UNIT 2

Angular

Part 4
Directives. Modules. Forms.

Client-side Web Development
2nd course – DAW
IES San Vicente 20120/2021
Author: Arturo Bernal Mayordomo

Index

Angular directives

Directives are custom elements or attributes (not standard) used by Angular to process the templates (HTML). There are 3 types of directives.

Component's directives

A component's directive is defined by the selector attribute in the @Component decorator. When an element (tag) is put in the HTML template and has this selector, an instance of the Component's class is created and its template is placed inside.

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

```
<star-rating *ngIf="product" [rating]="product.rating"
             (ratingChanged)="changeRating($event)"></star-rating>
```

Attribute directives

Attribute directives are custom attributes created to modify an element's behavior. Angular's directives NgClass and NgStyle are examples of these. We can also create custom attribute directives using the **@Directive** decorator with a class.

In this example, we'll create a directive to change the background color of an element when we click on it. First, we create the directive:

```
ng g directive set-color
```

```
@Directive({
  selector: '[setColor]'
})
export class SetColorDirective {
  constructor() { }
}
```

We must make sure that the directive is included in the declarations section of our module (like the components) in order to use it.

```
<h1 [setColor]="'yellow'">{{title}}</h1>
```

When an element has the **[setColor]** attribute, the directive has effect on it. We can pass parameters the same way as with nested components (**@Input**, **@Output**). The main difference is that we can define events like '**click**', '**mouseenter**', etc. with the decorator **@HostListener**.

```
# app.component.ts
@Component({
  ...
})
```

```
export class AppComponent {
  title = 'app works!';
  color = 'yellow';
}
```

app.component.html

```
<h1 [setColor]="color">{{title}}</h1>
<select [(ngModel)]="color">
  <option value="yellow">Yellow</option>
  <option value="red">Red</option>
  <option value="#06F">Blue</option>
</select>
```

set-color.directive.ts

```
@Directive({
  selector: '[setColor]'
})
export class SetColorDirective {
  private isSet: boolean;
  @Input('setColor') color: string;

  constructor(private elem: ElementRef) {
    this.isSet = false;
  }

  @HostListener('click') onClick() {
    if(this.isSet) {
      this.elem.nativeElement.style.backgroundColor = "";
    } else {
      this.elem.nativeElement.style.backgroundColor = this.color;
    }
    this.isSet = !this.isSet;
  }
}
```

https://angular.io/api/core/HostListener

We can also bind attributes (also CSS classes and styles) using the HostBinding decorator. This way, our directive becames easier to manage.

```
@Directive({
  selector: '[setColor]'
})
export class SetColorDirective {
  private isSet: boolean;
  @Input('setColor') color: string;

  @HostBinding('style.background-color') bgColor: string;

  @HostListener('click') onClick() {
    if(this.isSet) {
      this.bgColor = "";
    } else {
      this.bgColor = this.color;
    }
    this.isSet = !this.isSet;
  }
}
```

https://angular.io/api/core/HostBinding

https://angular.io/docs/ts/latest/guide/attribute-directives.html

Structural directives

Structural directives are attributes that modify the DOM structure. They're usually used to define if an element is placed in the DOM or removed from it. Although we can create custom structural directives, it's not very common.

NgSwitch

Examples of these directives are **NgIf** → *ngIf) and **NgFor** → *ngFor. There's another called **NgSwitch** → [ngSwitch] that we haven't seen yet and its very similar to NgIf. It allows to create a switch structure in the HTML template.

```
<span [ngSwitch]="property">
  <span *ngSwitchCase="'val1'">Value 1</span>
  <span *ngSwitchCase="'val2'">Value 2</span>
  <span *ngSwitchCase="'val3'">Value 3</span>
  <span *ngSwitchDefault>Other value</span>
</span>
```

If the value is a string, it must go between single quotes ('val'), otherwise (number, boolean) you don't use them. Also, you can use other variable or constant instead of a literal value.

NgIf (else statement)

Structural directives use **<ng-template>** internally to show the required info. Using the asterisk(*) notation makes the use of <ng-template> unnecessary (but internally, it translates to that). This <ng-template> element disappears when the directive is processed:

```
<div *ngIf="condition" >{{somevalue}}</div>
```

```
<ng-template [ngIf]="condition">
  <div>{{somevalue}}</div>
</ng-template>
```

Directives like ***ngIf** need to use <ng-template> if we want to create the equivalent of an **else** block. We need to create a reference name to this element (#reference) and use it in the **else** statement (**ngIf="condition; else reference"**),

```
<div *ngIf="show; else elseBlock">Show when condition is true</div>
<ng-template #elseBlock>Show when condition is false</ng-template>
```

