Item 1: Services

a) A respeito de services, responda:

I. Qual o propósito de services no angular?

O principal objetivo de um serviço é organizar e compartilhar lógica de negócios, modelos ou dados e funções com diferentes componentes de uma aplicação Angular.

II. Porque no Angular há uma distinção entre components e services?

Para ajudar a dividir o aplicativo da web em unidades lógicas pequenas e diferentes que podem ser reutilizadas quando necessário em outros componentes por meio da injeção de dependência.

III. Como as services são disponibilizadas em um component?

Declarando o atributo ‘providers’ no @ngModule, informando assim que o serviço esta disponível para uso.

VI. Que tipo de tarefas uma service pode ter?

Validar informações, fazer interações com o servidor.

V. Verdadeiro ou Falso. Uma service precisa obrigatoriamente estar em pelo menos um módulo? Justifique sua escolha.

Falso.

Ele pode ser disponibilizado direto no componente utilizando a instrução “ provideIn: 'root' “.

VI. Verdadeiro ou Falso. Uma service é do tipo de padrão de projeto Singleton? Justique sua escolha.

Verdadeiro.

O Angular trabalha por padrão de projeto Singleton, a instancia do provider é definida como root, ficando disponível para toda aplicação.

VII. Escreva uma service que deverá ter os seguintes métodos e em cada um deverá escrever uma implementação básica (pode ser usado Arrays):

- getUsers( ): { }
- getUserById(userId: number): { }
- setUsers(users: any): { }
- deleteUser(userId: number): { }

```
import { User } from '../model/user';
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
```

```
@Injectable({
  providedIn: 'root'
})
export class UserService {

  private readonly url = 'http://localhost:3000/user';

  constructor(private http: HttpClient) { }

  getUsers(): Observable<User>{
    return this.http.get(this.url);
  }

  getUserById(userId: number): Observable<User>{
    const urlById = `${this.url}/${userId}`;
    return this.http.get(urlById)
  }

  setUsers(user: User): Observable<User>{
    return this.http.post(this.url, user)
  }

  deleteUser(userId: number): Observable<any>{

    const urlDel = `${this.url}/$(userId)`;
    return this.http.delete(urlDel)
  }
}
```

b) A respeito de services, responda:

I. O que é Injeção de Dependências?

É quando uma classe ou objeto chama uma classe externa para complementar seu trabalho, utilizado quando é necessário manter um baixo nível de acoplamento entre diferentes partes do sistema.

II. Verdadeiro ou Falso. Injeção de Dependências pode ser apenas de services? Justifique sua escolha.

Falso.

Pode ser usado um decorador ou uma função para indicar que um component ou classe possui uma dependência.

III. Verdadeiro ou Falso. O Angular quando vai instanciar uma classe de componente verifica se as injeções de dependência já não estão sendo usadas?
Justifique sua escolha.

Verdadeiro.

Ele tem em sua própria estrutura um recurso integrado que verifica se as dependências foram providas e se estão disponíveis no injetor.

c) HTTP Client:

I. O que é o protocolo HTTP?

HTTP é um protocolo de transferência que possibilita que as pessoas que inserem a URL do seu site na Web possam ver os conteúdos e dados que nele existem. A sigla vem do inglês Hypertext Transfer Protocol.

II. Cite outros tipos de comunicação com o backend e faça um breve resumo de cada.

FTP (File Transfer Protocol) - protocolo de transferência bruta de arquivos, pode-se enviar um grande volume de dados, vários diretórios ao mesmo tempo e ainda oferece segurança nessa transferência.

TCP (Transmission Control Protocol) - Com o TCP temos uma conexão entre um ponto e outro, comumente chamados de servidor e cliente. Permite o envio simultâneo de dados de ambos os pontos ao outro, durante todo o fluxo de comunicação dando maior confiabilidade.

