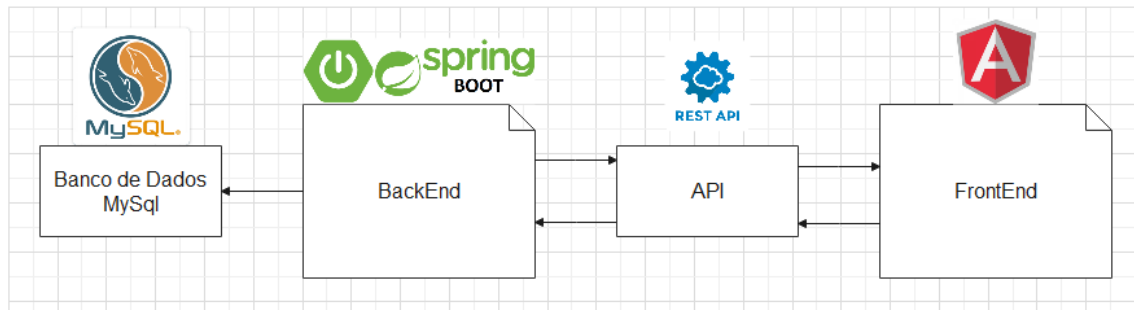


Especificação Técnica

AVALIAÇÃO TÉCNICA – CASTGROUP

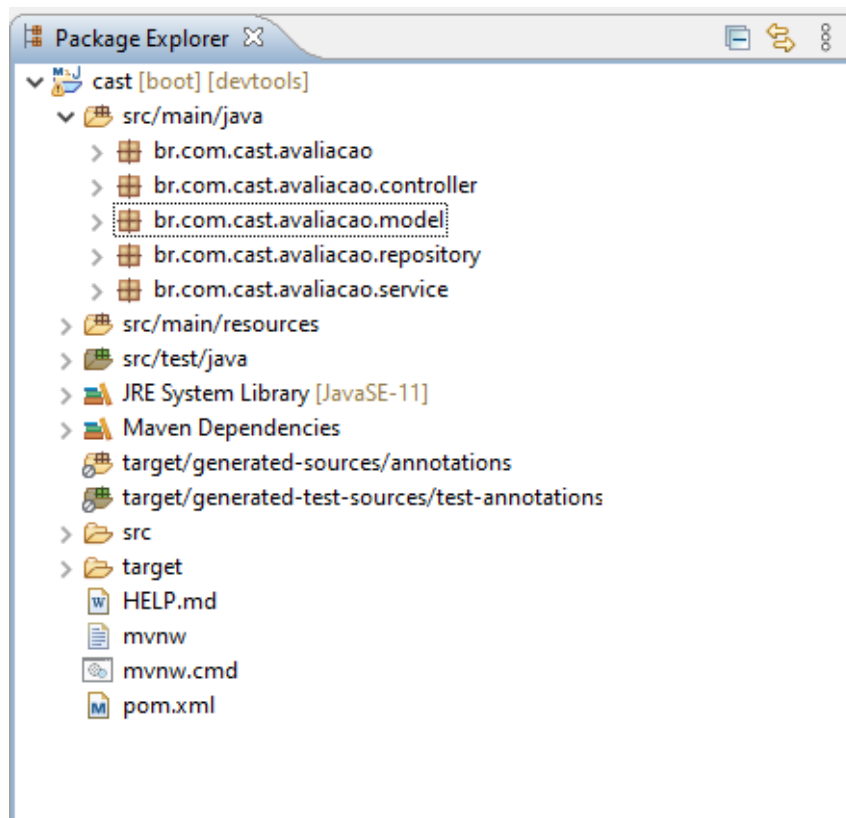
Após estudar as tecnologias para o desenvolvimento do projeto, foi desenhado o seguinte processo.