Creating a structural directive

Lets create a directive that repeats itself n (parameter) times. This directive is going to be called for example apRepeatTimes (assuming **ap** is our application prefix):

```
ng g directive repeat-times
```

This is how we are going to use it in our code:

```
<p *apRepeatTimes="3; let n = current">
  {{n}} welcome works!
</p>
```

To get the passed value (3) inside the structural directive, we need to define a setter that has the same names as the directive. Every time we call **createEmbeddedView**, we are putting the HTML inside this element

(**this.templateRef**) in the DOM. We can also pass variables to the template inside an object and reference them outside like you see (**let n = current**).

```
import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';

@Directive({
  selector: '[apRepeatTimes]'
})
export class RepeatTimesDirective {
  @Input()
  set apRepeatTimes(num: number) {
    this.viewContainer.clear(); // When input data changes resets content!

    for (let i = 0; i < num; i++) {
      this.viewContainer.createEmbeddedView(this.templateRef, {
        current: i + 1
      });
    }
  }

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) {}
}
```

1 welcome works!

2 welcome works!

3 welcome works!

Lets create another structural directive. This time very similar to ***ngFor** but also accepts a function that filters the array:

```
<p *apForFilter="let person from persons by filter">
  {{person | json}}
</p>

This is the filter

filter = (p) => p.age >= 18;
```

{ "name": "Clara", "age": 22 }

{ "name": "John", "age": 36 }

Now, there will be 2 main differences. The input we get comes preceded by **from** and **by** keywords. So, we need to create setters for **apForFilterFrom** and **apForFilterBy** to get the values (and generate the view when receiving the filter. Also, notice that the variable produced in every iteration (**let person ...**) gets the value from **$implicit**.

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[apForFilter]'
})
export class ForFilterDirective {
  @Input()
  set apForFilterFrom(array: any[]) {
    this.items = array;
  }

  @Input()
  set apForFilterBy(filter: (item: any) => boolean) {
    this.viewContainer.clear(); // When input data changes resets content!
    this.items.filter(filter).forEach(elem => {
      this.viewContainer.createEmbeddedView(this.templateRef, {
        $implicit: elem
      });
    });
  }
```

```
  items: any[] = [];

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) {}
}
```

ng-container

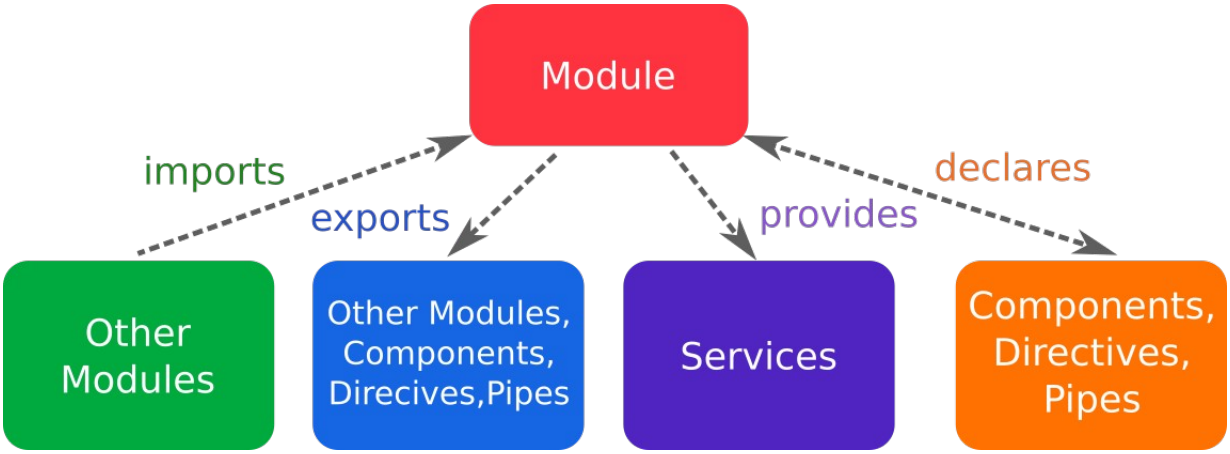
The **ng-container** element allows the use of structural directives. The main difference with a normal element is that this element disappears once it's been processed. It's very useful when you want to combine 2 structural directives, because it's not possible to do that on the same element:

```
<ng-container *ngFor="let person of persons">
  <ng-container *ngIf="person.age >= 18">
    <p>{{person | json}}</p>
  </ng-container>
</ng-container>
```

Angular modules

An angular module can be seen as a class with the **@NgModule** decorator. Until now, the only module we've defined is the **AppModule**. Everything (components, services, routes, pipes) is imported and used by that component. However, if our application grows more, it will become more and more difficult tom maintain because of all that centralization. Also, we won't be able to use a feature called **lazy loading**, which we'll see how it works soon, and makes our application perform better.

In this section, we are going to learn how we can separate our application functionality into different modules. And also, how these modules can be imported from other modules and also export things like components and even modules.



Boostrap array

As you know, an Angular application has a main module called **AppModule**, and a main component called **AppComponent**. The App module boostraps the App component, so it can be used in the **index.html** file as a starting pont for our application. This App component will be loaded in the **index.html** file's **<app-root>** selector, and the rest of the components will be loaded inside of it.

