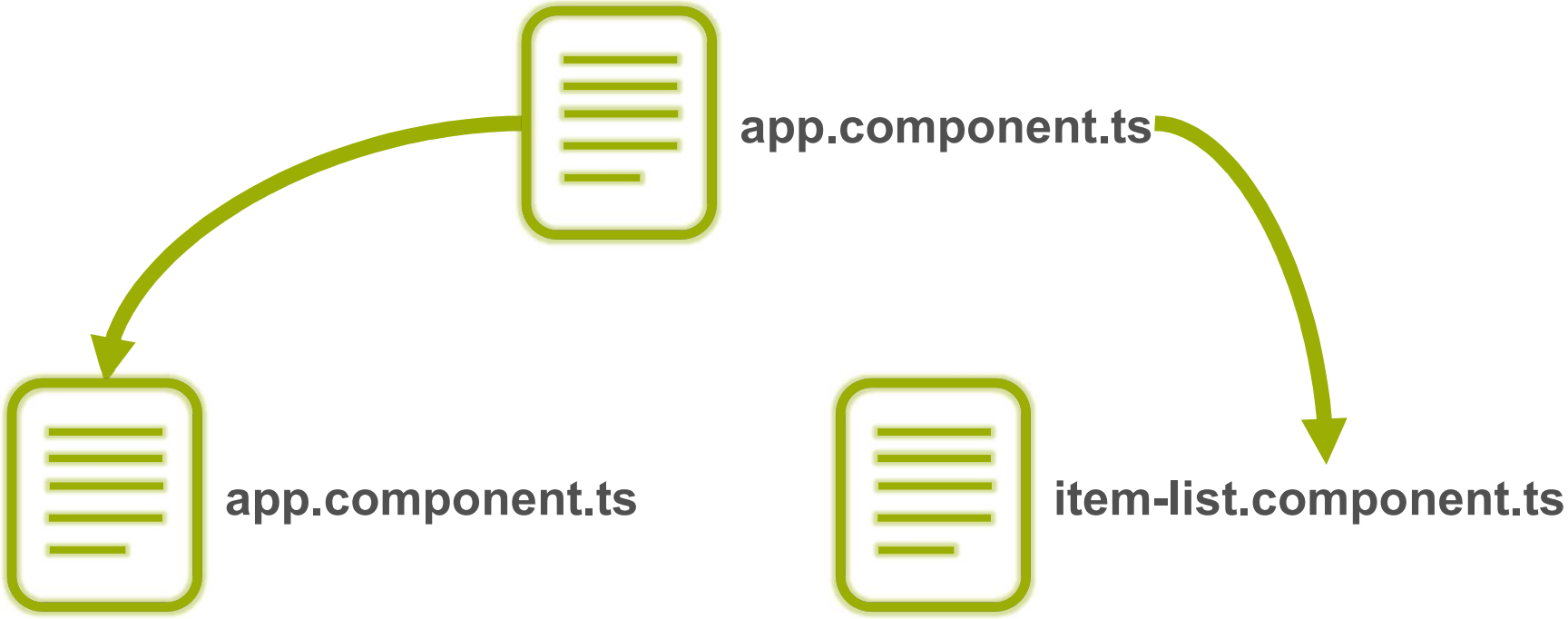
— 04

Splitting to Two Components

We've been developing Angular in one single file: app.component.ts. This isn't going to scale, so let's split things up into pieces and get organized.