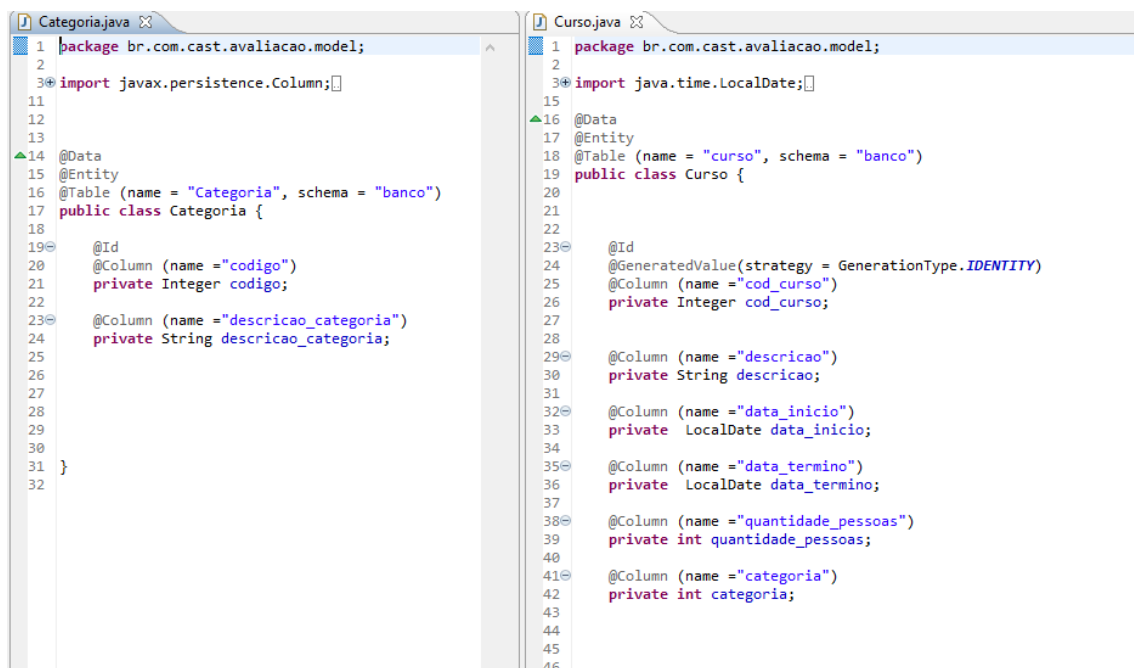
1. O primeiro passo foi criar o banco de dados, como solicitado nos critérios de aceite, foi necessário respeitar a estrutura. Então gerando duas tabelas no banco de dados, curso e categoria. As duas tabelas deveriam ser relacionadas de alguma maneira, além disso a tabela categoria precisa dos dados inseridos nela, por isso foi codificado da seguinte maneira.

```
banco-SCRIPT* x
Limit to 1000 rows
1 • create database banco;
2
3 • create table curso(
4     cod_curso int not null auto_increment,
5     descricao varchar(50) not null,
6     data_inicio date not null,
7     data_termino date not null,
8     quantidade_pessoas int,
9     categoria int not null,
10
11     primary key (cod_curso)
12 );
13
14
15 • create table categoria(
16     codigo int not null,
17     descricao_categoria varchar (50) not null,
18
19     primary key (codigo)
20 );
21
22 • alter table curso
23     add foreign key (categoria)
24     references categoria(codigo);
25
26 • insert into categoria
27     values
28     (1, 'Comportamental'),
29     (2, 'Programacao'),
30     (3, 'Qualidade'),
31     (4, 'Processos');
```

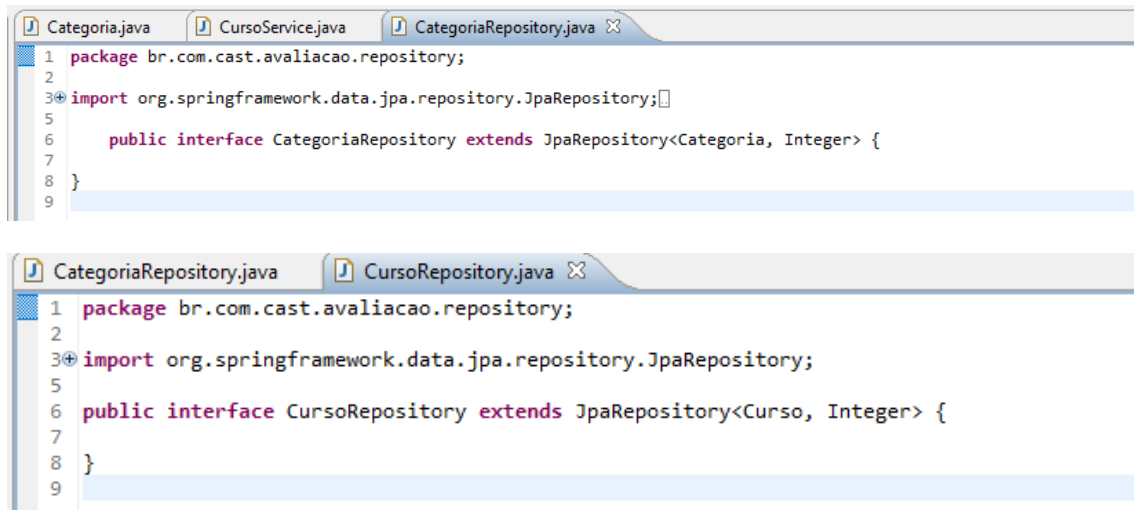
2. O segundo passo foi desenvolver o BackEnd, o projeto foi dividido em packages como foi solicitado pelos critérios de aceite. Ficando com o seguinte “esqueleto” do projeto.