UDP (sigla para User Datagram Protocol) - velocidade alta de transferência, porém com baixa confiabilidade, ou seja, dados podem se perder no meio do caminho.

III. Que recursos o HTTP nos fornece?

Solicitar o tipo de objeto de resposta do backend, fazer a manipulação de erros, realizar testes, usar interceptadores em requisições e resposta.

VI. Para usar o HttpClient no Angular, como devemos fazer sua importação e injeção?

Deve ser importado o HttpClientModule no AppModule e no importor @NgModule. Após isso é necessário importar no topo do service e fazer a injeção no construtor do tipo HttpClient.

V. Verdadeiro ou Falso. O HttpClient pode ser usado com RxJS? Justifique sua escolha.

Verdadeiro.

O HttpClient usa Observables nas comunicações e o RxJs o auxilia nesse processo.

VI. Cite os principais métodos HTTP e faça um breve resumo de cada.

- GET - solicita a representação de um recurso específico. Requisições utilizando o método GET devem retornar apenas dados.

- HEAD - solicita uma resposta de forma idêntica ao método GET, porém sem conter o corpo da resposta.
- POST - é utilizado para submeter uma entidade a um recurso específico, frequentemente causando uma mudança no estado do recurso ou efeitos colaterais no servidor.
- PUT - substitui todas as atuais representações do recurso de destino pela carga de dados da requisição.
- DELETE - remove um recurso específico.
- CONNECT - estabelece um túnel para o servidor identificado pelo recurso de destino.
-
- OPTIONS - é usado para descrever as opções de comunicação com o recurso de destino.
- TRACE - executa um teste de chamada loop-back junto com o caminho para o recurso de destino.
- PATCH - é utilizado para aplicar modificações parciais em um recurso.

VII. Verdadeiro ou Falso. Com o protocolo HTTP é possível dizer/setar o tipo de resposta do servidor backend? Justifique sua escolha.

Verdadeiro.

É possível tipar o dado que está sendo recebido.

VIII. Cite os tipos de retorno que uma requisição HTTP pode ter? Faça um breve resumo de cada.

- Respostas de informação (1XX) - o servidor recebeu e entendeu a requisição;
- Respostas de sucesso (2XX) - o servidor recebeu, aceitou e processou a requisição;
- Redirecionamentos (3XX) - a resposta da requisição direciona para outro caminho ou pede para o cliente seguir outra rota;
- Erros do cliente (4XX) - o servidor não conseguiu processar a requisição pois o cliente a fez de forma incorreta;
- Erros do servidor (5XX) - o servidor não conseguiu processar a requisição por erro dele próprio.

IX. Cite os principais status de uma requisição HTTP e faça um breve resumo de cada.

- 200 - tudo certo com a requisição;
- 201 - A requisição foi bem sucedida e um novo recurso foi criado como resultado. Esta é uma típica resposta enviada após uma requisição POST;
- 301 - Esse código de resposta significa que a URI do recurso requerido mudou. Provavelmente, a nova URI será especificada na resposta;
- 400 - Essa resposta significa que o servidor não entendeu a requisição pois está com uma sintaxe inválida;

- 404 - O servidor não pode encontrar o recurso solicitado. Este código de resposta talvez seja o mais famoso devido à frequência com que acontece na web;
- 500 - O servidor encontrou uma situação com a qual não sabe lidar.

X. Faça um exemplo de chamadas do tipo GET, POST, PUT, DELETE.

GET

```
getById(id: number): Observable<Category>{
        const url  = `${this.apiPath}/${id}`
        return this.http.get(url).pipe(
                catchError(this.handleError),
                map(this,jsonDataToCategory)
        )
}
```

POST

```
update(category: Category): Observable<Category>{
        const url  = `${this.apiPath}/${category.id}`

        return this.http.post(url, category).pipe(
                catchError(this.handleError),
                map(() => category)
        )
}
```

PUT

```
put(category: Category): Observable<Category>{
const url  = `${this.apiPath}/${category.id}`

        return this.http.put(url, category).pipe(
                catchError(this.handleError),
                map(() => category)
        )
}
```

DELETE

```
delete(id: number): Observable<any>{
const url  = `${this.apiPath}/${id}`

        return this.http.delete(url).pipe(
                catchError(this.handleError),
                map(() => null)
        )
```

}

XI. Para que serve o cabeçalho em uma requisição HTTP?

Os cabeçalhos HTTP permitem que o cliente e o servidor passem informações adicionais ou de segurança.