```
@NgModule({
  ...
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Declarations array

Every component, directive, or pipe must be **declared** inside an angular module so it can be used inside it. Also, any component or pipe **must belong to only one module**, they cannot be redeclared in other modules. Later we'll explain how to export those components so they can be used in other modules.

If we use a component's directive, other directive, or a pipe that's not declared in the present module or exported by an imported module, Angular won't be able to find it and will throw an error.

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

Exports array

The exports array allows us to share other imported modules, or declared components, directives, and pipes with other modules that want to import the present module (and its exported features).

It's important to note that we can export something without importing them. This happens when there is a module we don't want to use in the present module but we want to make it available to other modules. For example, a shared module made to export some common modules like the angular's FormsModule.

Never export a service. When services are provided in a module, they are injected in a root level, so they are immediately available to all the application.

Imports array

Angular modules can import other modules using the imports array. Some of the modules will be Angular modules (CommonModule, FormsModules, HttpModule,…), others will be third party modules (Boostrap, Angular material, …), while others will be our own created modules.

When a module imports another module, it will have access to all its exported features (exports array). It will also make available the possibility of routing to its components (as we'll see later). Import only the modules that are strictly necessary.

```
@NgModule({
  ...
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot(APP_ROUTES)
  ],
  ...
})
export class AppModule { }
```

Providers array

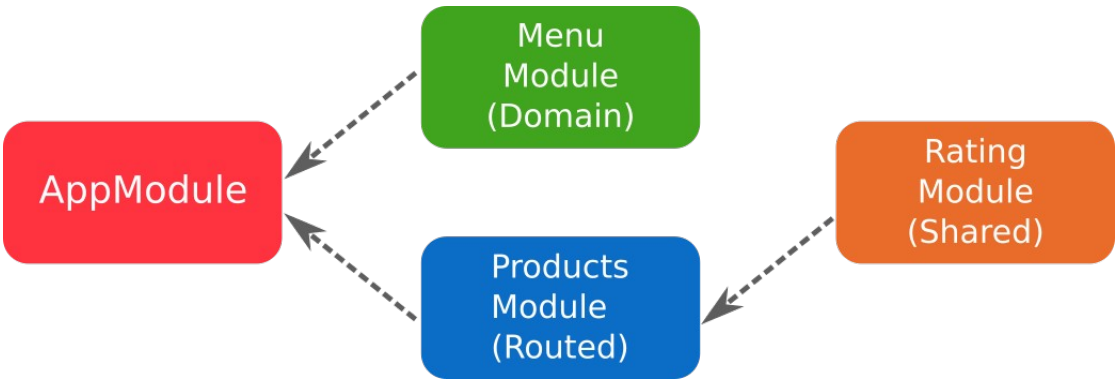
The providers array allows us to register services or other values to inject in the application. Usually, our services will auto-provide (provideIn: 'root'), so we won't need to include them in this array. All provided services and values (from any module) will be made available globally to all the application.

Dividing our application in different modules

Modules by features

First of all, we'll need to think how we could organize our application in modules the best way possible. We could divide our modules by features like this:

- **Domain modules**: Modules imported typically in the AppModule (no routes). An example of this would be a **Menu module**.
- **Routed modules**: Modules that represent a section of our application, but unlike domain modules, they're activated by the router. They usually handle related routes. For example a **Products module** could manage the routes and components for ProductsList, ProductDetail, ProductEdit, ….
- **Routing modules**: Module associated with a "Routed Module" that is used only to define the routes for that module.
- **Service modules**: Modules that provide common services that can be used anywhere in the application. Should be imported only once in the App module. They'll only have services or global values for being injected.
- **Widget or shared modules**: Modules that provide (and export) components, directives and pipes and other modules that can be used in different modules. For example our **StarRatingComponent** can be put in one of these modules.



