UNIT 2

Angular

Part 2
Custom pipes. More on components. Services.

Client-side Web Development
2nd course – DAW
IES San Vicente 2020/2021
Author: Arturo Bernal Mayordomo

Index

Building custom pipes

As we saw earlier. Pipes are used to transform data before displaying it. There are some predefined pipes in Angular for string transformation, date or currency display, as you can see here.

We can also create our custom pipes. For example, let's create a pipe that transforms the list of products (array of Iproduct), filtering those products by description. The fastest way to create a pipe is using the corresponding Angular CLI command:

```
ng g pipe pipes/product-filter
```

Remember that we should use snake-case for the names created with Angular CLI (ng). The tool will adapt that name depending of the context (in this example the created pipe's name will be **productFilter** and the class name **ProductFilterPipe**).

This is what the created Pipe class inside **product-filter.pipe.ts** looks like:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'productFilter'
})
export class ProductFilterPipe implements PipeTransform {
  transform(value: any, args?: any): any {
    return null;
  }
}
```

As you can see, it's similar to a component class, but it uses the **@Pipe** decorator so Angular can know how to use this class. The name '**productFilter**' will be used when filtering the list.

This class implements the **PipeTransform** Interface which obligates us to implement the **transform** method. This is the method that will be used to filter (transform) the array of products. As we are using TypeScript, we can type parameters (the first one is what we'll have to transform, in this case the products list) and the rest of arguments received (in this case a string with the description to filter).

```
import { Pipe, PipeTransform } from '@angular/core';
import { IProduct } from '../interfaces/iproduct';// Don't forget the import

@Pipe({
  name: 'productFilter'
})
export class ProductFilterPipe implements PipeTransform {
  transform(products: IProduct[], filterBy: string): IProduct[] {
    return null;
  }
}
```

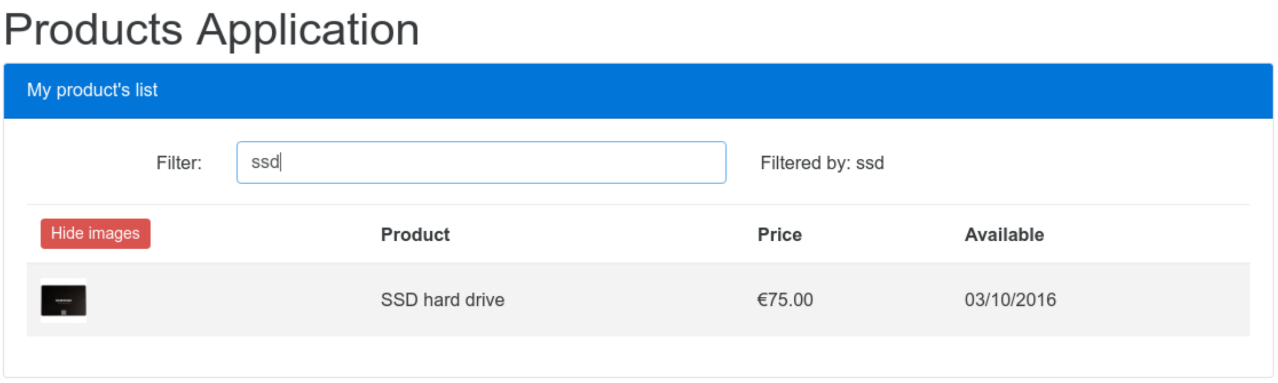
Now, we are going to implement the transform method. Basically, we'll transform the parameter **filterBy** to lowercase and compare to product's description also in lowercase (we want this filter to be case insensitive). Then we just filter the array and return the products which name contains the string received. If there’s no filter string, we’ll just return the original array (value).

```
  transform(products: IProduct[], filterBy: string): IProduct[] {
    const filter = filterBy ? filterBy.toLocaleLowerCase() : null;
    return filter ?
         products.filter(prod => prod.desc.toLocaleLowerCase().includes(filter)) :
         products;
  }
```

The last step is: go to the **product-list.component.html** template and apply this pipe to the list of products (*ngFor). We need to write the name of the pipe → **productFilter**, and pass a parameter which is the search string (referenced by the property **filterSearch** in the component).