XII. O que é um Interceptor e quais as suas aplicações?

Um Interceptor em Angular é um pattern simples que nos permite interceptar, tratar e gerenciar requisições HTTP, antes mesmo delas serem enviadas ao servidor

XIII. Quais cenários mais comuns podemos usar Interceptors?

Os mais comuns são fazer uma autenticação e login, adaptar informações e tratar erros do servidor.

d) RxJS:

I. O que é o RxJS?

Rxjs é uma biblioteca JavaScript que traz o conceito de programação reativa para a Web.

II. Qual a diferença de Promises e Observables?

Promises:

- são ansiosas (eager));
- sempre são assíncronas;
- sempre retornam o mesmo valor.

Observables:

- são preguiçosos (lazy);
- podem ser tanto síncronas quanto assíncronas;
- pode retornar um fluxo de valores, de nenhum a vários.

III. O que significa ser baseado em eventos?

Um evento é toda ação que gera mudança de estado como, por exemplo, uma atualização de e-mail no seu cadastro, o clique do mouse e um log que foi gerado. Assim, todas as ações do sistema podem ser eventos.

VI. O que é o padrão de projetos Observer?

É um projeto que oferece um modelo de assinatura no qual os objetos se inscrevem em um evento e são notificados quando o evento ocorre.

V. O que é o padrão de projetos Iterator?

É um padrão de projeto comportamental que permite a passagem sequencial através de uma estrutura de dados complexa sem expor seus detalhes internos.

VI. O que é programação funcional com coleções?

É um conceito de programação utilizando métodos que retornam novas instâncias contendo novos valores, porém sem modificá-los.

VII. Quais os conceitos básicos do RxJS para eventos assíncronos? Descreva todos e dê um exemplo de como utilizá-lo.

- **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);
        set Timeout(()  => {
                subscriber.next(4);
                subscriber.complete();
        }, 1000);
});

console.log(‘Após incrição‘);
observable.subscribe({
        next(x) { console.log(‘valor: ’ + x);
        },
        error(err) { console.error(‘Algo errado aconteceu: ‘ + err);
        },
        complete() { console.log(‘Feito’);
        }
});

console.log(‘Após incrição‘);
```

- **Observer:** é uma coleção de callbacks que sabem ouvir os valores entregues pelo Observable.

```
const observer = {
        next: (x: string) => console.log(‘Observer tem valor: ‘ +  x),
        error: (err: string) => console.error(‘Observer tem um erro: ‘ +  err),
        complete: () => console.log(‘Observer completo’),
};
observable.subscribe(observer);
```

- **Subscription:** representa a execução de um Observable, é principalmente útil para cancelar a execução.

```
const subscription = observable.subcribe(x => console.log(x));
subscription.unsubscribe();
```

- **Operators**: São funções que permitem editar, iterar e criar valores de uma observable.

- **Subject**: semelhante a observable, porém são enviados a múltiplos observers.

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

const subject = new Subject<number>();

subject.subscribe({
        next: (v) => console.log( 'ObserverA: ${v}' )
});
subject.subscribe({
        next: (v) => console.log( 'ObserverB: ${v}' )
});
subject.next (1);
subject.next (2);
```

