3 dicas para melhorar a performance da sua aplicação em Angular

Autor: João Paulo Hotequil

Repositório com códigos: https://github.com/hotequil/angular-improve-performance

Versão do Angular: 12.2.0

Introdução:

Olá, hoje vou demonstrar alguns exemplos simples, mas que podem se tornar gargalos e um peso no seu sistema feito em Angular ao passar do tempo se não forem implementados. Essas dicas são saídas que encontrei estudando e sofrendo muito, escrevi muitos códigos não performáticos, ainda devo escrever…, mas buscamos a evolução a cada dia, isso que importa, então, quero compartilhar com você! Irei mostrar todo código em imagens, menos as importações para não ocupar muito espaço e também o CSS porque não é o objetivo desse texto. Esse artigo é destinado principalmente a desenvolvedores que trabalham ou estão iniciando com Angular.

1) Use pipes:

Em uma aplicação geralmente temos que atualizar ou formatar valores no template (HTML) sempre que a sua variável muda, por exemplo, exibir strings concatenadas, calcular tamanho de elementos e outros. Quando tinha esse tipo de problema, eu pensava em já colocar uma função no template. Trouxe um exemplo de cálculo de saldo de uma lista de clientes, que está vazia por hora e depois recebe um valor após terminada a função assíncrona de *get*, que traz clientes aleatórios a cada chamada com nome e saldo diferentes, aproveitei e executei um *detectChanges* a cada 1 segundo para verificar mudanças no componente. Usei também um decorator chamado *counter* que criei para mostrar quantas vezes essa função é executada.

home.component.ts:

Criação da função *calculateSumOfCustomerBalances*.

```ts
@Component({
    selector: 'app-home',
    templateUrl: './home.component.html',
    styleUrls: ['./home.component.scss']
})
export class HomeComponent implements OnInit, OnDestroy{
    customers: Customer[] = [];

    private intervalId!: number;

    constructor(private customersService: CustomersService, private changeDetectorRef: ChangeDetectorRef){}

    async ngOnInit(): Promise<void>{
        this.customers = await this.customersService.get();
        this.intervalId = window.setInterval(() => this.changeDetectorRef.detectChanges(), 1000);
    }

    @counter("Function")
    calculateSumOfCustomerBalances(): number{
        return this.customers.reduce((accumulator, item) => accumulator + item.balance, 0);
    }

    ngOnDestroy(): void{
        clearInterval(this.intervalId);
    }
}
```

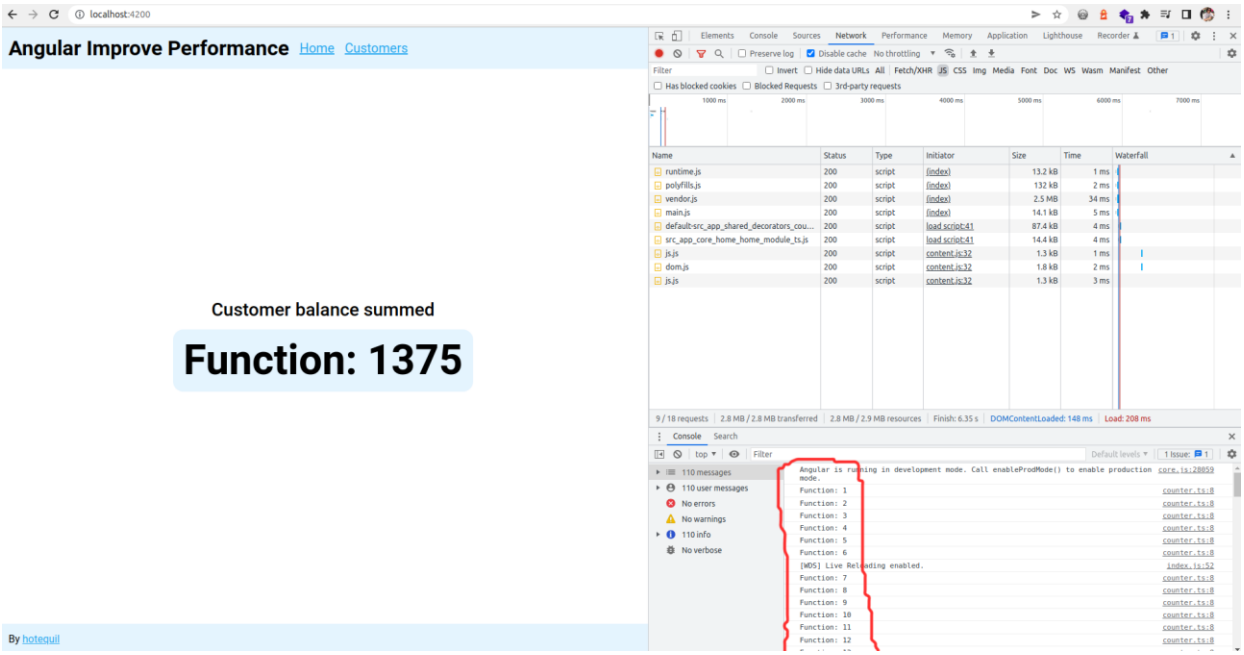
home.component.html:

Colocada função no HTML para exibir o valor monetário do saldo dos clientes.

```html
<section class="home">
    <h2 class="home__title">Customer balance summed</h2>
    <span class="home__badge">Function: {{ calculateSumOfCustomerBalances() }}</span>
</section>
```

Browser:

Se você perceber, tivemos inúmeras chamadas, isso acontece porque o Angular chama as funções que estão no template sempre que há uma alteração na tela, esse foi um exemplo pequeno, mas imagine várias funções no HTML sendo chamadas simultaneamente com algoritmos pesados, ficaria extremamente lento seu site, falo por experiência própria.



Porém, agora vamos utilizar o pipe, nosso pipe tem a mesma lógica e objetivo, somar o saldo de todos os clientes.

home.component.ts:

Mesma ideia de pegar a lista de clientes pelo service e gerar detecção no componente a cada 1 segundo.

```
@Component({
    selector: 'app-home',
    templateUrl: './home.component.html',
    styleUrls: ['./home.component.scss']
})
export class HomeComponent implements OnInit, OnDestroy{
    customers: Customer[] = [];

    private intervalId!: number;

    constructor(private customersService: CustomersService, private changeDetectorRef: ChangeDetectorRef){}

    async ngOnInit(): Promise<void>{
        this.customers = await this.customersService.get();
        this.intervalId = window.setInterval(() => this.changeDetectorRef.detectChanges(), 1000);
    }

    ngOnDestroy(): void{
        clearInterval(this.intervalId);
    }
}
```

home.component.html:

Pipe na view.

```html
<section class="home">
    <h2 class="home__title">Customer balance summed</h2>
    <span class="home__badge">Pipe: {{ customers | calculateSumOfCustomerBalances }}</span>
</section>
```