```
<tr *ngFor="let product of products | productFilter:filterSearch">
    ...
</tr>
```

And now it should work!


Products Application

My product's list

Filter: ssd

Filtered by: ssd

Hide images	Product	Price	Available
	SSD hard drive	€75.00	03/10/2016

Remember that the **ProductFilterPipe** class must be added to our **AppModule** class (**src/app/app.module.ts**). Angular CLI has already done this for us:

```
@NgModule({
  declarations: [
    AppComponent,
    ProductListComponent,
    ProductItemComponent,
    ProductFilterPipe
  ],
  ...
})
```

More on components

In this section we'll learn more about Angular components: How can we apply custom styles just to our component, how component lifecycle works and which built-in methods we can use, how to take advantage of nesting components, and communicate those nested components with each other (@input, @output).

Component styles encapsulation

You may have noticed that components created with Angular CLI, already have a related CSS file, which is referenced in the @Component decorator. The styles added to this file will **only apply** to this component. They won't affect the rest of the application.

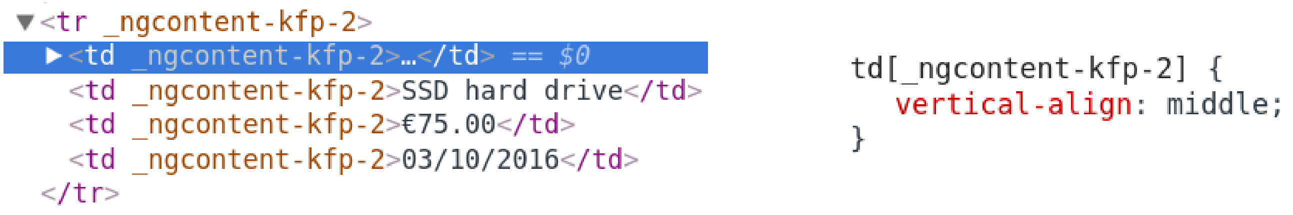
If we just wanted to add some CSS properties instead of using a file, we could use the **style** attribute in the @Component decorator:

```
@Component({
  selector: 'product-list',
  templateUrl: './product-list.component.html',
  styles: [`
    td {
        vertical-align: middle;
    }`,`

    td:first-child img {
        height: 40px;
    }`
  ]
})
```

However, it's usually more recommended to include a CSS file (or more) using the **styleUrls** attribute (array) and define the component style in those files:

```
@Component({
  selector: 'product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
```

In order to isolate the style only to this component, Angular will generate a random HTML attribute and assign that to this component's elements, and also to the CSS properties defined (so they only affect this HTML).



```
▼<tr _ngcontent-kfp-2>
  ►<td _ngcontent-kfp-2>…</td> == $0
   <td _ngcontent-kfp-2>SSD hard drive</td>
   <td _ngcontent-kfp-2>€75.00</td>
   <td _ngcontent-kfp-2>03/10/2016</td>
 </tr>
```

```
td[_ngcontent-kfp-2] {
   vertical-align: middle;
}
```

Global styles

If you want to define global CSS styles in your application, you have 2 choices:

- Include them in your **src/styles.css** file:

```
/* You can add global styles to this file, and also import other style files */
@import "../node_modules/bootstrap/dist/css/bootstrap.css"
```

- Include them in your **angular.json** file (styles property):

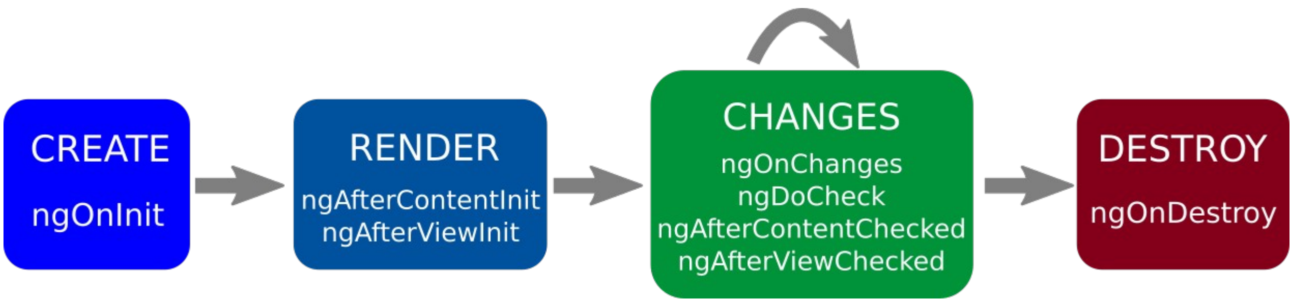
```
"styles": [
    "styles.css",
    "node_modules/bootstrap/dist/css/bootstrap.css"
],
```

Angular CLI is capable of working with CSS preprocessors or compilers like **SASS**. Use the **--style** option when creating a new project (possible values: **css, scss, less, sass, styl**). If you want to change the current project's configuration to **scss**, for example, just run:

```
ng add schematics-scss-migrate

ng g schematics-scss-migrate:scss-migrate
```

Components lifecycle

There are several stages a component goes through during its lifetime (created → destroyed). Angular allows us to establish some behavior on each of those stages using methods in the component that are executed when some stage is reached.



- **ngOnInit** → Executes once when the component is ready. It's usually used to get data from the server for example.

- **ngAfterContentInit** and **ngAfterViewInit** → Executed once (in that order) just after ngOnInit. They execute when the content is ready to display in the view (ngAfterContentInit) and when the component and children component's views are first displayed (ngAfterViewInit).

- **ngOnChanges** → Executed every time a property bound to the view changes.

- **ngDoCheck** → This method is used to detect changes not automatically detected by Angular (for example, a property of an object inside a list changes. Angular would only detect changes to the list like insertion, delete or object substitution). However this method (if set) runs very often and it's recommended to void using it for performance purposes (better use nested components!).

- **ngAfterContentChecked** and **ngAfterViewChecked** → Run after every ngDoCheck in that order. They're not usually implemented.

- **ngOnDestroy** → Called once when the component is destroyed (not used in the view). Use this method if you need to clean some data.

More information about components lifecycle here.

Let's see a very simple example using ngOnInit:

```
export class ProductListComponent implements OnInit {
  ...
  ngOnInit() {
    console.log("ProductListComponent has been initialized!");
  }

}
```

See that we're implementing the interface **OnInit**, which obligates us to declare the method **ngOnInit** inside the component. Every lifecycle hook has its own interface which is recommended to implement if we need to respond to any of these events. The interface is always called the same as the event method without the '**ng**' prefix, for example: **ngOnDestroy** → **OnDestroy**. Don't forget to import from '**@angular/core**'.

You should see a message like this on the console when the page is loaded:

```
ProductListComponent has been initialized!     product-list.component.ts:40
```

Nested components

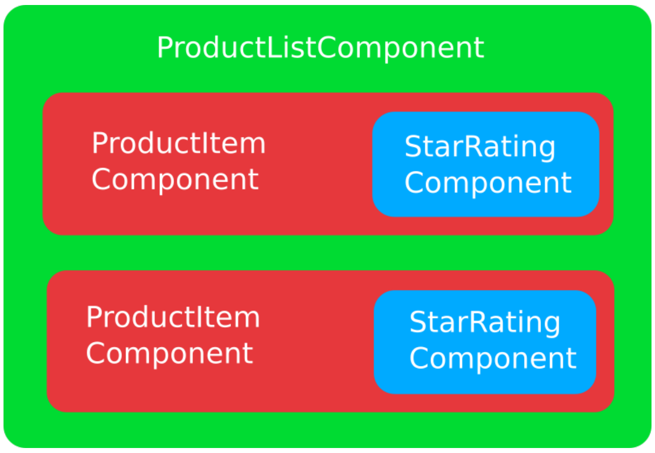
A nested or child component, is a component whose template represents a fragment of the actual view. For example, we can make every product in the products' list a child component of ProductListCompoment.

First of all, we'll create a component that represents a product. Its selector will be **product-item**.

```
ng g component product-item
```

We're also going to create another component to rate the product (using stars). This component will be **star-rating**.

```
ng g component star-rating
```



Nesting ProductItemComponent

Now, we are going to implement **ProductItemComponent** with all the logic that represents a single product in our list. We'll use fixed information until we know how to get product info from the parent component.