More information → https://angular.io/guide/module-types

Creating the Products module

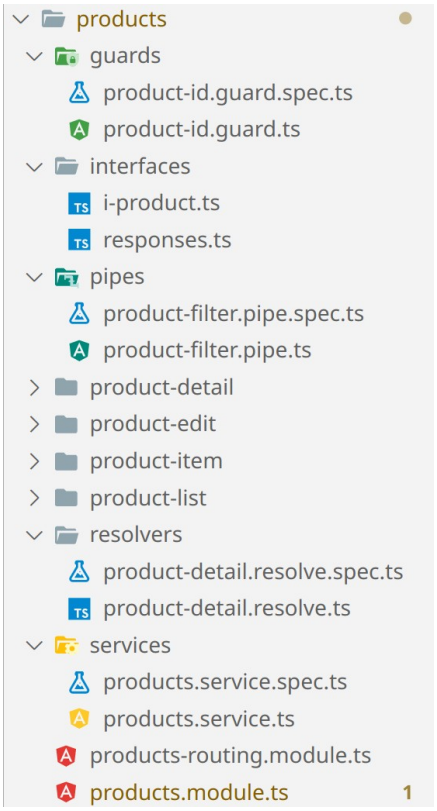
First of all, we are going to create our **ProductsModule** where we'll place our components and routes related to products. Everything declared (components, pipes, …) in this module will be placed inside a directory called '/**products**'. We can create the directory and module class by hand or by executing from 'app/src':

```
ng g module products --routing true
```

These are the contents of the created **products/products.module.ts** file:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class ProductsModule { }
```

Angular's **CommonModule** is similar to the BrowserModule imported in the AppModule. It allows use to use HTML related directives like ***ngIf**, ***ngFor**, etc. But the BrowserModule can only be imported in the AppModule and nowhere else.



Let's put all related products content inside the newly created directory, including ProductsService (it will only be used for products, so its logical it only loads when this module is also loaded) and specific CanActivate and Resolve guards:

As you can see, our files now seem more organized, with all logic related to products inside the new directory.

Now, it's time to fix our imports (check all code files) and adjust them to the new routes. Also, we are going to copy all the moved stuff from AppModule to ProductModule.

Also, we want to import **FormsModule** so we can use directives like **ngModule** and the **HttpModule** for our ProductService (they won't be necessary in the AppModule). Don't forget to declare the StarRatingComponent, used in the products components (we'll create a module for this component later). This is the result:

```
@NgModule({
  declarations: [
    ProductListComponent,
    ProductFilterPipe,
    ProductItemComponent,
    ProductDetailComponent,
    ProductEditComponent,
    StarRatingComponent,
    MinDateDirective,
  ],
  imports: [
    CommonModule,
    FormsModule,
    HttpClientModule,
    ProductsRoutingModule,
  ]
})
export class ProductsModule { }
```

Also, all the routes should be put in the companion routing module (**ProductsRoutingModule**):

```
const routes: Routes = [
  { path: 'products', component: ProductListComponent },
  {
    path: 'products/add',
    component: ProductFormComponent,
    canDeactivate: [PageLeaveGuard]
  },
  {
    path: 'products/:id',
    component: ProductDetailComponent,
    canActivate: [ProductIdGuard],
    resolve: {
      product: ProductDetailResolve
    }
  }
];

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

And this is the resulting **AppModule (**now with much less fat):

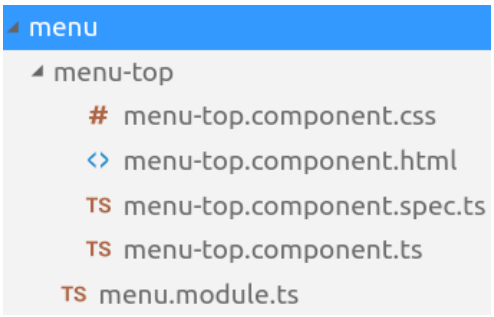
```
const APP_ROUTES: Route[] = [
  { path: 'welcome', component: WelcomeComponent },
  { path: '', component: WelcomeComponent },
  { path: '**', component: WelcomeComponent }
];

@NgModule({
  declarations: [
    AppComponent,
    WelcomeComponent,
  ],
  imports: [
    ProductsModule, // We must load the newly created module
    BrowserModule,
    RouterModule.forRoot(APP_ROUTES)
  ],
  providers: [
    Title,
    CanDeactivateGuard,
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Creating the Menu module

Let's modularize a little more our application by creating modules for the menu (also a menu component is needed) and rating.

As you can see, this module has only one component for the menu. It will be imported by the AppModule and used directly in the App Component (example of domain module).

Move the <nav> HTML to the new component's template and also the CSS if there's any. These are the resulting component and module files' contents:



menu-top.component.ts