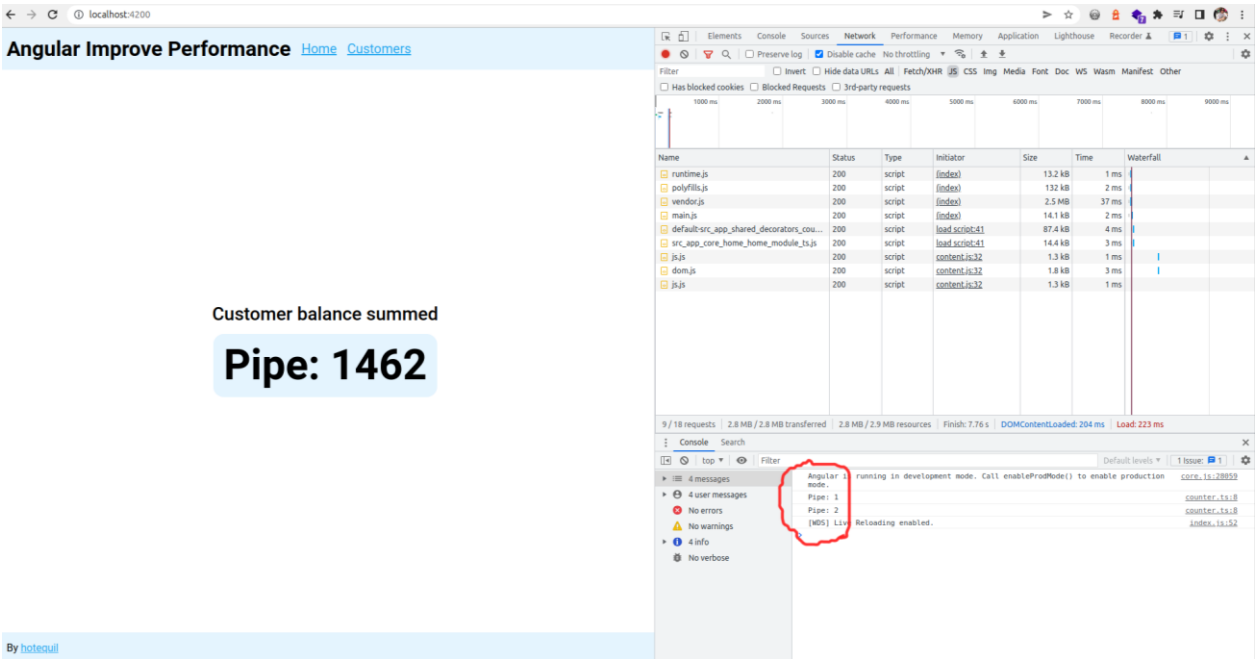
calculate-sum-of-customer-balances.pipe.ts:

Criação do nosso pipe.

```typescript
@Pipe({
    name: 'calculateSumOfCustomerBalances'
})
export class CalculateSumOfCustomerBalancesPipe implements PipeTransform{
    @counter("Pipe")
    transform(customers: Customer[]): number{
        return customers.reduce((accumulator, item) => accumulator + item.balance, 0);
    }
}
```

Browser:

Como podemos ver, obtivemos apenas duas chamadas, uma quando a lista de clientes estava vazia e outra quando foi preenchida, isso ocorre, porque o pipe é apenas chamado quando ocorre uma alteração na propriedade que ele está escutando e não na template do componente como as funções. Muito mais performático com o mesmo fim!



2) Se desinscrever dos subscribes:

No Angular, umas das poucas bibliotecas de terceiros que ele traz, é o RxJS, dentro dele temos os *Observables*, utilizamos ele para diversas coisas, como controlar eventos, fazer chamadas assíncronas, se comunicar com outras partes da aplicação e demais. Eles são maravilhosos, porque resolvem um problemão, porém, quando

se inscrevemos em *Observables*, devemos sempre se desinscrever, mas muitas vezes se esquecemos e isso se torna um problema silencioso que só percebemos com o vazamento de memória que gera lentidão para o usuário.

customers.component.ts:

Aqui temos uma lista de clientes e um evento de click que criei no objeto window do browser, ou seja, toda vez que clicarmos na tela estando dentro do componente de *CustomersComponent* (lembrando que esse componente está situado na rota "/customers") ele irá executar a função *onClickAtWindow*, gerar mais um incremento na variável *count* e criar um novo log por causa do nosso decorator *counter*.

```
const options: OptionsCounter = { router: null };

@Component({
    selector: 'app-customers',
    templateUrl: './customers.component.html',
    styleUrls: ['./customers.component.scss']
})
export class CustomersComponent implements OnInit{
    customers: Customer[] = [];
    count = 0;

    constructor(private customersService: CustomersService, router: Router){
        options.router = router;
    }

    async ngOnInit(): Promise<void>{
        this.customers = await this.customersService.get();
        fromEvent(window, "click").subscribe(() => this.onClickAtWindow())
    }

    @counter("You clicked on window", options)
    private onClickAtWindow(){
        this.count++;
    }
}
```

customers.component.html:

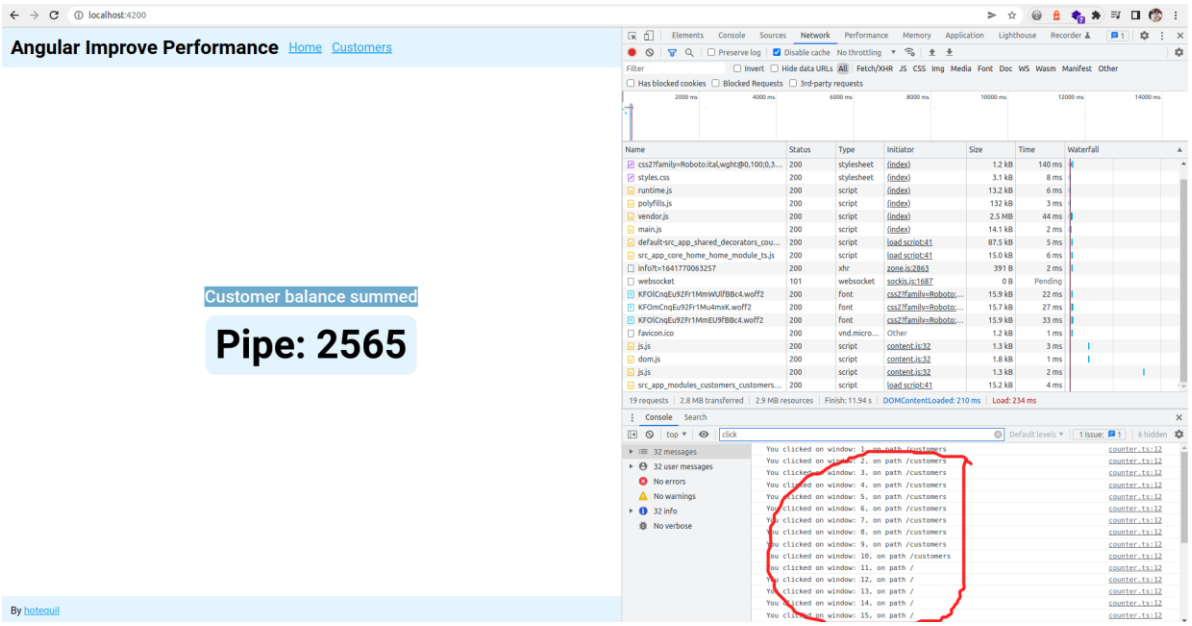
Simples HTML com o contador de clicks e uma lista de clientes.

```
<section class="customers">
    <h2 class="customers__title">
        Name and balance customers (click on window to launch an event: {{ count }} clicks)
    </h2>
    <ul class="customers__list">
        <li class="customers__item"
            *ngFor="let customer of customers">
            {{ customer.name }} ({{ customer.balance }})
        </li>
    </ul>
</section>
```

Browser:

Percebemos pelo log do inspecionador de elementos do browser que estávamos clicando na tela enquanto estávamos na rota "/customers" e quando fui para a "/" que é a rota do componente *HomeComponent*, continuei clicando na tela e ele

seguiu emitindo o evento que declarei apenas no *CustomersComponent*, então, esse tipo de vazamento de memória é muito perigoso, pois, queremos que ele seja emitido apenas no componente *CustomersComponent*, vale lembrar que essa foi uma pequena amostra, imagine isso com um método que faça requisições para uma API.



Para isso precisamos encerrar esse evento quando o componente for destruído, existem muitas formas de se desinscrever de um *Observable*, existem operadores do RxJS como take, takeUntil e outros que fazem isso, mas vamos se desinscrever da forma mais simples.

