Desafio 6

1. a) Evitar duplicação de código, acessar e armazenar dados no servidor. Retorna os dados para serem mostrados no component.

b) O ideal é que os componentes tenham apenas a interação com o usuário, e a lógica fique na classe de services.

c) Através da injeção de dependências. É preciso informar o @Injectable() na classe.

d) Pode realizar qualquer tarefa. Ex.**:** Registro no console, validar entrada de usuários, buscar dados no servidor.

e) Falso. Pode estar vinculado a um módulo, mas não é obrigatório. Ao invés de estar em um único módulo separadamente, pode estar disponível a toda a aplicação.

f) Verdadeiro. O angular garante uma única instância do provider.

g)

c > app > data-form > data-form.service.ts > DataFormService > setDataForm

```
1
2   import { Injectable } from '@angular/core';
3   import { HttpClient, HttpResponse, HTTP_INTERCEPTORS } from '@angular/common/http';
4   import { Observable } from 'rxjs';
5   import { map } from 'rxjs/operators';
6   import { Form } from '@angular/forms';
7
8
9   @Injectable({
10    providedIn: 'root'
11  })
12  export class DataFormService {
13
14    private apiPath: string = "api/DataForm";
15    jsonToForm: any;
16
17    constructor(private http: HttpClient) {}
18
19
20        getDataForm(): Observable<Form[]> {
21          return this.http.get(this.apiPath).pipe(
22            map(this.jsonToForm)
23          )
24        }
```

```
    getDataFormById(id: number): Observable<Form> {
      const url = '${this.api}/${id}'
      return this.http.get(url).pipe(
        map(this.jsonToForm)
      )
    }

  setDataForm(form: Form[]): Observable<Form[]> {
    return this.http.post(this.apiPath, form).pipe(
      map(this.jsonToForm)
    )
  }

    deleteDataForm(id: number): Observable<any> {
    const url = '${this.api}/${id}'
    return this.http.delete(url).pipe(
      map(() => null)
    )
  }
}
```

1.1 a) A classe 1 precisa de outra clase para funcionar. Precisa criar uma instância e passar automaticamente para a classe 1. (Ex.: Classe *curso.service.ts* fornece informações para *curso.component.ts* que tem uma dependência).

b) Falso. Pode ser usado um decarator ou uma função para indicar que um component ou classe possui uma dependência.

c) Verdadeiro. O injetor cria uma instância de serviço usando o provedor registrado e adiciona a ao injetor antes de retornar o serviço ao angular.

2. a) O angular utiliza http para se comunicar com um servidor, fazer download e upload de dados e acessar outros serviços back-end.

b) Solicitar objetos de respostas digitadas, tratamento de erros simplificado, recursos de testabilidade e interceptação de solicitação e resposta.

c) Fazer requests com objetos tipados; tratamento de erros nessas requisições; testes; interceptors nos requeste

d) Importar o HttpClientModule no módulo, depois importar o HttpClient e injetar sua dependencia.

