Cache

O que é Cache? De acordo com a Wikipedia:

> *Na área da computação, cache é um dispositivo de acesso rápido, interno a um sistema, que serve de intermediário entre um operador de um processo e o dispositivo de armazenamento ao qual esse operador acede. A principal vantagem na utilização de um cache consiste em evitar o acesso ao dispositivo de armazenamento - que pode ser demorado -, armazenando os dados em meios de acesso mais rápidos.*

No âmbito de uma aplicação web, esse conceito pode ser aplicado em várias etapas, as quais serão discutidas individualmente neste material. São elas:

- Armazenamento em memória client-side do navegador do usuário.
- Uso de cabeçalhos de manipulação de cache suportados pela própria especificação HTTP. Este pode ser entendido como cache HTTP client-side.
- Na camada de modelo (ou qualquer outra camada do back-end), através do uso de *memoização*.

Memória client-side (navegador)

Imagine que sua aplicação tenha um processamento considerável para mostrar uma lista de informações na página inicial. Não há a necessidade de atualizar esses dados a cada vez que o usuário acessa essa tela, logo uma estratégia de cache pode poupar muitos recursos dos seus servidores. A biblioteca `rxjs` oferece uma ótima API para isso, baseada no uso da classe `ReplaySubject` (opcionalmente através do método fábrica `shareReplay`).

Para implementar um exemplo dessa estratégia, comece criando uma simples API que retorna uma lista de nomes depois de 3 segundos (simulando um processamento pesado). Inicie um projeto npm e adicione o express nele:

```
npm init
npm i express cors
```

Implemente agora o arquivo `index.js`:

```
const express = require('express');
const cors = require('cors');


const app = express();


app.use(cors());


app.get('/nomes', (_, res) => {
    setTimeout(() => {
        res.send([ 'Pessoa 1', 'Pessoa 2', 'Pessoa 3' ]);
    }, 3000);
});


app.listen(3000);
```

Crie agora um projeto Angular em outro diretório (lembre-se de optar pela adição do módulo de roteamento):

```
ng new
```

Navegue até o diretório do projeto e gere dois componentes que farão o papel de páginas:

```
ng generate component home
ng generate component outra-pagina
```

Gere também um serviço que encapsulará a chamada para a API de nomes:

```
ng generate service nomes
```

Edite o arquivo `app.component.html` deixando apenas a tag `router-outlet` :

```
<router-outlet></router-outlet>
```

Edite o arquivo `app-routing.module.ts` configurando as rotas para os componentes correspondentes:

```
// ...

import { HomeComponent } from './home/home.component';
import { OutraPaginaComponent } from './outra-pagina/outra-pagina.component';

const routes: Routes = [
  {
    path: '',
    component: HomeComponent
  },
  {
    path: 'outra',
    component: OutraPaginaComponent
  }
];

// ...
```

Edite agora o arquivo `home.component.html` , adicionando link para a outra página e mostrando a lista de nomes:

```
<p *ngIf="!nomes">Carregando...</p>
<ul *ngIf="nomes">
  <li *ngFor="let nome of nomes">{{nome}}</li>
</ul>

<p>
  <a [routerLink]="['/outra']">Ir para a outra página</a>
</p>
```

E também o arquivo `home.component.ts` , para carregar a lista de nomes:

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

import { NomesService } from '../nomes.service';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {

  nomes: string[];

  constructor(private nomesService: NomesService) { }

  ngOnInit() {
    this.nomesService
      .getNomes()
      .subscribe(nomes => this.nomes = nomes);
  }

}
```

Registre o módulo `HttpClient` no arquivo `app.module.ts` e implemente o método `getNomes` no arquivo `nomes.service.ts`:

```
// ...
import { HttpClientModule } from '@angular/common/http';
// ...

@NgModule({
  // ...
  imports: [
    // ....
    HttpClientModule
  ],
  // ...
})
export class AppModule { }
```

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

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

  constructor(private http: HttpClient) { }

  getNomes(): Observable<string[]> {
```

```
    return <Observable<string[]>>this.http
      .get('http://localhost:3000/nomes');
  }

}
```

Por fim adicione um link na página `outra-pagina.component.html` para retornar à página inicial:

```
<p>outra-pagina works!</p>

<p>
  <a [routerLink]="['/']">Retornar à página inicial.</a>
</p>
```

Teste a aplicação deixando a API no ar e executando `ng serve`. Note que a cada retorno na página inicial são mais 3s de espera. Veja como é fácil resolver essa questão usando cache no serviço `NomesService`:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { shareReplay } from 'rxjs/operators';

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

  private nomes: Observable<string[]>;

  constructor(private http: HttpClient) { }

  getNomes(): Observable<string[]> {
    if (!this.nomes) {
      this.nomes = <Observable<string[]>>this.http
        .get('http://localhost:3000/nomes')
        .pipe(shareReplay(1));
    }
    return this.nomes;
  }

}
```

Para adicionar um tempo de vida para este cache, por exemplo 10 segundos, basta usar um `setTimeout`, limpando a propriedade `nomes`:

```
// ...
export class NomesService {

  // ...

