Achei o sistema muito simples de ser utilizado
4	Quando errei o sistema me alertou o que estava errado
5	Acho que há muita inconsistência com o sistema
6	O código gerado pelo sistema é bom e não precisei apagar nada do que foi gerado

Tabela 3.1 - Legenda para o gráfico de barras da imagem 3.1

4. TRABALHOS RELACIONADOS

Algumas ferramentas, como o *NestCLI*, consegue fazer algo muito similar, com um comando é possível gerar um projeto com os arquivos base e com outros comandos é possível gerar módulos, serviços, controladores, etc. Mas esses arquivos possuem apenas *imports* da biblioteca e uma classe com o nome passado para criação.

5. TRABALHOS FUTUROS

Para que a ferramenta se torne relevante e seja utilizada o suporte contínuo é necessário. Com isso há algumas melhorias que poderiam ser feitas para que o sistema fique ainda mais simples.

A primeira coisa a ser feita é trazer mensagens de erro para o usuário e mensagens de ajuda e talvez referências para uma explicação mais clara do que é o que ele está vendo e como funciona.

Outro ponto é a criação de um novo gerador, para que o *core* da aplicação evolua e se torne mais genérico e consiga se comunicar mais facilmente com geradores que outras pessoas poderiam criar.

A facilitação da criação de geradores faria com que o sistema se tornasse mais acessível também, uma vez que o usuário poderia criar o seu gerador conseguir criar APIs mais facilmente. Para isso seria necessário criar um gerador de geradores, isso poderia ser alcançado com uma especificação de cada função que os geradores executa para gerar um trecho de código, podendo assim criar uma interface simples para que o usuário implemente suas próprias funções e então crie o seu próprio gerador.

6. AGRADECIMENTOS

Queria agradecer primeiramente a minha mãe por ter trabalhado tanto e se esforçado tanto para me criar e para que eu tivesse acesso a escola e posteriormente a uma universidade, aos meus amigos e colegas de faculdade por terem me ajudado a passar por todos esses anos de faculdade, ao professor Adalberto por ter me dado uma oportunidade em um projeto onde conheci o desenvolvimento web e desde então me apaixonei e a todos os outros professores e docentes por compartilharem o conhecimento que possuem, em especial os professores Matheus Gaudêncio do Rêgo e Francisco Villar Brasileiro pelas melhores aulas que tive na graduação e ao professor Rohit Gheyi pelo seu projeto maravilhoso de maratonas de programação.

7. REFERENCES

[1] MVC definition, https://techterms.com/definition/mvc

[2] REST: Princípios e boas práticas, https://blog.caelum.com.br/rest-principios-e-boas-praticas/

[3] What is a Comand Line Interface (CLI)? https://www.w3schools.com/whatis/whatis_cli.asp

[4] PascalCase, https://techterms.com/definition/pascalcase

[5] Diagramas com a arquitetura completa, https://miro.com/app/board/o9J_kxdTfSA=/.

[6] Repositório da aplicação core, https://github.com/Klynger/create-backend-app-core

[7] Template strings in javascript, MDN documentation. Link: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/template_strings.

[8] Especificação da API que um gerador de código deve ter. Link: https://github.com/Klynger/nest-rest-generator/blob/master/README.md.

[9] React, https://reactjs.org/

[10] MaterialUI, https://material-ui.com/

[11] Material Design, https://material.io/design/

[12] Formik, https://jaredpalmer.com/formik/docs/overview

Sobre o autor:

Rafael Klynger da Silva Dantas, programador web, gosta muito de desenvolvimento frontend, mexer com animações, adora entender sobre como o navegador funciona, como estilizar aplicações web de forma performática entre outras coisas.