O package Model contém duas classes, sendo elas curso e categoria, cada uma delas com suas próprias características. Todas as características foram mapeadas de acordo com as colunas do banco de dados. É importante dizer que foi utilizado a biblioteca Lombok, que me permitiu não codificar os getters, setters, hash e etc.



O package Repository, organiza as interfaces de acesso à dados, é importante destacar que foi utilizado o JPA, que descreve uma interface comum para frameworks de persistência de dados.



```
1 package br.com.cast.avaliacao.repository;
2
3+import org.springframework.data.jpa.repository.JpaRepository;
4
5
6 public interface CategoriaRepository extends JpaRepository<Categoria, Integer> {
7
8 }
9
```

```
1 package br.com.cast.avaliacao.repository;
2
3+import org.springframework.data.jpa.repository.JpaRepository;
4
5
6 public interface CursoRepository extends JpaRepository<Curso, Integer> {
7
8 }
9
```

O package Controller, é responsável por criar o caminho das APIs para o consumo front, então basicamente temos as seguintes URLs para o consumo. E o package Service faz as validações necessárias para então salvar todos os dados no banco de dados, através do package Repository.

Get: <http://localhost:8080/api/cursos>

Get por Código específico: http://localhost:8080/api/cursos/{cod_curso}

Put: http://localhost:8080/api/cursos/{cod_curso}

Post: <http://localhost:8080/api/cursos/>

Delete: http://localhost:8080/api/cursos/{cod_curso}

Para exemplificar na explicação irei descrever o passo a passo do método Post

```
//Realiza a tentativa cadastro de cursos, e envia para o metodo salvarCurso lá na classe CursoService
@PostMapping (path = "/cursos")
public ResponseEntity<String> salvarCurso (@RequestBody Curso curso) {
    try {
        return cursoService.salvarCurso(curso);
    } catch (Exception e) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

Fazendo o mapeamento da URL, para o consumo e enviar para um método existente na classe CursoService. O método se chama salvarCurso, aqui será feita a validação das regras de negócio, chamando assim outros métodos dentro da classe CursoService, uma das regras é não permitir o cadastro de cursos com a data inferior a de hoje. Então o primeiro processo foi capturar a data de hoje e deixá-la no mesmo formato das que serão comparadas.

```
//Metodo para retornar data atual
public String dataAtual() {

    //Pegando data atual e formatando strDateFormat
    Date date = new Date();
    String strDateFormat = "yyyy-MM-dd";
    DateFormat dateFormat = new SimpleDateFormat(strDateFormat);
    return dateFormat.format(date);
}
```

E depois chamada a data atual para a primeira regra e assim podemos realizar a verificação.

```
//Metodo para validar 1 regra de negocio
public boolean primeiraRegra (Curso curso) {

    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

    LocalDate localDate = LocalDate.parse(dataAtual(), formatter);

    // VERIFICANDO SE A DATA INICIO OU FIM ESTÁ SENDO AGENDADA EM UM PERIODO ANTES DA ATUAL
    boolean init = curso.getData_inicio().isBefore( localDate );
    boolean fim = curso.getData_termino().isBefore( localDate );

    if(init == true || fim == true) {
        return false;
    }

    return true;
}
```

E assim também é desenvolvido o método para comparar se existe alguns cursos no mesmo período. Isso foi possível através da função isBefore, isAfter e isEqual do java.