```
@Component({
  selector: 'menu-top',
  templateUrl: './menu-top.component.html',
  styleUrls: ['./menu-top.component.css']
})
export class MenuTopComponent implements OnInit {
  @Input() title;

  constructor() { }

  ngOnInit() { }
}
```

menu.module.ts

```
@NgModule({
  declarations: [
    MenuTopComponent
  ],
  imports: [
    CommonModule,
    RouterModule
  ],
  exports: [
    MenuTopComponent // We need to export the component
  ]
})
export class MenuModule { }
```

In the AppModule, we must import this module and also use the <menu-top> selector in its template to load the menu component there (that's why it must be exported).

app.module.ts

```
...
@NgModule({
  ...
  imports: [
    MenuModule,
    ProductsModule,
    BrowserModule,
    RouterModule.forRoot(APP_ROUTES)
  ],
  ...
})
export class AppModule { }
```

app.component.html

```
<menu-top [title]="title"></menu-top>

<div class="container">
  <router-outlet></router-outlet>
</div>
```

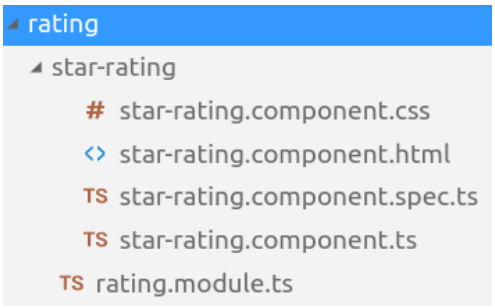
Creating the Rating Module

Now, we'll create a module for our rating system (for now only with the star-rating component, but we could create other kind of rating systems like, for example, a sliding bar).

The methodology is the same as before, but instead of importing it in the

AppModule, it will be imported in other modules like **ProductModule**.

This is an example of a shared module. Don't forget to export the component.



 # rating.module.ts

```
@NgModule({
  declarations: [
    StarRatingComponent
  ],
  imports: [
    CommonModule
  ],
  exports: [
    StarRatingComponent
  ]
})
export class RatingModule { }
```

 # products.module.ts

```
@NgModule({
  ...
  imports: [
    RatingModule,
    CommonModule,
    FormsModule,
    HttpClientModule,
    ProductsRoutingModule,
  ],
  ...
})
export class ProductsModule { }
```

Lazy loading modules

Basically, until now, we have divided our code into sections or modules. This helps to keep it more organized and maintainable. But there's one more advantage we are still not using and that's **lazy loading**. What we are doing now is called eager loading (all modules are loaded at the application start).

Lazy loading consist in loading only the necessary modules to start the application, and load other modules (products) only when necessary. In this case, when we activate a route that goes to any component in that module.

Activating lazy loading is very easy by default. Just go to AppModule and **remove ProductsModule from the imports array**, then tell the Angular Router that when a route starts with 'products', it must (lazy) load the ProductsModule and its components (but not before):

```
const APP_ROUTES: Route[] = [
 { path: 'welcome', component: WelcomeComponent },
 { path: 'products', loadChildren: () =>
      import('./products/products.module').then(m => m.ProductsModule)},
 { path: '', component: WelcomeComponent },
 { path: '**', component: WelcomeComponent }
];
```

Now, in the ProductsModule's routes, we don't need the '**products**' prefix as it

will be implicit, so **remove** it:

```
const routes: Routes = [
  { path: '', component: ProductListComponent },
  {
    path: 'add',
    component: ProductFormComponent,
    canDeactivate: [PageLeaveGuard]
  },
  {
    path: ':id',
    component: ProductDetailComponent,
    canActivate: [ProductIdGuard],
    resolve: {
      product: ProductDetailResolve
    }
  }
];
```

It's done. Everything inside ProductsModule will only be loaded when we go to a route starting by 'products' but not before. Our application should load faster now.

We could have created a lazy loaded products module from the beginning with this command:

ng g module products --route products --module app

Defining the lazy loading strategy (pre-loading)

The default lazy loading behavior is to load the module only when we go to a route inside it. There's a better strategy called **pre-loading**. This will load first the main module without the lazy loading modules (fast load time), but as soon as it finishes and loads the application, it will get in the background the rest of the modules from the server, so when we go to the 'products' route, for example, **ProductModule** will already be loaded resulting in faster loading times for its routes.

To activate pre-loading, we need to tell the angular router in the AppModule to apply the **PreloadAllModules** (@angular/router) strategy:

```
@NgModule({
  ...
  imports: [
    MenuModule,
    BrowserModule,
    RouterModule.forRoot(APP_ROUTES, {preloadingStrategy: PreloadAllModules})
  ],
  ...
})
export class AppModule { }
```

The default strategy is **NoPreloading**. We could use the quicklink library to preload only the modules we can access directly from the links in our current page: https://web.dev/route-preloading-in-angular/.

Angular forms (template driven)

Forms are usually an important part of a web application (or any graphical application). They allow us to create new data, update it, create a login form, etc. Angular has some interesting built-in features to manage forms, validate them and help showing errors and information to the user about its fields.

By default, the FormsModule is included in our Angular application (at least if it was created with Angular CLI). This module allows us to use some useful directives like **ngModel** or **ngForm**.

For this example, we are going to create a form for editing products (**products/edit/:id** route). First thing we are going to do is getting the product data for this component (**ProductEditComponent**) and we can do it with the same Resolve service we used for ProductDetailComponent → **ProductDetailResolve**.