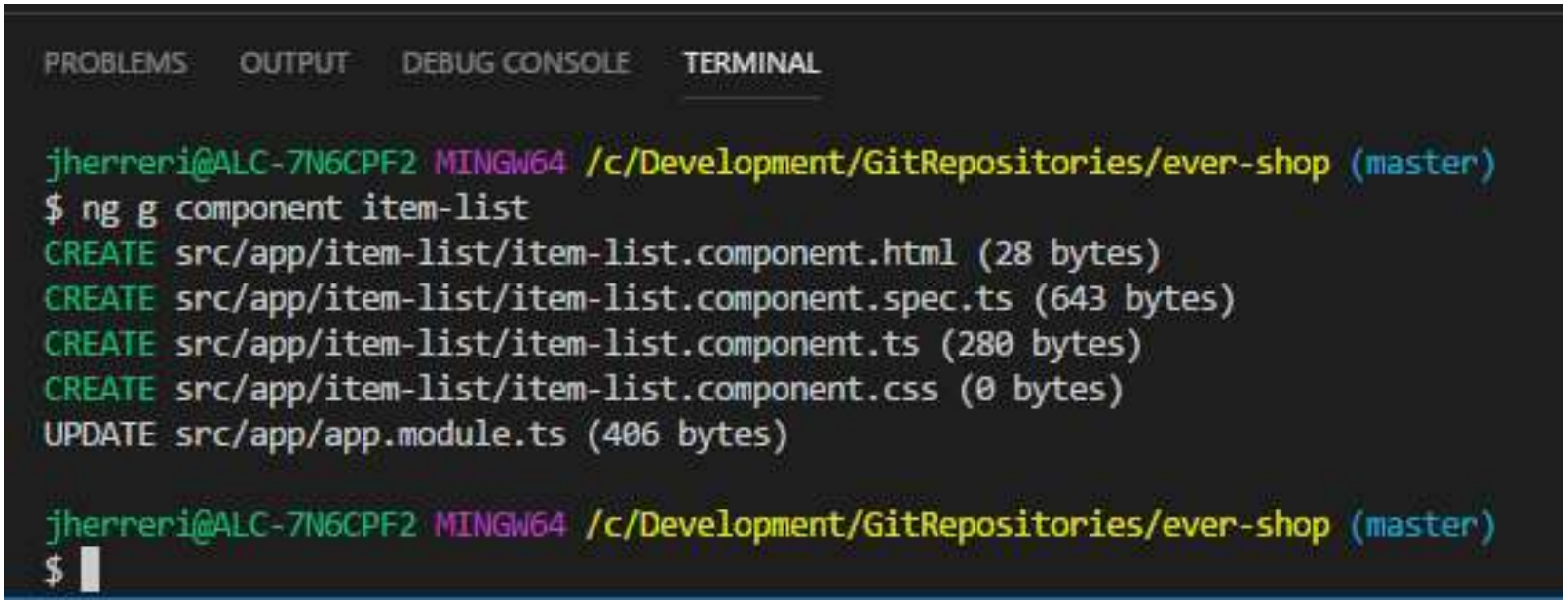


Create Our New Component

We need to create a item-list.component.ts

Let's use Angular-CLI again!

```
ng g component item-list
```



Splitting Out Our Item List Component

Moving code from app.component.ts to item-list.component.ts and code from app.component.html to item-list.component.html

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Ever Shop';
  myItems = [{
    'id': 1,
    'name': 'Item name',
    'description': 'These item is the best one',
    'stock': 5,
    'price': 14.99
  },
  ...
  totalItems() {
    let sum = 0;
    for (let myItem of this.myItems) {
      sum += myItem.stock;
    }
    return sum;
  }
```

New Selector

item-list.component.ts

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

Splitting Out Our Item List Component

Components are meant to be reusable, so we could use them in different parts of our application.

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Ever Shop';
}
```

item-list.component.ts

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

@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html',
  styleUrls: ['./item-list.component.css']
})
export class ItemListComponent implements OnInit {
  myItems = [{
    'id': 1,
    'name': 'Item name',
    'description': 'This item is the best one',
    'stock': 5,
    'price': 14.99
  }....
  constructor() { }

  ngOnInit() {
  }

  totalItems() {
    return this.myItems.reduce( (prev, current) => prev + current.stock, 0);
  }
    'id': 3,
    'name': 'A cheap Item',
    'description': 'The cheapest item!',
    'stock': 0,
    'price': 3.99
  }];
```

app.component.html