  getNomes(): Observable<string[]> {
    if (!this.nomes) {
```

```
      this.nomes = <Observable<string[]>>this.http
        .get('http://localhost:3000/nomes')
        .pipe(shareReplay(1));
      setTimeout(() => {
        this.nomes = null;
      }, 10000);
    }
    return this.nomes;
  }

}
```

Cabeçalhos HTTP (também client-side)

Além da utilização de estruturas JavaScript, é possível utilizar o suporte existente no próprio protocolo HTTP para cache. Inspecione os cabeçalhos de resposta de uma requisição para `GET /nomes`, e veja que o cabeçalho `Cache-Control` não está ali. Por padrão, na ausência desse, o navegador vai usar uma série de heurísticas para definir o tempo de vida do resultado, o que normalmente resulta em requisitar novamente esse conteúdo sempre que solicitado. Ajuste isso no arquivo `index.js` do backend para fornecer um tempo de vida explícito (10 segundos) para a resposta da requisição:

```
app.get('/nomes', (_, res) => {
    setTimeout(() => {
        res.header('Cache-Control', 'public, max-age=10');
        res.send([ 'Pessoa 1', 'Pessoa 2', 'Pessoa 3' ]);
    }, 3000);
});
```

Desabilite o cache do frontend implementado na seção anterior (retornando o Observable original diretamente do service ao invés do cache) e teste novamente, navegando da página inicial para a outra página e vice-versa. Note que o Chrome Dev Tools possui um checkbox para desabilitar completamente o cache do navegador, assegure-se que ele esteja desabilitado para efetuar este teste.

Existe um middleware para Express que ajuda na definição de estratégias de cache para cada endpoint, sem a necessidade de manusear os headers diretamente. Esse middleware é o `apicache`. Para utilizá-lo, instale-o no projeto do backend e configure o middleware em cada roteamento conforme apropriado:

```
npm i apicache
```

```
// ...
const apicache = require('apicache');

// ...

const cache = apicache.middleware;

// ...

app.get('/nomes', cache('10 seconds'), (_, res) => {
    console.log('Executando a busca de nomes real...');
```

```
    setTimeout(() => {
        // res.header('Cache-Control', 'public, max-age=10');
        res.send([ 'Pessoa 1', 'Pessoa 2', 'Pessoa 3' ]);
    }, 3000);
});
// ...
```

Memoização server-side

Uma das maneiras mais fáceis de explicar o conceito de memoização é, sem dúvida, a aplicação dela no cálculo do enésimo número da sequência de fibonacci. Considere a seguinte implementação Node:

```
const process = require('process');

const start = process.hrtime();
console.log(fib(43));
const diff = process.hrtime(start);
console.log(`Tempo de execução: ${diff[0]}s ${diff[1]}ms`);

function fib(n) {
    if (n <= 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

Na máquina onde esse código foi escrito, ele demora em média 4s para calcular o 43º número da sequência. Por qual motivo existe essa demora? Repare que o algoritmo calcula várias vezes o mesmo valor, em uma estrutura de árvore:

```
fib(5)
  fib(4)
    fib(3)
      fib(2)
        fib(1)
        fib(0)
      fib(1)
    fib(2)
      fib(1)
      fib(0)
  fib(3)
    fib(2)
      fib(1)
      fib(0)
    fib(1)
```

Veja quantas vezes o `fib(2)` foi calculado, por exemplo. É fácil ver que conforme o número aumenta, essa situação só piora.

Mas onde a memoização entra nesse cenário? Imagine investir um pouco de *espaço em memória* para *memorizar* os cálculos e *reutilizá-los* quando solicitados novamente no futuro. Veja:

```
const process = require('process');

const memory = {};

const start = process.hrtime();
console.log(fib(1000));
const diff = process.hrtime(start);
console.log(`Tempo de execução: ${diff[0]}s ${diff[1]}ms`);

function fib(n) {
    if (!memory[n]) {
        if (n <= 1) {
            memory[n] = n;
        } else {
            memory[n] = fib(n - 1) + fib(n - 2);
        }
    }
    return memory[n];
}
```

Note que agora o limitador passa a ser o tamanho da pilha de chamadas, e não mais o tempo de processamento.

Repare que implementar essa estrutura é um processo repetitivo, e por isso é natural que exista uma biblioteca para facilitá-lo. No caso do Node uma delas é a `memoizee` [1]. Use-a no projeto da API de back-end para implementar um cache de memoização. Comece adicionando a biblioteca no projeto:

```
npm i memoizee
```

Ajuste agora o `index.js` para desabilitar o cache via `apicache` e implementar um cache de memoização:

```
const express = require('express');
const cors = require('cors');
//const apicache = require('apicache');
const memoizee = require('memoizee');


const app = express();
// const cache = apicache.middleware;


app.use(cors());


app.get('/nomes'/*, cache('10 seconds')*/, (_, res) => {
    getNomes().then(nomes => {
        res.send(nomes);
    });
});

const getNomes = memoizee(() => {
    return new Promise((resolve, _) => {
```

```
        console.log('Executando a busca de nomes real...');
        setTimeout(() => {
            resolve([ 'Pessoa 1', 'Pessoa 2', 'Pessoa 3' ]);
        }, 3000);
    });
}, { maxAge: 10000, promise: true });


app.listen(3000);
```

[1] https://github.com/medikoo/memoizee