- **Schedulers:** são despachantes para a simultaneidade de controle centralizado, permitindo-nos para coordenar quando computação acontece .

```
import { Observable, asyncScheduler } from 'rxjs';
import { observeOn } from 'rxjs/operators';

const observable  = new Observable((observer)  => {
        observer.next(1);
        observer.next(2);
        observer.next(3);
        observer.complete();
}).pipe(
        observeon(asyncScheduler)
);

console.log( 'Após incrição' );
observable.subscribe({
        next(x) { console.log('valor: ' + x);
        },
        error(err) { console.error('Algo errado aconteceu: ' + err);
        },
```

```
        complete() { console.log('Feito');
        }
});

console.log('Após incrição');
```

VIII. Quais são os operadores de criação?
ajax; bindCallback, bindNodeCallback, defer, empty, from, fromEvent, fromEventPattern, generate, of, interval, throwError, timer, range, iif.

IX. Ainda dentro de operadores de criação, explique melhor e dê um exemplo para os seguintes operadores:

- **Ajax:** Cria uma observable para uma solicitação ajax com objetivo de solicitação URL, Headers ou uma string para uma URL.

Ex:

```
import { ajax } from 'rxjs/ajax' ;
import { map , catchError } from 'rxjs/operators';
import { of } from 'rsjs;

const obs$ = ajax(`https://api.github.com/users?per_page=5`).pipe(
        map( userResponse => console.log( 'users: ', userResponse)),
                catchError(error => {
                console.log( 'error: ' , error);
                return of (error);
        })
);
```

- **from:** Converte vários outros objetos e tipos de dados em observáveis.

Ex:

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

const array = [10, 20, 30];
const result = from(array);

result.subscribe( x => console.log(x));
```

- **fromEvent:** cria um observable que emite eventos de um tipo específico provenietes de um determinado event target.

Ex:

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

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

- **generate:** permite que você crie um fluxo de valores gerado com um loop.

Ex:

```
importe { generate } from ‘rxjs’ ;

const result = generate(
2,
x => x <= 8,
x => x + 3
)
```

- **of:** converte os argumentos em uma sequência em observable.

Ex:

```
import { from } from ‘rxjs’ ;

const array = [10, 20, 30 ];
const result = of(array);

result.subscribe(x => console.log(x));
```

- **interval:** retorna um Observable que emite uma sequência infinita de inteiros ascendentes, com um intervalo de tempo constante de sua escolha entre essas emissões.

Ex:

```
import { interval } from ‘rxjs’ ;
import { take } from ‘rxjs/operators’ ;

const numbers = interval(1000);
const takeFourNumbers = numbers.pipe(take(4));

takeFourNumbers.subscribe( x => console.log(‘Próximo: ‘ , x ));
```

- **throwError:** Esta função de criação é útil para criar um observável que criará um erro e um erro toda vez que for inscrito.

Ex:

```
import { throwError } from ‘rxjs’ ;

const  source = throwError( ‘Isso é um erro!’ );

const subscribe = source.subscribe({
        next: val => console.log(val),
        complete: () => console.log( ‘Completo’ ),
        error: val => console.log( `Erro: ${val}`)
});
```

- **timer:** Emite números em sequência de uma observable durante um período de tempo especificado..

Ex:

```
import { timer } from ‘rxjs’ ;

const source = timer(1000, 2000);
const subscribe = source.subscribe(val => console.log(val));
```

X. Quais os operadores de criação de associação?

combineLatest, concat, forkJoin, merge, partition, race, zip.

XI. Ainda dentro de operadores de criação de associação, explique melhor e dê um exemplo para os seguintes operadores:

- concat - cria uma nova Observable que emitirá em ordem os valores da primeira observable e assim sequencialmente.

ex:

```
import { of, concat } from ‘rxjs ‘ ;
const result = concat(
    of(1, 2, 3),
    of(4, 5, 6),
    of(7, 8, 9)
)

result.subscribe(x => console.log(x));
```

- forkJoin - emite o valor de cada uma das observables quando elas são finalizadas.

ex:

```
import { ajax } from ‘rxjs/ajax’ ;
import { forkJoin } from ‘rxjs’ ;

forkJoin({
    google: ajax.getJSON( ‘https://api.github.com/users/google’ ),
    microsoft: ajax.getJSON( ‘https://api.github.com/users/microsoft’ ),
    users: ajax.getJSON( ‘https://api.github.com/users’ )
}).subscribe(console.log);
```

- merge - organiza múltiplas observables em uma única observable.

ex:

```
import { mapTo } from ‘rxjs/operators’
import { interval, merge } from rxjs’ ;

const primeiro = interval(2500)
const segundo = interval(2000)
```

```
const terceiro = interval(1500)
const quarto = interval(1000)

const exemplo = merge(
        primeiro.pipe(mapTo( ‘Primeiro’ )),
        segundo.pipe(mapTo( ‘Segundo!’ )),
        terceiro.pipe(mapTo( ‘Terceiro’ )),
        quarto.pipe(mapTo( ‘Quarto!’ )),
);
const subscribe = exemplo.subscribe( val => console .log(val));
```

XII. Quais os operadores de transformação?

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

XIII. Ainda dentro de operadores de transformação, explique melhor e dê um exemplo para os seguintes operadores:

- concatMap - mapeia cada de origem de uma observable, mesclando ela a uma observable de saída.

ex:

```
import { of } from ‘rxjs’ ;
import { concatMap, delay, mergeMap } from ‘rxjs/operators’ ;

const source = of (2000, 1000);
const example = source.pipe(
        concatMap(val => of( `Delayed by> ${val}ms` ).pipe(delay(val)))
);

const.subscribe = example.subscribe(val = > console.log( `With concatMap: ${val}`)
);
```

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

Ex:

```
import { from } from ‘rxjs’ ;
import { map } from ‘rxjs/operators’ ;

const source = from[1, 2, 3, 4,5]);
const exemplo = source.pipe(map(val => val +10));

const subscribe = exemplo.subscribe(val => console.log(val));
```

- mapTo - emite o valor constante fornecido na saída Observable sempre que a fonte Observable emite um valor.
Ex:

```
import { interval } from ‘rxjs’ ;
import { mapTo } from ‘rxjs/operators’ ;

const source = interval(2000);
const exemplo = source.pipe(mapTo(‘ Olá Mundo! ‘));
const subscribe = exemplo.subscribe(val => console.log(val));
```

- mergeMap - Projeta cada valor de origem para um observable que é mesclado na saída observable.
Ex:

```
import { of } from ‘rxjs’ ;
import { mergeMap, map} from ‘rxjs/operators’ ;

const obsOne = of(‘a’, ‘b’ , ‘c’)
const obsTwo = of(1, 2, 3)

const result = obsOne.pipe(mergeMap( x=> {
        resturn obsTwo.pipe(map( y=> x + y))
}))

const subscribe = result.subscribe(val => console.log(val));
```

- mergeMapTo - Projeta cada valor de origem para um mesmo observable que é mesclado na saída observable.
Ex:

```
import { interval, fromEvent } from ‘rxjs’ ;
import { mergeMapTo } from ‘rxjs/operators’ ;

const clicks = fromEvent(document, ‘clique’);
const result = clicks.pipe(mergeMapTo(interval(1000)));

const subscribe = result.subscribe(val => console.log(val));
```

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

```
import { interval, fromEvent } from ‘rxjs’ ;
import { switchMap } from 'rxjs/operators' ;
```

```
fromEvent(document, ‘clique’)
  .pipe(
    switchMap(() => interval(1000))
  )
  .subscribe(console.log);
```

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

Ex:

```
import { fromEvent, interval } from ‘rxjs’ ;
import { switchMapTo } from ‘rxjs/operators’ ;

const clicks = fromEvent(document, ‘clique’);
const result = clicks.pipe(switchMapTo(interval(1000)));
result.subscribe(val  => console.log(val));
```