```html
<div style="text-align:left">
  <h1>
    Welcome to {{title}}!
  </h1>

  <app-item-list></app-item-list>
</div>
```

item-list.component.html

```html
<p>There are {{totalItems()}} total items in stock.</p>
<ul>
  <li *ngFor="let item of myItems">
    <h2>{{item.name | uppercase}}</h2>
    <p>{{item.description}}</p>
    <p>{{item.price | currency:'EUR':true}}</p>
    <p *ngIf="item.stock > 0">{{item.stock}} in Stock</p>
    <p *ngIf="item.stock === 0">Out of Stock</p>
  </li>
</ul>
```

Angular crea los componentes

Los renderiza

Crea y renderiza a sus hijos

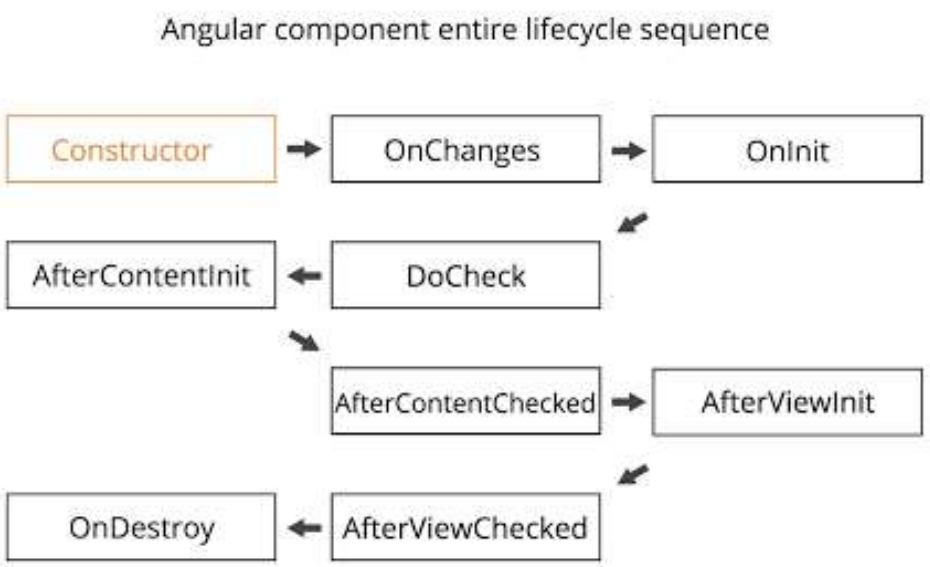
Actualiza y re-renderiza los componentes cuando detecta cambios

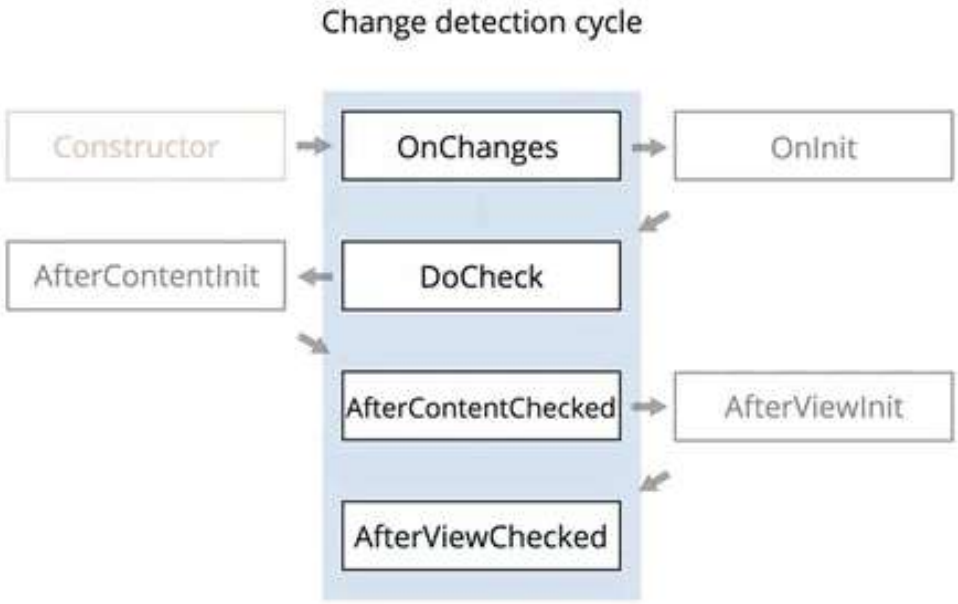
Finalmente los destruye antes de eliminarlos del DOM

Todo gestionado por Angular

Angular te da acceso a los momentos clave de su vida a través de unos callbacks, los denominados **lifecycle hooks**

Component Lifecycle Hooks





La creación no necesita ningún lifecycle hook. Directamente tenemos el constructor

Sólo se llamarán aquellos que tengamos implementados

ngOnInit() - Inicializa el componente una vez ha recibido las propiedades de entrada

ngOnChanges() - Inicio y cada vez que Angular detecta un cambio en los inputs del componente. Recibe como parámetro un objeto SimpleChanges, con los valores actuales y anteriores (si había) de los inputs

ngDoCheck() - Sirve para detectar y actuar sobre cambios que Angular no va a detectar por si mismo. Se llama después de ngOnChanges().

ngAfterContentInit() - Se ejecuta una sola vez, justo después de que Angular proyecte contenido externo en la vista del componente (con ng-content).

ngAfterContentChecked() - Se ejecuta después de que Angular compruebe el contenido proyectado en el componente. Se ejecuta también durante los ciclos de detección de cambios, después de ngDoCheck().

ngAfterViewInit() - Se llama una única vez, tras inicializar las vistas y sub-vistas del componente.

ngAfterViewChecked() - Se llama después de comprobar los cambios de las vistas y sub-vistas del componente. Se ejecuta también durante el ciclo de detección de cambios, después de ngAfterContentChecked().

ngOnDestroy() - Se llama solo una vez, justo antes de que Angular destruya el componente, y sirve para prevenir memory leaks, eliminando por ejemplo suscripciones a Observables e event handlers.

¿Cómo se usan?

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

@Component({
  selector: 'app-foo'
})
export class FooComponent implements OnInit {
  constructor() { }

  ngOnInit() {
    // do something here
  }
}
```