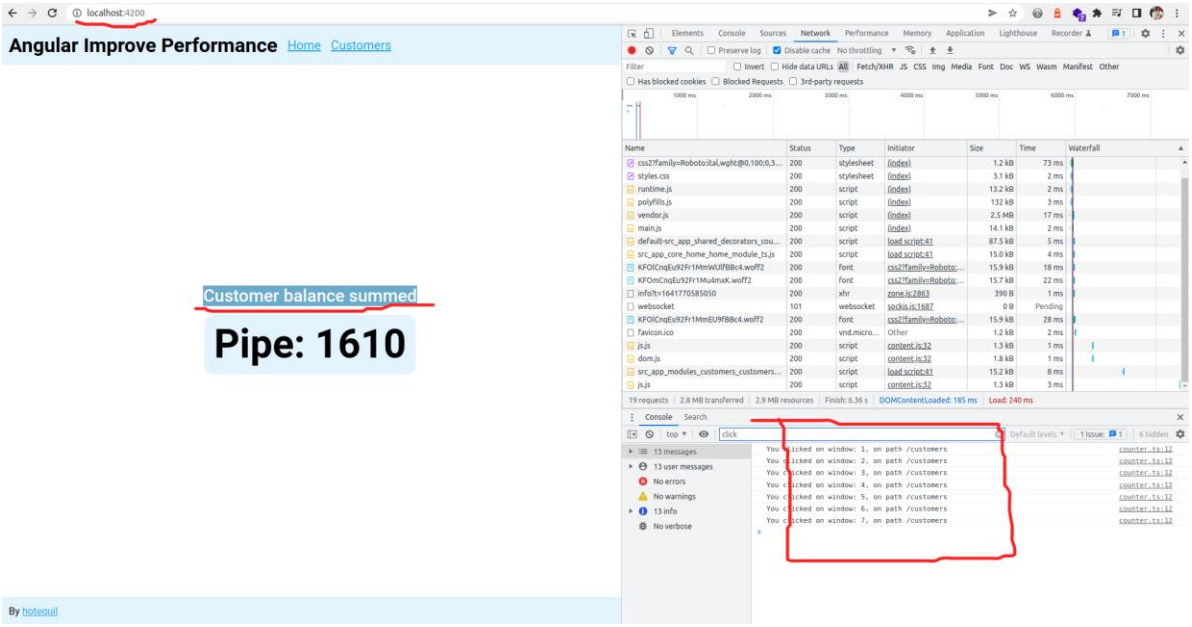
customers.component.ts:

Criamos uma nova propriedade no componente chamada *clickWindow$*, será atribuída nela a inscrição do *Observable*, com isso implementamos a interface do *OnDestroy* que nos obriga a criar o método *ngOnDestroy* que contém a desinscrição da nossa variável.

```typescript
const options: OptionsCounter = { router: null };

@Component({
    selector: 'app-customers',
    templateUrl: './customers.component.html',
    styleUrls: ['./customers.component.scss']
})
export class CustomersComponent implements OnInit, OnDestroy{
    customers: Customer[] = [];
    count = 0;

    private clickWindow$!: Subscription;

    constructor(private customersService: CustomersService, router: Router){
        options.router = router;
    }

    async ngOnInit(): Promise<void>{
        this.customers = await this.customersService.get();
        this.clickWindow$ = fromEvent(window, "click").subscribe(() => this.onClickAtWindow())
    }

    @counter("You clicked on window", options)
    private onClickAtWindow(){
        this.count++;
    }

    ngOnDestroy(): void{
        this.clickWindow$.unsubscribe();
    }
}
```

Browser:

Após esse ajuste no nosso código, percebemos que o click na tela só é emitido na rota “/customers” e não mais na “/”, pois, nosso evento é finalizado quando o *CustomersComponent* é destruído. Até selecionei o texto na tela para mostrar que o click não está mais emitindo logs na rota “/” (*HomeComponent*).



3) Use lazy loading:

Você já entrou em um site e ele demorou uma eternidade para carregar? Certamente sim. Isso pode ser causado pela sua banda larga ou dispositivo, mas geralmente é causado pelo site carregar recursos demais ao iniciar, como scripts, folhas de estilos, fontes e outros. O lazy loading é uma solução extremamente eficaz para você carregar apenas o módulo que seu usuário está usando, assim diminuindo o tamanho da sua aplicação ao iniciar. Também o lazy loading é muito bom quando você tem aquele sistema grande com diversos tipos de usuários que podem acessar só determinadas telas, assim o lazy loading permite que você carregue apenas o que o seu usuário precise, porque não faz sentido um usuário carregar um módulo que não tem permissão para usar. Vamos para a implementação!

home-routing.module.ts:

Criação do módulo de rotas da home.