```
  {
    path: 'edit/:id', component: ProductEditComponent,
    canActivate: [ProductDetailGuard], canDeactivate: [CanDeactivateGuard],
    resolve: {
      product: ProductDetailResolve
    }
  }
```

Then, we'll get our product in the component and we'll be ready to start:

```
export class ProductEditComponent implements OnInit, CanComponentDeactivate {
  product: IProduct;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.product = this.route.snapshot.data['product'];
  }

  canDeactivate() {
    return confirm("Are you sure you want to leave? Changes can be lost.");
  }
}
```

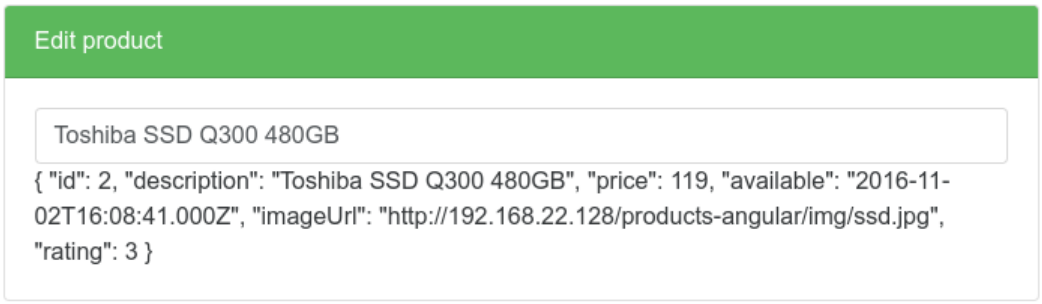
ngModel

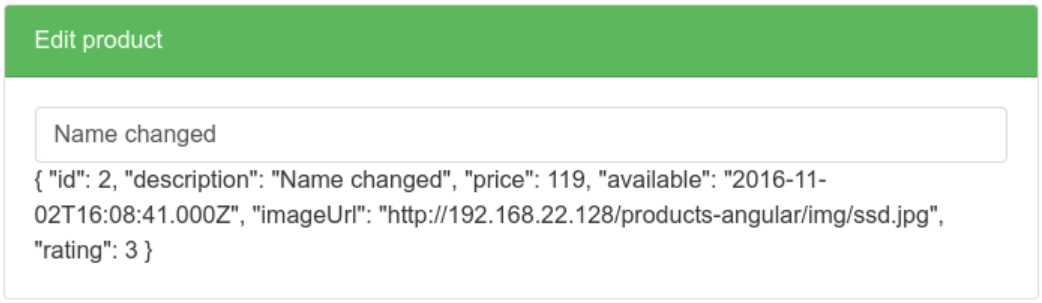
We've seen the **ngModel** directive as an example of 2 way binding data. This means that an <input> element which has this directive, will read its value from a property, and also modify it when the user changes its value.

We're going to create a standalone <input> for the product's description. We'll bind this input's value to the **product.description** property in the component and we'll show the product's information to see how it changes when the input element is also changed.

```
<div class="card">
  <div class="card-header bg-success text-white">
    Edit product
  </div>
  <div class="card-block">
```

```
    <div>
      <input type="text" class="form-control" [(ngModel)]="product.description">
    </div>
    <div>{{product | json}}</div>
  </div>
</div>
```





With form controls, we can use HTML5 validation attributes like **required**, **min**, **max**, pattern and other custom attributes like **minlength** and **maxlength**. These attributes are used by Angular to create some classes automatically in the element so we can add style to it.