```
import { IProduct } from '../interfaces/i-product';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'product-item',
  templateUrl: './product-item.component.html',
  styleUrls: ['./product-item.component.css']
})
export class ProductItemComponent implements OnInit {
  product: IProduct = {
    id: 1,
    description: 'SSD 250GB hard drive',
    available: new Date('2016-10-03'),
    price: 75,
    imageUrl: 'assets/ssd.jpg',
    rating: 5
  };

  showImage = true;

  constructor() {}

  ngOnInit() {}
}
```

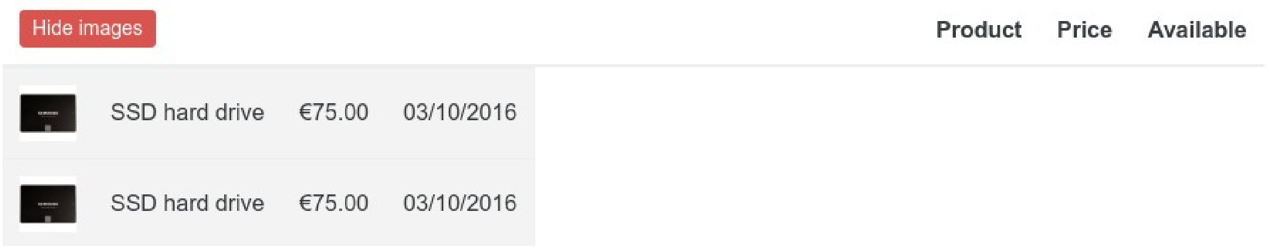
As you can see, this component has all the information needed to display a single product. All we need to do is complete its template (**product-item.component.html**).

```
<tr>
  <td>
    <img [src]="product.imageUrl" *ngIf="showImage" alt=""
      [title]="product.description | uppercase">
  </td>
  <td>{{ product.description }}</td>
  <td>{{ product.price | currency:'EUR':'symbol' }}</td>
  <td>{{ product.available | date:'dd/MM/y' }}</td>
</tr>
```

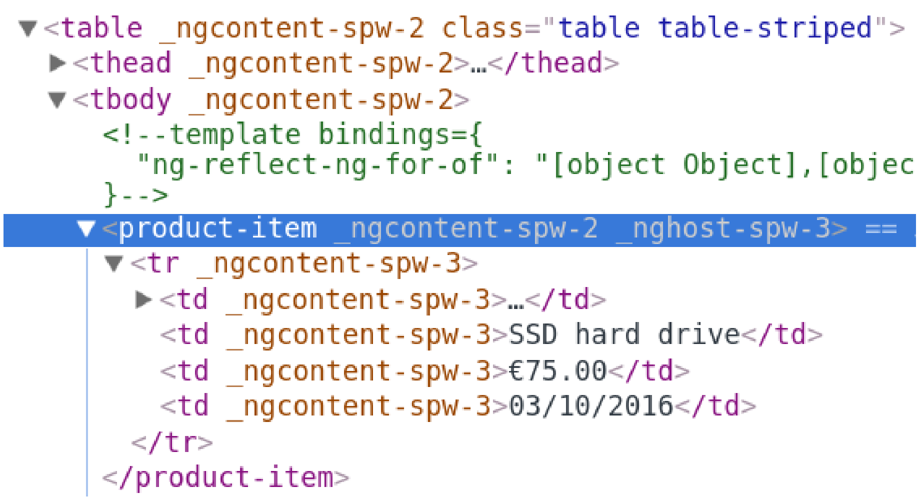
It's just the piece of HTML corresponding to a product's row in the table (without the ***ngFor** directive). The last part of this process is using this selector (**product-item**) in the **ProductListComponent**'s template (**product-list.component.html**):

```
<tbody>
    <product-item *ngFor="let product of products | productFilter:filterSearch">
    </product-item>
</tbody>
```

Move also the corresponding **CSS styles** to the new component.



However, as you can see, there's a problem with the table. It seems that this table's rows aren't integrating very well with it. This is because the **<product-item>** element is in the DOM and the browser doesn't detect rows <tr> as being inside table's <tbody>:



```
▼<table _ngcontent-spw-2 class="table table-striped">
  ▶<thead _ngcontent-spw-2>…</thead>
  ▼<tbody _ngcontent-spw-2>
      <!--template bindings={
        "ng-reflect-ng-for-of": "[object Object],[objec
      }-->
    ▼<product-item _ngcontent-spw-2 _nghost-spw-3> ==
      ▼<tr _ngcontent-spw-3>
        ▶<td _ngcontent-spw-3>…</td>
         <td _ngcontent-spw-3>SSD hard drive</td>
         <td _ngcontent-spw-3>€75.00</td>
         <td _ngcontent-spw-3>03/10/2016</td>
       </tr>
     </product-item>
```

Let’s fix this changing the HTML structure (using Boostrap) and CSS a little bit.

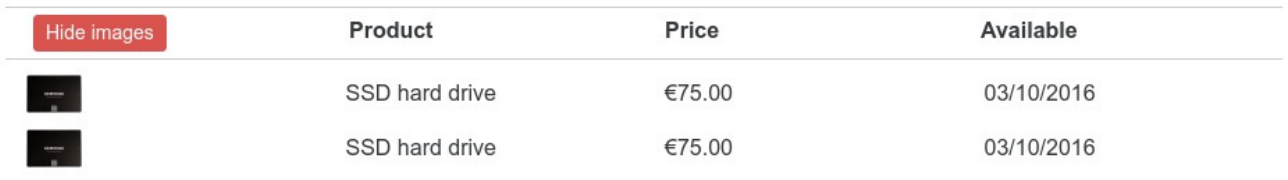
```
#product-list.component.html

<div *ngIf="products && products.length">
  <div class="row headers no-gutters">
    <div class="col-2">
        <button class="btn btn-sm"
            [ngClass]="{'btn-danger': showImages, 'btn-primary': !showImages}"
            (click)="toggleImage()">
            {{showImage?'Hide':'Show'}} images
        </button></div>
    <div class="col-4">{{headers.description}}</div>
    <div class="col">{{headers.price}}</div>
    <div class="col">{{headers.available}}</div>
  </div>
  <product-item class="row no-gutters"
      *ngFor="let product of products | productFilter:filterSearch">
  </product-item>
</div>
```

```
#product-list.component.css

.headers {
    font-weight: bold;
    border-bottom: 1px solid #CCC;
    border-top: 1px solid #CCC;
    padding: 5px;
    margin-bottom: 5px;
}

product-item:nth-child(odd) {
    background: #EEE;
}
```

```
#product-item.component.html

<div class="col-2 pl-2">
  <img [src]="product.imageUrl" *ngIf="showImage" alt=""
     [title]="product.description | uppercase">
</div>
<div class="col-4">{{ product.description }}</div>
<div class="col">{{ product.price | currency:'EUR':'symbol' }}</div>
<div class="col">{{ product.available | date:'dd/MM/y' }}</div>
```

```
#product-item.component.css

div:first-child img {
    height: 40px;
}

div:not(:first-child) {
    display: flex;
    justify-content: center;
    flex-direction: column;
}
```

Much better now!



Hide images	Product	Price	Available
	SSD hard drive	€75.00	03/10/2016
	SSD hard drive	€75.00	03/10/2016

Nesting StartRateComponent

The next step is to create a rating system (from 1 to 5) based on star icons. In order to have those icons, we could use corresponding Unicode characters, or install an icon set font like Font Awesome:

```
npm install @fortawesome/fontawesome-free
```

And import it in the **angular.json** file (styles array):