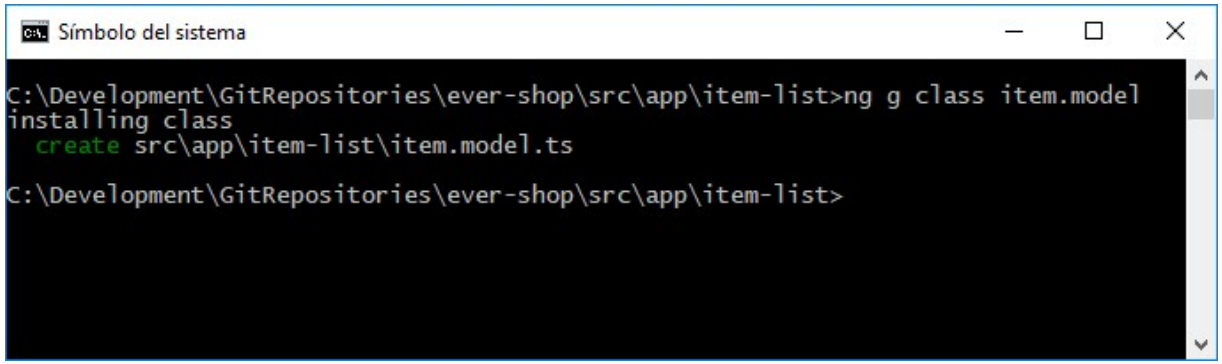
Sólo hay que implementar el método del hook que quieras utilizar

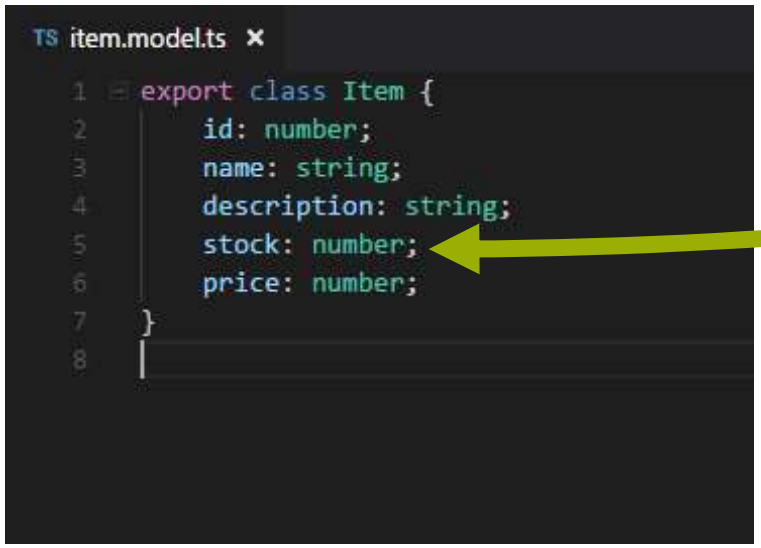
— 05

Mocks & Models

TypeScript gives us the ability to be more object oriented with our data, so let's create a model.

```
ng g class item.model
```





Thanks to TypeScript we're declaring what type each of our properties are.