```
public boolean segundaRegra (Curso curso) {
    List<Curso> listCurso = cursoRepository.findAll();

    for (int i = 0; i < listCurso.size(); i++) {
        System.out.println(listCurso.get(i));
        Curso cursoAux = listCurso.get(i);
        System.out.println(cursoAux.getDescricao());

        if(curso.getData_inicio().isAfter(cursoAux.getData_inicio()) && curso.getData_inicio().isBefore(cursoAux.getData_termino()))
        {
            return false;
        }
        else if (curso.getData_termino().isAfter(cursoAux.getData_inicio()) && curso.getData_termino().isBefore(cursoAux.getData_termino())){
            return false;
        }
        else if (curso.getData_inicio().isBefore(cursoAux.getData_inicio()) && curso.getData_termino().isAfter(cursoAux.getData_termino())){
            return false;
        }
        else if (curso.getData_inicio().isEqual(cursoAux.getData_inicio())&& curso.getData_termino().isEqual(cursoAux.getData_termino())) {
            return false;
        }
    }

    return true;
}
```

Feito isso, trouxe esses retornos do tipo boolean para o método salvarCurso. Então com as duas regras terminadas, já é possível terminar o método para salvar um curso.

```
//Salvar um curso
public ResponseEntity<String> salvarCurso (Curso curso) throws Exception {

    boolean resultadoPrimeiraRegra = primeiraRegra(curso);

    if (resultadoPrimeiraRegra) {

        System.out.println("A data deve ser à partir de hoje");
        //Metodo para validar periodos das datas
        boolean resultadoSegundaRegra = segundaRegra(curso);
        if (!resultadoSegundaRegra) {

            return new ResponseEntity<>("Existe(m) curso(s) planejados(s) dentro do período informado.",HttpStatus.NOT_FOUND);
        }

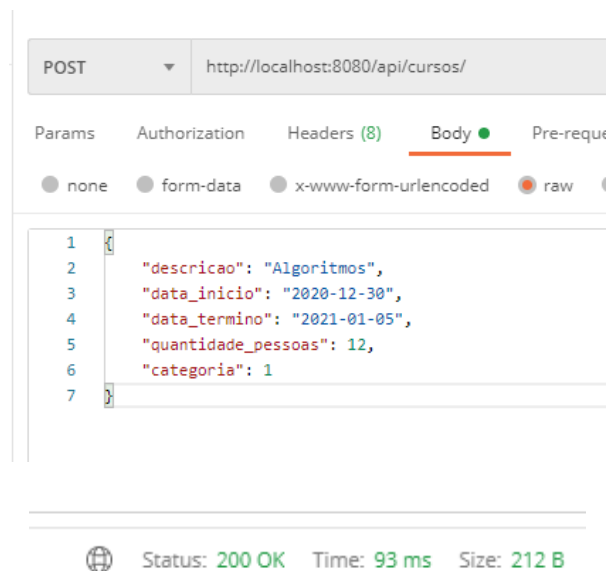
    }else {

        return new ResponseEntity<>("A data é inferior a de hoje",HttpStatus.NOT_FOUND);
    }

    cursoRepository.save(curso);