```
"./node_modules/@fortawesome/fontawesome-free/css/all.min.css"
```

Let's see what the component should look like, we only need a rating number (integer) right now:

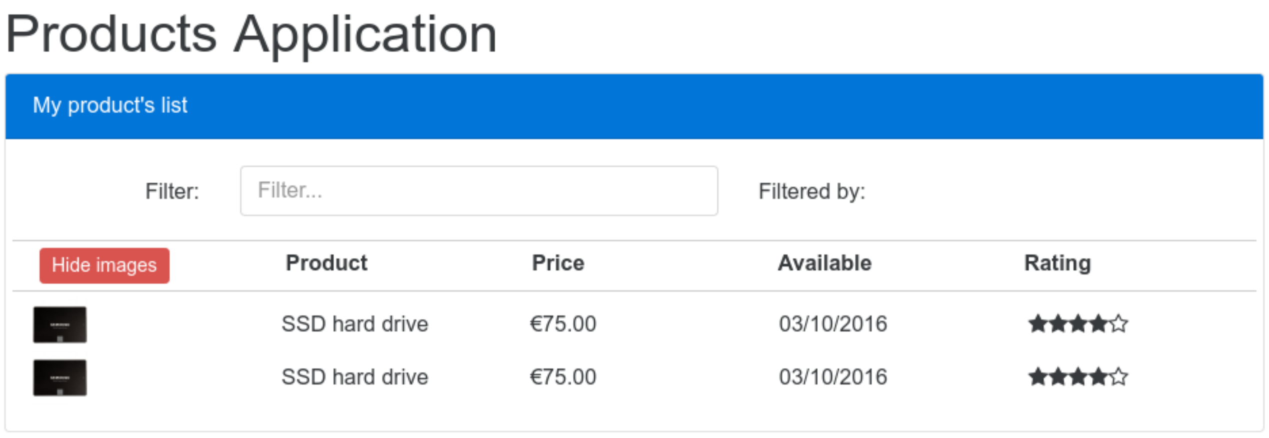
```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'star-rating',
  templateUrl: './star-rating.component.html',
  styleUrls: ['./star-rating.component.css']
})
export class StarRatingComponent implements OnInit {
  rating: number = 4;
  ...
}
```

On the template, we'll produce 5 stars (we can do it with *ngFor using a fixed array). This elements will have a different class depending on the rating condition to display a full star (<=rating) or an empty star (> rating).

```
#star-rating.component.html

<div class="star-container">
  <span *ngFor="let star of [1,2,3,4,5]" class="fa-star"
    [ngClass]="{'fas': star <= rating, 'far': star > rating}"></span>
</div>

#product-item.component.html
```

```
...
<div class="col"><star-rating></star-rating></div>
```

And this is the result (keep in mind that rating is now fixed to 4 (it's not using the real product's rating, at least until we learn how to pass information between components).



Products Application

My product's list

Filter: Filter... Filtered by:

Hide images	Product	Price	Available	Rating
	SSD hard drive	€75.00	03/10/2016	★★★★☆
	SSD hard drive	€75.00	03/10/2016	★★★★☆

Communication with nested components

In the previous section, we learned how to nest components. However, we don't have the necessary functionality yet (display correct product, hide/show image, show correct star rating) as we still don't know how components communicate.

Passing data to a child component → @Input

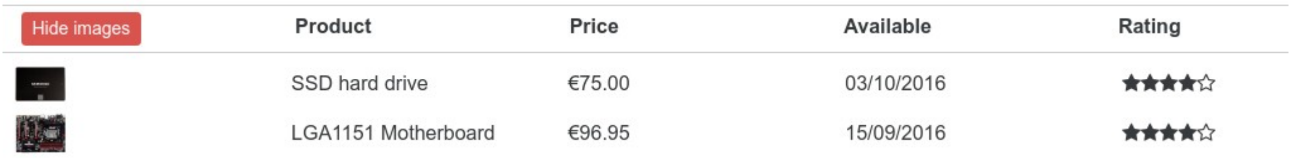
To indicate that a component receives input data from a parent component, we use the **@input()** decorator before the property declaration. This tells angular, that this property should be taken from an attribute with the same name. This attribute must be placed in the component's selector element (in the parent's template). Let's see an example passing the product we want to display, and the showImage boolean value, to the **ProductItemComponent**.

```
#product-list.component.html

<product-item class="row no-gutters"
   [product]="product" [showImage]="showImages"
   *ngFor="let product of products | productFilter:filterSearch">
     Loading product...
</product-item>
```

We have created an input attribute named **product**, and set the product object as the value, and other attribute called **showImage**. Now, we just receive them in our **ProductItemComponent** using **@input()**:

```
import { Component, Input, OnInit } from '@angular/core';
import { IProduct } from './iproduct';