```
<input type="text" class="form-control" [(ngModel)]="product.description"
       minlength="5" maxlength="60" required>
```


```
<input _ngcontent-qwe-9 class="form-control ng-untouched ng-pristine ng-valid"
maxlength="60" minlength="5" required type="text" ng-reflect-minlength="5" ng-
reflect-maxlength="60" ng-reflect-model="Toshiba SSD Q300 480GB">
```


```
<input _ngcontent-iut-9 class="form-control ng-touched ng-dirty ng-invalid"
maxlength="60" minlength="5" required type="text" ng-reflect-minlength="5" ng-
reflect-maxlength="60" ng-reflect-model="New">
```

State	Class if true	Class if false
Control has been visited	ng-touched	ng-untouched
Control's value has changed	ng-dirty	ng-pristine
Control's value is valid	ng-valid	ng-invalid

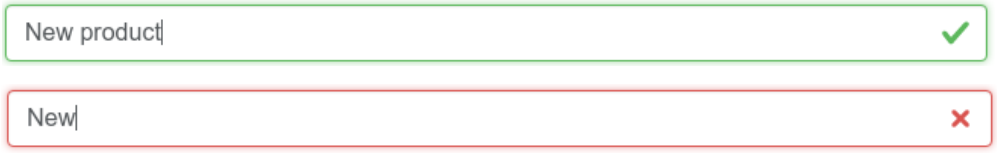
We can also create a reference to the ngModel directive's object. A reference is created with an attribute that starts with '#'. We can then use that reference to access

some of its properties and check things like if it has been changed (dirty), if it's validated (valid), or its value (value).

```
<div>
    <input type="text" class="form-control" [(ngModel)]="product.description"
    minlength="5" maxlength="60" required #descModel="ngModel">
</div>
<div>Dirty: {{descModel.dirty}}</div>
<div>Valid: {{descModel.valid}}</div>
<div>Value: {{descModel.value}}</div>
<div>Errors: {{descModel.errors | json}}</div>
```

We can use these classes and/or referenced properties to add style to our form control using CSS, ngClass, *ngIf, etc...(Bootstrap example):

```
<div class="form-group">
  <input type="text" class="form-control" [(ngModel)]="product.description"
    minlength="5" maxlength="60" required #descModel="ngModel"
   [ngClass]="{'is-valid': descModel.touched && descModel.valid,
              'is-invalid': descModel.touched && !descModel.valid}"></div>
```



If we put a form control inside a <form> element. A name is required, or the ngModel directive and validation won't work.

```
<form>
    ...
    <input type="text" name="description" ...>
    ...
</form>
```

Finally, we can filter what we are writing in the input control by dividing our

ngModel directive into **[ngModel]** and **(ngModelChange)**. $event represents the actual input text.

```
<input type="text" class="form-control"
       [ngModel]="product.description"
       (ngModelChange)="product.description = $event.toUpperCase()" ...>
```

ngForm

The **ngForm** directive is automatically applied to any **<form>** tag. We can include a reference to it the same way as we did before with an input's ngModel. From there we can check more or less the same conditions (dirty/pristine, touched/untouched, valid/invalid). A form is valid only if all its controls (with the ngModel directive) are valid. The value of a form is a JSON object containing the values of all its controls.

```
<form #productForm="ngForm" novalidate>
    <div class="form-group" ...>
        <input type="text" name="description" ...>
    </div>
</form>
<div>Touched: {{productForm.touched}}</div>
<div>Valid: {{productForm.valid}}</div>
<div>Value: {{productForm.value | json}}</div>
```

The **novalidate** attribute is recommended to deactivate the automatic browser validation (different experience depending the browser), leaving all the validation to be controlled by Angular.

Form controls

Every text <input> of type text, number, date, email, etc.. works more or less the same way. Angular will validate it and you can use that validation to style your form. In this example, the datetime-local must use a specially formatted string to get the initial value, and that's why we need to split [ngModel] → input and (ngModelChange) → output to do necessary transformations:

```
<form #productForm="ngForm" novalidate>
 <div class="form-group row">
   <label class="col-sm-2 col-form-label text-sm-right">Description</label>
   <div class="col-sm-10">
     <input type="text" class="form-control" name="description" #descModel="ngModel"
        [(ngModel)]="product.description" minlength="5" maxlength="60" required
        [ngClass]="validClasses(descModel, 'is-valid', 'is-invalid')">
   </div>
 </div>
 <div class="form-group row">
   <label class="col-sm-2 col-form-label text-sm-right">Price</label>
   <div class="col-sm-10">
     <input type="number" class="form-control" name="price" [(ngModel)]="product.price"
        min="0" step="0.01" required #priceModel="ngModel"
        [ngClass]="validClasses(priceModel, 'is-valid', 'is-invalid')">
   </div>
 </div>
 <div class="form-group row">
   <label class="col-sm-2 col-form-label text-sm-right">Available</label>
   <div class="col-sm-10">
     <input type="datetime-local" class="form-control" name="available"
        [ngModel]="product.available.toISOString().slice(0, 16)"
        (ngModelChange)="setProductDate($event)" #availModel="ngModel" required
        [ngClass]="validClasses(availModel, 'is-valid', 'is-invalid')">
   </div>
 </div>
 <div class="form-group row">
   <div class="offset-sm-2 col-sm-10">
     <button type="submit" class="btn btn-primary" [disabled]="productForm.invalid">
        Submit
```

```
      </button>
    </div>
  </div>
</form>
```

Also, the submit button will be **disabled** when the form is invalid.

If we want to use the **ngModel** directive with a **checkbox**, we'll need to make sure that the value is a boolean (true → checked, false → not checked).