Import the Model & Define Type

item-list.component.ts

```ts
import { Component, OnInit } from '@angular/core';
import { Item } from './item.model';

@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html',
  styleUrls: ['./item-list.component.css']
})
export class ItemListComponent implements OnInit {
  myItems: Item[] = [{
    'id': 1,
    'name': 'Item name',
    'description': 'This item is the best one',
    'stock': 5,
    'price': 14.99
  },
```

Import the Item Model

Tells TypeScript to treat this like an array of Items

item.model.ts

```ts
export class Item {
    id: number;
    name: string;
    description: string;
    stock: number;
    price: number;
}
```

item-list.component.ts

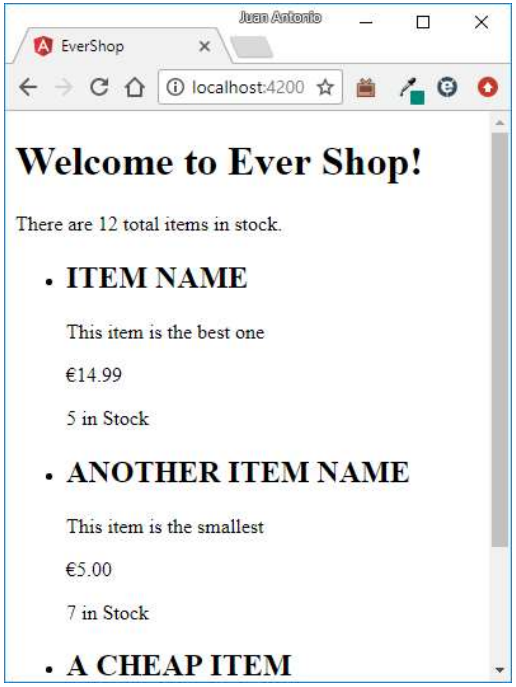
```ts
import { Component, OnInit } from '@angular/core';
import { Item } from './item.model';

@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html',
  styleUrls: ['./item-list.component.css']
})
export class ItemListComponent implements OnInit {
  myItems: Item[] = [{
    'id': 1,
    'name': 'Item name',
    'description': 'This item is the best one',
    'stock': 5,
    'price': 14.99
  },
```

item.model.ts

```ts
export class Item {
    id: number;
    name: string;
    description: string;
    stock: number;
    price: number;
}
```

item-list.component.html

```html
<p>There are {{totalItems()}} total items in stock.</p>
<ul>
  <li *ngFor="let item of myItems">
    <h2>{{item.name | uppercase}}</h2>
    <p>{{item.description}}</p>
    <p>{{item.price | currency:'EUR':true}}</p>
    <p *ngIf="item.stock > 0">{{item.stock}} in Stock</p>
    <p *ngIf="item.stock === 0">Out of Stock</p>
  </li>
</ul>
```



Cleaning Up Our Mock Data

Eventually we want to call out to a web service (API) to get the item list, so it's a good practice to move our mock (fake) data out into its own file.

item-list.component.ts

```ts
import { Component, OnInit } from '@angular/core';
import { Item } from './item.model';
import { ITEMS } from './mocks';

@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html',
  styleUrls: ['./item-list.component.css']
})
export class ItemListComponent implements OnInit {
  myItems: Item[];

  constructor() { }

  ngOnInit() {
    this.myItems = ITEMS;
  }

  totalItems() {
    return this.myItems.reduce( (prev, current) => prev + current.stock, 0);
  }
}
```

ngOnInit is invoked after the component is constructed and is the best place to initialize property values.

We could have initialized in the constructor, but that'd be harder to test.