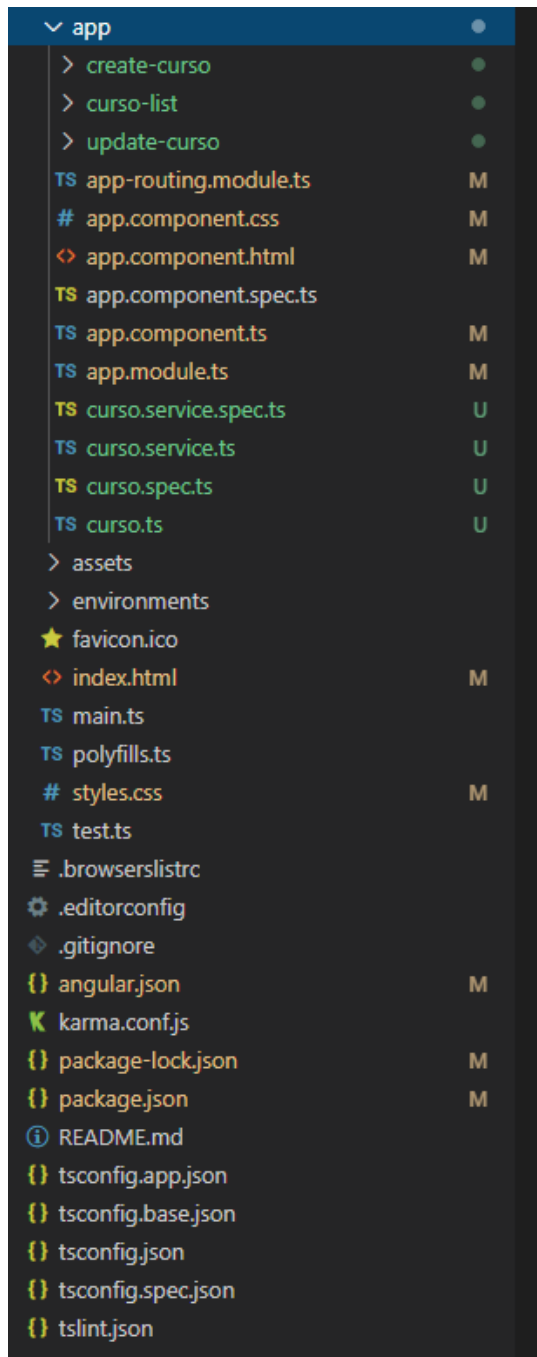
    return new ResponseEntity<>(HttpStatus.OK);
}
```

A partir disso é possível cadastrar um curso através da API, então usando a ferramenta PostMan, realizar os cadastros e testar se estava tudo correto. Então passando os parâmetros para o cadastro e realizando o POST pelo postman. E recebemos o status 200, que significa que deu tudo certo.



Para verificar usamos o método Get, que foi implementado do mesmo jeito, criando a chamada da API na classe Controller e depois validando na Services. Da mesma forma foi possível usar o Get, o Delete e o Put no postman para simular. Depois de feito e testados todos os passos do Crud e preparado o back para enviar os dados para o front com as exceções mapeadas. Já era possível começar a trabalhar no front.

3. O terceiro passo foi entender e aplicar as regras no front utilizando o angular 10, após a instalação e a configuração do ambiente, criamos o seguinte esqueleto, com as pastas dos Métodos Create, List e Update neste caso o Delete fica dentro do list para remover os itens direto da lista de cursos. Cada uma das pastas criadas com suas configurações, como HTML, CSS e TS. O Front ficou com o seguinte esqueleto.



Para exemplificar falarei dos métodos Create que seria o Salvar e o List o buscar todos os cursos. Mas antes é preciso falar mostrar o service, que foi criado e funciona exatamente como no BackEnd, onde recebemos as URLs mapeando todas as requisições e tratando os erros.

```
src > app > TS curso.service.ts > CursoService > getCursosList
1  import { Injectable } from '@angular/core';
2  import { HttpClient, HttpHeaders } from '@angular/common/http';
3  import { Observable } from 'rxjs';
4  import { Curso } from './curso';
5  import { HttpResponse } from '@angular/common/http';
6  import { catchError, retry } from 'rxjs/operators';
7  import { throwError } from 'rxjs';
8
9
10
11
12  @Injectable({
13    providedIn: 'root'
14  })
15  export class CursoService {
16
17    private baseUrl = "http://localhost:8080/api/cursos";
18
19    constructor(private httpClient: HttpClient) { }
20  }
```

Método Post, com o tratamento de erros feito no back para alertar sobre as datas dos cursos.

```
// Metodos para mapeamento das apis

getCursosList(): Observable<Curso[]>{
  return this.httpClient.get<Curso[]>(`${this.baseUrl}`);
}

createCurso(curso: Curso): Observable <Object>{
  return this.httpClient.post(`${this.baseUrl}`, curso ).pipe(
    catchError(error => {
      this.handleError(error);
      return throwError(error);
    })
  );
}

//Mapeando o erro para trazer do back as datas invalidas

private handleError(error: HttpResponse): string{
  switch (error.status) {
    case 404: {
      alert(error.error);
    }
    default: {
    }
  }
  return error.message;
}
```

E os demais métodos.

```

getCursoById(cod_curso: number): Observable<Curso>{
  return this.httpClient.get<Curso>(`${this.baseUrl}/${cod_curso}`);
}

updateCurso (cod_curso: number, curso: Curso): Observable<Object>{
  return this.httpClient.put(`${this.baseUrl}/${cod_curso}`, curso);
}

deleteCurso(cod_curso:number): Observable<Object>{
  return this.httpClient.delete(`${this.baseUrl}/${cod_curso}`);
}

```

Mostrando o método List. A parte da lógica ficou da seguinte maneira.

```

export class CursoListComponent implements OnInit {

  cursos: Curso[];

  constructor(private cursoService: CursoService,
    private router: Router) { }

  ngOnInit(): void {
    this.getCursos();
  }

  private getCursos(){
    this.cursoService.getCursosList().subscribe(data => {
      this.cursos = data;
    });
  }

  updateCurso(cod_curso: number){

    this.router.navigate(['update-curso', cod_curso]);

  }

  deleteCurso(cod_curso: number){
    this.cursoService.deleteCurso(cod_curso).subscribe( data =>{
      console.log(data);
      this.getCursos();
    })
  }

}

```

Mapeando isso no Html fica da seguinte maneira, criado um com as classes vinda lá do bootstrap, para ficar mais agradável o layout e o design, com as descrições no primeiro TR e os dados vindo da API, isso sendo possível através do ngFor. E por fim mapeado dois botões, editar e excluir.