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

  @Input() product: IProduct;
  @Input() showImage: boolean;

  constructor() { }

  ngOnInit() { }

}
```

It's now showing the correct products!



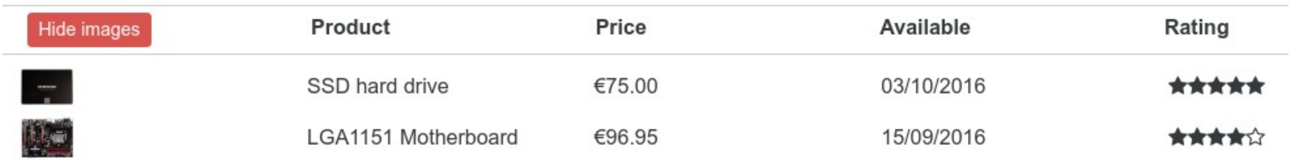
Hide images	Product	Price	Available	Rating
	SSD hard drive	€75.00	03/10/2016	★★★★☆
	LGA1151 Motherboard	€96.95	15/09/2016	★★★★☆

Finally, we'll do the same with **StarRatingComponent**, sending the rating from the product to this component:

```
#product-item.component.html

<div class="col">
    <star-rating [rating]="product.rating"></star-rating>
</div>
```

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

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

  @Input() rating: number;
  ...
}
```

And now it shows the correct rating for every product!



Hide images	Product	Price	Available	Rating
	SSD hard drive	€75.00	03/10/2016	★★★★★
	LGA1151 Motherboard	€96.95	15/09/2016	★★★★☆

Getting data from a child component → @Output

What if we need to modify something inside a component and inform the parent component?. For that purpose, we have the **@Output()** decorator using in combination with an **EventEmitter** object.

First of all, we want that when we go over a star with our mouse pointer, set a temporal rating corresponding to the position of that star. As it's usually **not recommended to modify @Input properties** (if we do that, they won't be binded to the parent anymore), we create an auxiliary private property called **auxRating**, and initialize it to the same value we receive from the parent (using **ngOnInit**).

We'll also create a method to restore **auxRating** to the input rating value:

```
export class StarRatingComponent implements OnInit {
  private auxRating: number;
  @Input() rating: number;

  constructor() { }

  restoreRating() {
    this.auxRating = this.rating;
  }

  ngOnInit() {
    this.restoreRating();
  }
}
```

In the template, when the mouse pointer enters a star (**mouseenter** event), we'll change the **auxRating** value (which is the one used now for displaying stars), to the corresponding star position. Once the mouse goes out from the component (**mouseleave** event), **auxRating** value will be restored again, taking parent's value.

```
<div class="star-container" (mouseleave)="restoreRating()">
  <span *ngFor="let star of [1,2,3,4,5]" class="fa"
    [ngClass]="{'fa-star': star <= auxRating, 'fa-star-o': star > auxRating}"
    (mouseenter)="auxRating = star"></span>
</div>
```

It works, but now we need to send a message to the parent, so when we click a star, the product's rating really changes:

```
<div class="star-container" (mouseleave)="restoreRating()">
  <span *ngFor="let star of [1,2,3,4,5]" class="fa"
    [ngClass]="{'fa-star': star <= auxRating, 'fa-star-o': star > auxRating}"
    (mouseenter)="auxRating = star" (click)="setRating()></span>
</div>
```

Now, this **setRating()** method must emit an event to the parent component sending the new rating value. To do that, we create an **EventEmitter<number>** (because we want to send a **number**) property, with the **@Output()** decorator. Using that property, we can emit a value when the user clicks on a star:

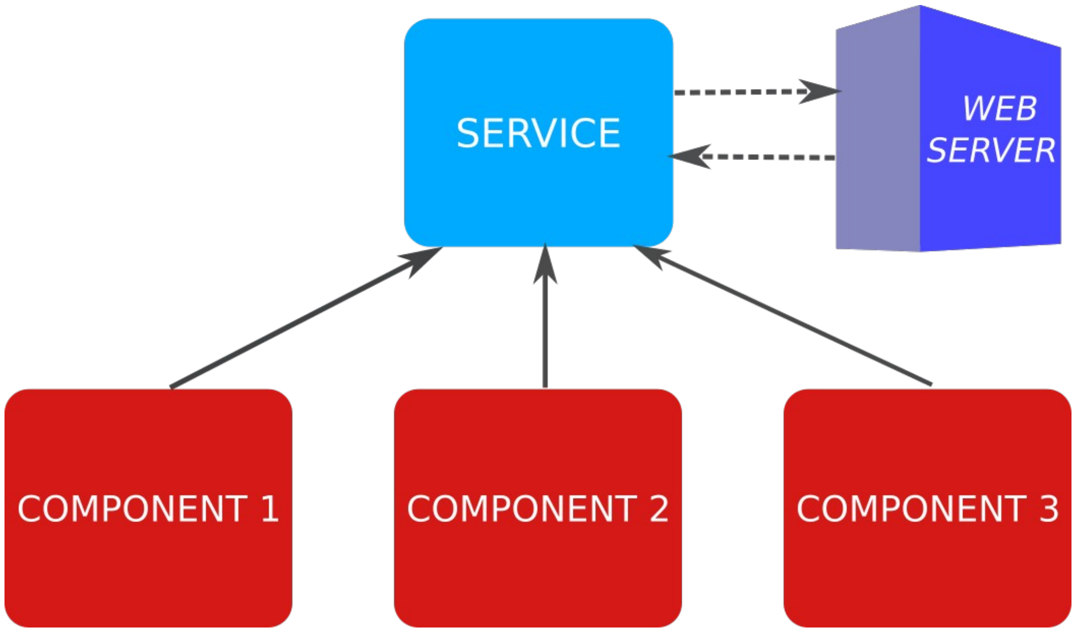
```
import { Component, Input, Output, EventEmitter, OnInit } from '@angular/core';

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

  ...
  @Output() ratingChanged = new EventEmitter<number>();
  ...
  setRating() {
    this.ratingChanged.emit(this.auxRating);
  }
  ...

}
```

Then, in the parent component, we listen to this event (**ratingChanged**), we obtain the value emitted (using **$event** special variable), and this will work perfectly.

```
<div class="col-xs">
  <star-rating [rating]="product.rating"
       (ratingChanged)="changeRating($event)"></star-rating>
</div>

export class ProductItemComponent implements OnInit {
  ...
  changeRating(rating: number) {
    this.product.rating = rating;
  }
  ...
}
```

Services. Dependency Injection.

A service is a class whose purpose is to provide some data and logic shared between different components. They're also used to access external data, using for example web services. When angular detects a component wants to use a service, the service injector will give that component a service class' instance.



In our application example, we could use a server to store product's data (and later retrieve them from a web server). Create a directory named **services**, and place our new service there. So, we go to that directory (src/app/services) and execute:

```
ng g service services/products
```

This will create a file called **products.service.ts** with our new service. This service is decorated with **@Injectable()** to tell Angular's dependency injector that it can "give" this service to any component (or another type of class) that requests it.

Inside the @Injectable decorator, the property **providedIn** means where should be the service injected. By default is 'root', which means it's available to all the components in the application. We can specify a module instead (we'll learn more about that in the future).

```
@Injectable({
  providedIn: 'root'
})
```

Now, let's add a method to our service that simply returns the array of product's (in the future we'll get them from a web server):

```
import { Injectable } from '@angular/core';
import { IProduct } from '../product-item/iproduct';
```

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

  constructor() { }

  getProducts(): IProduct[] {
    return [{
      id: 1,
      description: 'SSD hard drive',
      available: new Date('2016-10-03'),
      price: 75,
      imageUrl: 'assets/ssd.jpg',
      rating: 5
    }, {
      id: 2,
      description: 'LGA1151 Motherboard',
      available: new Date('2016-09-15'),
      price: 96.95,
      imageUrl: 'assets/motherboard.jpg',
      rating: 4
    }];
  }
}
```

Now that our products are in this service, we need our ProductListComponent to get its products from here. To inject this service in our component, we'll use a feature from TypeScript, creating a private attribute directly as a constructor's parameter. Just by indicating the type of the parameter (**ProductService**), Angular will assign to that variable the instance of our service's class automatically (that's dependency injection).

Finally, we'll use the **ngOnInit** method to call the service's method an get our product list. Notice that our product list is empty until we call this method.

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

import { IProduct } from '../product-item/iproduct';
import { ProductService } from '../shared/product.service';

...
export class ProductListComponent implements OnInit {
  ...
  products: IProduct[] = [];

  ...
  constructor(private productService: ProductService) { }
  ...

  ngOnInit() {
    this.products = this.productService.getProducts();
  }
}
```

The application should run as before, but this time our products are not in the ProductListComponent, and every component that wants to access the product's list can do it now easily.