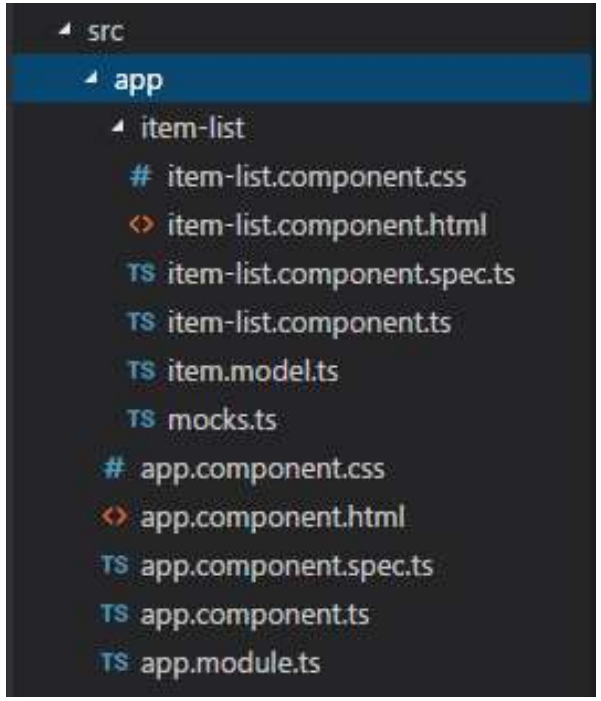
mocks.ts

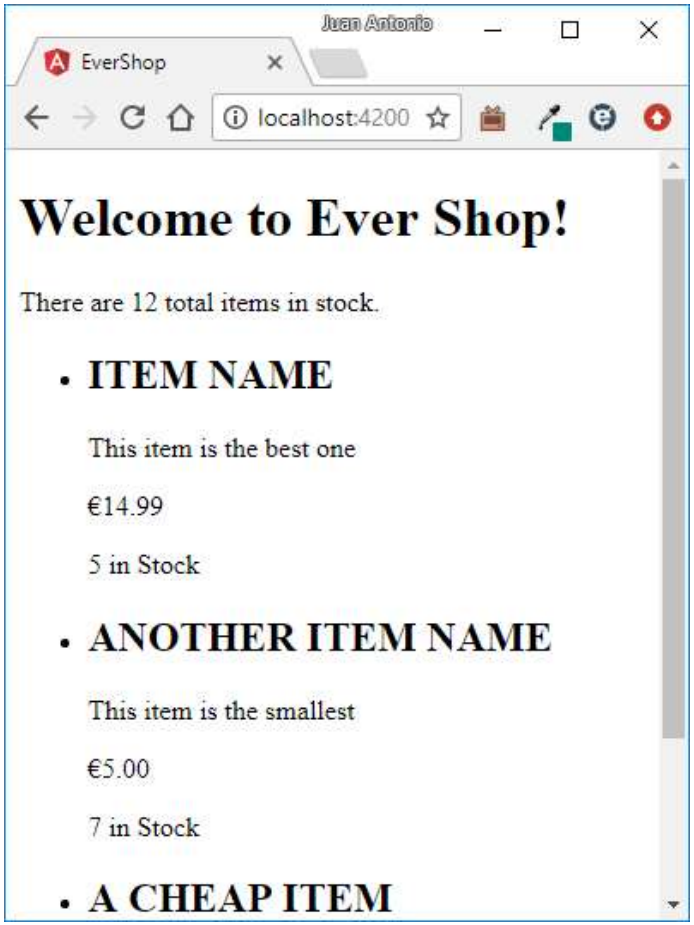
```ts
import { Item } from './item.model';

export const ITEMS: Item[] = [{
    'id': 1,
    'name': 'Item name',
    'description': 'This item is the best one',
    'stock': 5,
    'price': 14.99
  },
  {
    'id': 2,
    'name': 'Another Item name',
    'description': 'This item is the smallest',
    'stock': 7,
    'price': 5
  },
  {
    'id': 3,
    'name': 'A cheap Item',
    'description': 'The cheapest item!',
    'stock': 0,
    'price': 3.99
  }];
```

Notice we use const instead of let — that makes sure items can't be reassigned.

It still works after refactoring

We didn't add any more functionality, but our code became a lot easier to maintain and scale.





It's a good practice to keep your mock data separate from your model and your components, in its own file.