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
```

e) Verdadeiro. É necessário importar o rxjs para todas as transações com o HttpClient.

f) request() - constrói um observable para uma solicitação Http genérica, que quando assinada, dispara a solicitação por meio da cadeia de interceptores registrados e no servidor.

delete() - constrói um observable que, quando inscrito, faz com que a solicitação configurada seja executada no servidor.

head() - constrói um observable que, quando inscrito, faz com que a solicitação configurada seja executada no servidor.

patch() - constrói um observable que, quando inscrito, faz com que a solicitação configurada seja executada no servidor.

post() - constrói um observable que, quando inscrito, faz com que a solicitação configurada seja executada no servidor.

put() - constrói um observable que, quando inscrito, faz com que a solicitação configurada seja executada no servidor.

g) Verdadeiro. É possível tipar o dado que está sendo recebido.

h) **observable<ArrayBuffer>** - retorna um observable de ArrayBuffer;
observable<Blob> - retorna um observable de um Blob;
observable<string> - retorna um observable de string;
observable<HttpEvent<ArrayBuffer>> - retorna um observable com todos os eventos do HttpEvents para o request, com ArrayBuffer no corpo do retorno;
observable<HttpEvent<Blob>> - retorna um observable com todos os eventos do HttpEvents para o request, com um Blob no corpo do retorno;
observable<HttpEvent<string>> - retorna um observable com todos os eventos do HttpEvents para o request, com um conteudo string no corpo do retorno;
observable<HttpEvent<Object>> - retorna um observable com todos os eventos do HttpEvents, com um Object no corpo do retorno;
observable<HttpEvent<T>> - retorna uma observable com todos eventos do HttpEvents para o request, com um corpo do retorno tipado;
observable<HttpResponse<ArrayBuffer>> - retorna um observable com todos os eventos HttpResponse para o request com um ArrayBuffer no corpo do retorno;
observable<HttpResponse<Blob>> - retorna um observable com todos os eventos do HttpEvents para o request, com o Blob no corpo do retorno;
observable<HttpResponde<string>> - retorna um observable com todos os eventos do HttpEvents para o request com conteúdo string no corpo do retorno;
observable<HttpResponde<Object>> - retorna um observable com todos os eventos do HttpEvents para o request com um Object no corpo do retorno;
observable<HttpResponde<T>> - retorna um observable com todos os eventos do HttpEvents para o request com um corpo do retorno tipado;
observable<Object> - retorna um observable com um object no corpo do request;
observable<T> - retorna um observable com um objeto tipado no corpo do request.

i) 1 - Respostas de informação (100 199);

2 - Respostas de sucesso (200 299);
3 Redirecionamentos (300 399);
4 - Erros do cliente (400 499);
5 - Erros do servidor (500 599).

j) DELETE - constrói um observável que, quando inscrito, faz com que a DELETE solicitação configurada seja executada no servidor.

```
deleteDataForm(id: number): Observable<any> {
  const url = '${this.api}/${id}'
  return this.http.delete(url).pipe(
    map(() => null)
  )
}
```

POST - constrói um observable que, quando inscrito, faz com que a solicitação POST configurada seja executada no servidor.O servidor responde com a localização do recurso substituído.

```
setDataForm(form: Form[]): Observable<Form[]> {
  return this.http.post(this.apiPath, form).pipe(
    map(this.jsonToForm)
  )
}
```

PUT - constrói um observable que, quando inscrito, faz com que a solicitação PUT configurada seja executada.

```
update(Form: Form): Observable<Form> {
  return this.http.put('${this.api}/${form.id}', Form).pipe(take(1));
}
```

GET - constrói um observable que, quando inscrito, faz com que a solicitação GET configurada seja executada no servidor.

```
getDataForm(): Observable<Form[]> {
  return this.http.get(this.apiPath).pipe(
    map(this.jsonToForm)
  )
}
```

k) Permitem que o cliente e o servidor passem informações adicionais com a solicitação ou a resposta Http.

l) Permitem interceptar as entradas e saídas de chamadas http na nossa aplicação, que estão usando HttpClient.

m) Fazer log de requisições, modificar cabeçalhos, modificar o próprio corpo da requisição, tratar erros de forma genérica.

3. a) É um paradigma de programação assíncrona preocupado com os fluxos de dados e a propagação da mudança. É uma biblioteca para programação reativa usando observables.

b) Promisses é um calculo que pode ou não eventualmente retornar um unico valor, já o observable pode retornar de forma sincrona ou assincrona de zero a valores infinitos quando invocado.

c) Quando há interação com o usuário. Cliques na tela, digitar valores em campos, por exemplo. Essas interações geram um alerta de que algo aconteceu. Quando passa a ter controle das ações realizadas, trabalhando no código em respostas a elas, você tem um programa baseado em eventos.

d) Quando se deseja enviar uma notificação de estado para algum lugar da sua aplicação, de maneira centralizada, pode-se utilizar o padrão de projeto observer.

e) Permite que sejam projetadas construções de looping que sejam mais flexiveis e efetivas em uma coleção de objetos. Essas soleções podem ser armazenadas como uma matriz ou algo mais complexo. Podemos precisar acessar os itens na coleção em uma determinada ordem.

f) Utilização de métodos que retornam novas instancias contendo novos valores, sem modificá-los.

g) Observable - São coleções que podem receber dados ao longo do tempo