```

<div class="table-responsive">
  <table class="table table-bordered">
    <thead class = "thead-dark">
      <tr>
        <th>Código do Curso</th>
        <th>Descrição</th>
        <th>Data de Início</th>
        <th>Data de Término </th>
        <th>Quantidade de Pessoas</th>
        <th>Categoria</th>
        <th>Ações</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor = "let curso of cursos" >
        <td>{{curso.cod_curso}}</td>
        <td>{{curso.descricao}}</td>
        <td>{{curso.data_inicio}}</td>
        <td>{{curso.data_termino}}</td>
        <td>{{curso.quantidade_pessoas}}</td>
        <td>{{curso.categoria}}</td>
        <td>
          <button (click) = "updateCurso(curso.cod_curso)" class="btn btn-info"> Atualizar </button>
          <button (click) = "deleteCurso(curso.cod_curso)" class="btn btn-danger" style="margin-left: 5px;"> Excluir </button>
        </td>
      </tr>
    </tbody>
  </table>
</div>

```

Já o método Create ficou com a parte lógica como é possível ver na imagem.

```

// Método para salvar um novo curso validando se todos os valores foram preenchidos
saveCurso(){

  let resultado: boolean

  if (this.curso.descricao = "" || (this.curso.data_inicio = "") || (this.curso.data_termino = "") || (this.curso.categoria= null)) {
    resultado = true;
  }

  if (resultado){

    this.cursoService.createCurso(this.curso).subscribe( data =>{
      console.log(data);
      this.goToCursoList();
    },
    error =>console.error(error));

  }

  alert("Preencha todos os campos");
}

goToCursoList(){
  this.router.navigate(['/cursos']);
}

onSubmit(){
  console.log(this.curso);
  this.saveCurso();
}

```

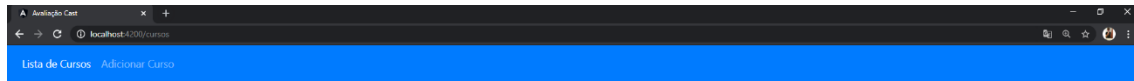
Primeiro fazendo uma verificação se todos os campos foram preenchidos, caso não emito um alert solicitando a preencher. Caso seja tudo preenchido, chamo método salvar feito lá no service e válido as regras de negócio feitas pelo back.

Em resumo o front é isso, as linhas de código estão disponíveis nas pastas dentro do Git.

Simulando a aplicação

Vamos simular aqui uma operação.

- Primeiro abrindo o site.



Código do Curso	Descrição	Data de Início	Data de Término	Quantidade de Pessoas	Categoria	Ações
121	Algoritmos	2020-12-30	2021-01-05	12	1	<button>Atualizar</button> <button>Excluir</button>

- Já foi possível ver o curso que cadastramos pela API no back.
- Agora testando colocando uma data inferior a de hoje.

localhost:4200 diz
A data é inferior a de hoje

OK

Adicionar curso

Descrição

Data de Início

Data de Término

Quantidade de Pessoas

Categoria

Salvar

- Agora que uma das regras de negócio foi validada, tentarei colocar a data, dentro do período que cadastramos o curso via postman.

Avaliação Cast

curso

localhost:4200 diz
Existe(m) curso(s) planejados(s) dentro do período informado.

OK

Adicionar Curso

Descrição

Data de Início

Data de Término

Quantidade de Pessoas

Categoria

Salvar

- Dessa forma é possível travar datas que interferem outros cursos, para testar colocamos um curso para daqui 2 meses. Logo o curso será salvo e será redirecionado para a lista de cursos.

[Lista de Cursos](#)
[Adicionar Curso](#)



Código do Curso	Descrição	Data de Início	Data de Término	Quantidade de Pessoas	Categoria	Ações
121	Algoritmos	2020-12-30	2021-01-05	12	1	<div>Atualizar</div> <div>Excluir</div>
122	Banco de Dados	2021-02-01	2021-02-07	5	1	<div>Atualizar</div> <div>Excluir</div>

- Dessa mesma forma o atualizar e o excluir, fazem suas funções também. Fechando então o CRUD proposto.