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

@NgModule({
    imports: [RouterModule.forChild(routes)],
    exports: [RouterModule]
})
export class HomeRoutingModule{}
```

home.module.ts:

Importação do módulo de rotas no módulo principal da home.

```
@NgModule({
    declarations: [
        HomeComponent
    ],
    imports: [
        CommonModule,
        HomeRoutingModule,
        CalculateSumOfCustomerBalancesModule
    ]
})
export class HomeModule{}
```

customers-routing.module.ts:

Criação do módulo de rotas de customers.

```ts
const routes: Routes = [
    {
        path: '',
        component: CustomersComponent
    }
];

@NgModule({
    imports: [RouterModule.forChild(routes)],
    exports: [RouterModule]
})
export class CustomersRoutingModule{}
```

customers.module.ts:

Importação do módulo de rotas no módulo principal de customers.

```ts
@NgModule({
    declarations: [
        CustomersComponent
    ],
    imports: [
        CommonModule,
        CustomersRoutingModule
    ]
})
export class CustomersModule{}
```

app-routing.module.ts:

Carregamento dos módulos ao acessar nossas rotas definidas.

```ts
const routes: Routes = [
    {
        path: '',
        loadChildren: () => import("./core/home/home.module").then(module => module.HomeModule)
    },
    {
        path: 'customers',
        loadChildren: () => import("./modules/customers/customers.module").then(module => module.CustomersModule)
    }
];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
})
export class AppRoutingModule { }
```

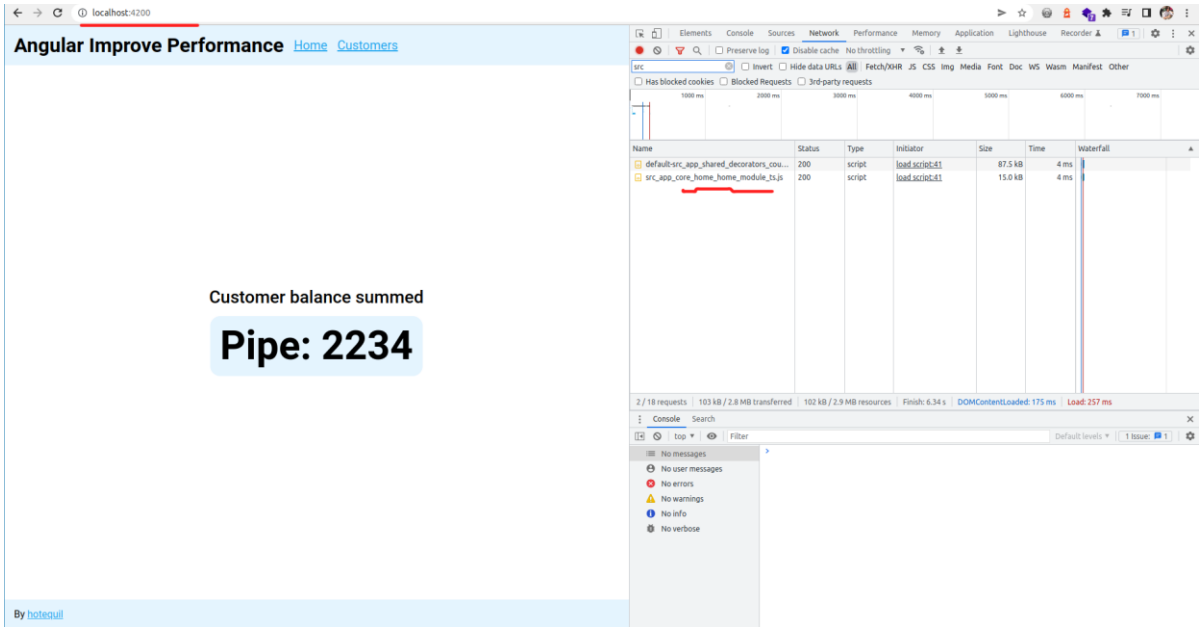
app.module.ts:

Importação do módulo de rotas principal no *AppModule*.