```
import { Observable } from 'rxjs';

const observable = new Observable(subscriber => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
});

console.log('just before subscribe');
observable.subscribe({
  next(x) { console.log('got value ' + x); },
  error(err) { console.error('something wrong
  occurred: ' + err); },
  complete() { console.log('done'); }
});
console.log('just after subscribe');
```

Console:

just before subscribe	cursos.service.ts:16
got value 1	cursos.service.ts:18
got value 2	cursos.service.ts:18
got value 3	cursos.service.ts:18
just after subscribe	cursos.service.ts:22
got value 4	cursos.service.ts:18
done	cursos.service.ts:20

Observer - É um consumidor valores fornecidos por um observable. É um conjunto de chamadas de retorno, um para cada tipo de notificação entregue pelo Observable.

```
5   const observer = {
6     next: (x: string) => console.log('Observer got a next value: ' + x),
7     error: (err: string) => console.error('Observer got an error: ' + err),
8   };
9   observable.subscribe(x => console.log('Observer got a next value: ' + x));
```

Operators -

Subscription -

```
3   const subscription = observable.subscribe(x => console.log(x));
4   subscription.unsubscribe();
5
```

h) Ajax; bindCallback; bindNodeCallback; defer; empty; from; fromEvent; fromEventPattern; generate; interval; of; range; throwError; timer; iif.

i) Ajax - Cria um observable para uma solicitação Ajax com objetivo de solicitação com url, cabeçalhos, ou uma string para uma URL.

buscar dados da API

```
7   import { ajax } from 'rxjs/ajax';
8   import { map, catchError } from 'rxjs/operators';
9   import { of } from 'rxjs';
10
11  const obs$ = ajax.getJSON(`https://api.github.com/users?per_page=5`).pipe(
12    map(userResponse => console.log('users: ', userResponse)),
13    catchError(error => {
14      console.log('error: ', error);
15      return of(error);
16    })
17  );
18
```

from - Cria um observable a partir de um Array, um objeto semelhante a um array, uma Promisse, um objeto iterável ou um objeto semelhante a um observable.

converte uma matriz em um observable

```
11  import { from } from 'rxjs';
12
13  const array = [10, 20, 30];
14  const result = from(array);
15
16  result.subscribe(x => console.log(x));
17
```

fromEvent - cria um observable que emite eventos de um tipo específico provenientes de um determinado destino de evento.

cliques que acontecem no documento DOM

```
import { fromEvent } from 'rxjs';

const clicks = fromEvent(document, 'click');
clicks.subscribe(x => console.log(x));
```

generate - gera uma sequência observable executando um loop orientado por estado que produz os elementos da sequência, usando o escalonador especificado para enviar mensagens do observer.

produz sequencia de número

```
import { generate } from 'rxjs';

const result = generate(0, x => x < 3, x => x + 1, x => x);
result.subscribe(x => console.log(x));
```

of - converte os argumentos em uma sequência observável. Cada argumento se torna uma nextnotificação.

emitir os valores 10, 20, 30

```
import { of } from 'rxjs';

of(10, 20, 30)
.subscribe(
  next => console.log('next:', next),
  err => console.log('error:', err),
  () => console.log('the end'),
);
```

interval - cria um observable que emite números sequenciais a cada intervalo de tempo especificado, em um determinado SchedulerLike(interfaces do agendador)

emite números crescentes, um a cada segundo até o número 3.

```
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';

const numbers = interval(1000);
const takeFourNumbers = numbers.pipe(take(4));
takeFourNumbers.subscribe(x => console.log('Next: ', x));
```

throwError - cria um observable que criará um erro e um erro toda vez que for inscrito.

observable que criará um novo erro com um carimbo de data/hora e o registrará junto com a mensagem toda vez que você se inscrever nele.

```
import { throwError } from 'rxjs';

let errorCount = 0;

const errorWithTimestamp$ = throwError(() => {
    const error: any = new Error(`This is error number ${++errorCount}`
    error.timestamp = Date.now();
    return error;
});

errorWithTimestamp$.subscribe({
    error: err => console.log(err.timestamp, err.message)
});
errorWithTimestamp$.subscribe({
    error: err => console.log(err.timestamp, err.message)
});

// Logs the timestamp and a new error message each subscription;
```

time - usado para emitir uma notificação após um atraso.