```ts
@NgModule({
    declarations: [
        AppComponent
    ],
    imports: [
        BrowserModule,
        AppRoutingModule
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

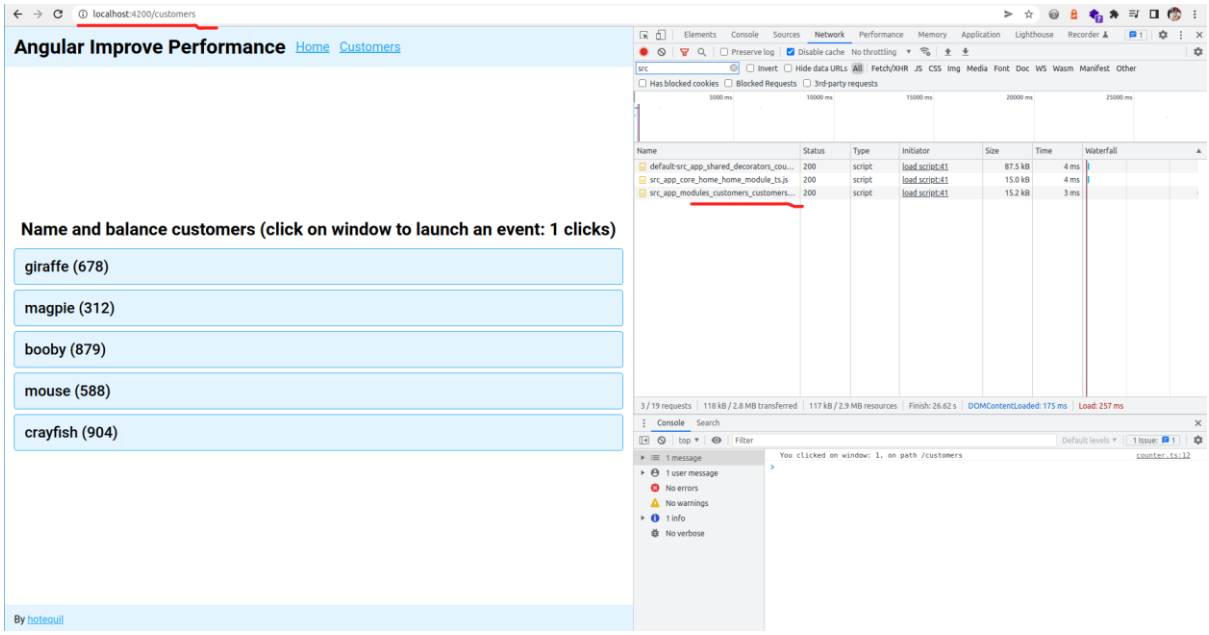
Home browser:

Conseguimos observar que ao acessar a rota do módulo da home, ele baixa apenas o script da home.



Customers browser:

Para finalizar, ao acessar a rota de “/customers”, o Angular se encarrega de baixar o script do customer.



Conclusão:

Muitas vezes escutamos das pessoas que o que importa é entregar a feature para o usuário, e não está errado, mas entregar algo funcional e lento não é muito bacana para quem vai usar no final, mais do que funcional, devemos ter a velocidade como

um princípio. Isso é muito relevante principalmente para usuários móveis que geralmente tem uma banda larga menor e para pessoas com um hardware não muito sofisticado. Espero ter ensinado, questionado ou deixado uma pulga atrás da orelha para você estudar e ajustar o seu código atual! Para aqueles que já sabiam disso, show de bola, fico feliz, com esse artigo, dou mais ênfase a esse conhecimento que já lhe pertence. Qualquer dúvida não excite em me procurar nas mídias sociais, abraços e vamos melhorar o mundo.