```
import { timer } from 'rxjs';
import { concatMapTo } from 'rxjs/operators';

// This could be any observable
const source = of(1, 2, 3);

const result = timer(3000).pipe(
  concatMapTo(source)
)
.subscribe(console.log);
```

j) combineLatest; concat; forkJoin; merge; partition; race; zip.

k) concat - cria uma saida observable que emite sequencialmente todos os valores do primeiro observable fornecido e, em seguida, passa para o proximo.

```
import { concat, interval, range } from 'rxjs';
import { take } from 'rxjs/operators';

const timer = interval(1000).pipe(take(4));
const sequence = range(1, 10);
const result = concat(timer, sequence);
result.subscribe(x => console.log(x));

// results in:
// 0 -1000ms-> 1 -1000ms-> 2 -1000ms-> 3 -immediate-> 1 ... 10
```

l) buffer, bufferCount, bufferTime, bufferToggle, bufferWhen, concatMap, concatMapTo, exhaustMap, expand, groupBy, map, mapTo, mergeMap, mergeMapTo, mergeScan, pairwise, partition, pluck, scan, switchScan, switchMap, switchMapTo, window, windowCount, windowTime, windowToggle, windowWhen.

m) **concatMap** - mapeia cada valor para um observable e, em seguida, nivela todos esses observaveis internos usando concatAll.

```
import { concat, interval, range } from 'rxjs'

import { concatMap, take } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const result = clicks.pipe(
  concatMap(ev => interval(1000).pipe(take(4)))
);
result.subscribe(x => console.log(x));

// Results in the following:
// (results are not concurrent)
// For every click on the "document" it will emit values 0 to 3 spaced
// on a 1000ms interval
// one click = 1000ms-> 0 -1000ms-> 1 -1000ms-> 2 -1000ms-> 3
```

para cada evento de clique, marca a cada segundo de 0 a 3 sem simultaneidade.

map - aplica uma determinada função a cada valor emitido pela fonte observable e emite os valores resultantes como um observable.

```
import { fromEvent } from 'rxjs';
import { map } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const positions = clicks.pipe(map(ev => ev.clientX));
positions.subscribe(x => console.log(x));
```

mapTo - emite o valor constante fornecido na saída observable sempre que a fonte observable emite um valor.

```
import { mapTo } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const greetings = clicks.pipe(mapTo('Hi'));
greetings.subscribe(x => console.log(x));
```

mapeie cada clique para a string ‘Hi’.

mergeMap - projeta cada valor de origem para um observable que é mesclado na saída observable.

```
import { mergeMap } from 'rxjs/operators';

const letters = of('a', 'b', 'c');
const result = letters.pipe(
  mergeMap(x => interval(1000).pipe(map(i => x+i))),
);
result.subscribe(x => console.log(x));

// Results in the following:
// a0
// b0
// c0
// a1
// b1
// c1
// continues to list a,b,c with respective ascending integers
```

mapeie e nivele cada letra para um tique-taque observable a cada 1 segundo.

mergeMapTo - projeta cada valor de origem para o mesmo observable, que é mesclado vários vezes na saída observable.

```
import { mergeMapTo } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const result = clicks.pipe(mergeMapTo(interval(1000)));
result.subscribe(x => console.log(x));
```

para cada evento de clique, inicie um intervalo observable marcando a cada 1 segundo.

switchMap - projeta cada valor de origem para um observable que é mesclado na saída observable, emitindo valores apenas do observable projetado mais recentemente.

```
import { switchMap } from 'rxjs/operators';

const switched = of(1, 2, 3).pipe(switchMap((x: number) => of(x, x ** 2, x ** 3)))
;
switched.subscribe(x => console.log(x));
// outputs
// 1
// 1
// 1
// 2
// 4
// 8
// ... and so on
```

switchMapTo - projeta cada valor de origem para o mesmo observable, que é achatado várias vezes na saída observable.

```
import { switchMapTo } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const result = clicks.pipe(switchMapTo(interval(1000)));
result.subscribe(x => console.log(x));
```