```
<div class="form-check">
  <label class="form-check-label">
    <input class="form-check-input" type="checkbox" name="check"
           [(ngModel)]="booleanProperty">
    Checkbox example
  </label>
</div>
```

To create a group of **radio buttons**, just give them the same name and the same ngModel property binding, just change the value:

```
<div class="form-check">
  <label class="form-check-label">
    <input class="form-check-input" type="radio" name="radio" value="1"
           [(ngModel)]="someProperty">
    Option 1
  </label>
</div>
<div class="form-check">
 <label class="form-check-label">
    <input class="form-check-input" type="radio" name="radio" value="2"
           [(ngModel)]="someProperty">
    Option 2
  </label>
</div>
```

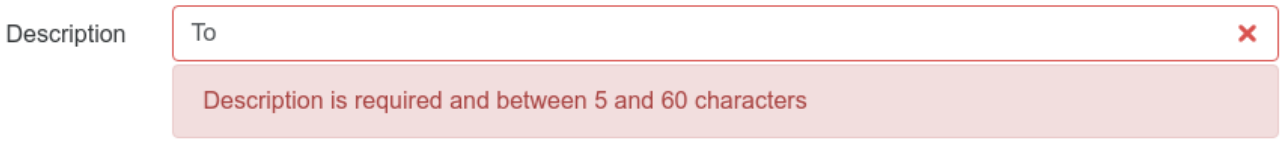
Finally, to bind ngModel to a **select** element, put in inside the <select> tag only and not inside any <option> tag.

Validation

As we have seen, we can use the properties present in the ngModel object and classes that are automatically assigned (.ng-valid, .ng-dirty, …) to show or hide error elements or to assign or not some classes.

```
<form #productForm="ngForm" novalidate>
  <div class="form-group row">
    <label for="inputEmail3" class="col-sm-2 col-form-label text-sm-right">
       Description
    </label>
```

```
        <div class="col-sm-10">
          <input type="text" class="form-control" name="description"
            [(ngModel)]="product.description" minlength="5" maxlength="60"
            required #descModel="ngModel" [ngClass]="validClasses(descModel,
'is-valid', 'is-invalid')">
        </div>
        <div class="offset-sm-2 col-sm-10">
          <div *ngIf="descModel.touched && descModel.invalid"
               class="alert alert-danger">
            Description is required and between 5 and 60 characters
          </div>
        </div>
      </div>
      ...
    </form>
```



Notice that a method called **validClasses** has been created. It's a way to reduce code in the HTML template. The 3 parameters are (1 → reference to the element's ngModel, 2 → class when validation is successful, 3 → class when validation fails).

```
  validClasses(ngModel: NgModel, validClass: string, errorClass: string) {
    return {
      [validClass]: ngModel.touched && ngModel.valid,
      [errorClass]: ngModel.touched && ngModel.invalid
    };
  }
```

To validate a **select** element's value, for example, when we use a default option that can't be selected, we can't do it automatically using classes (there's no HTML5 validation to use there). Instead, we can validate it by using events like **change**, **blur** or when the form's **submit** event happens → use **(ngSubmit)** event. The same happens with other elements in the form like file inputs, checkboxes or radio buttons.

Creating a custom validator

You can register a new validator that doesn't exist in Angular by creating a directive that implements the Validator interface. This directive's class will have a method called **validate** that will receive the current form control to validate. Lets create a validator that checks that a date input must be after another date.

```
ng g directive min-date
```

```
import { Directive, Input } from '@angular/core';
import { Validator, AbstractControl, NG_VALIDATORS } from '@angular/forms';

@Directive({
  selector: '[apMinDate]',
  providers: [{provide: NG_VALIDATORS, useExisting: DateAfterDirective, multi:
true}]
})
export class MinDateDirective implements Validator {
  @Input() apMinDate;

  validate(c: AbstractControl): { [key: string]: any } {
    if (this.apDateAfter && c.value) { // There's a date to validate
```

```
      const dateControl = new Date(c.value);
      const dateMin = new Date(this.apMinDate);
      if (dateMin > dateControl) {
        return {'minDate': true}; // Error returned
      }
    }

    return null; // No errors
  }

  constructor() {}
}
```

Register this directive in the AppModule or in a module imported directly from the AppModule (and only from here). And this is how we'd use it:

```
<div class="form-group col">
    <label for="filterDesc">Available:</label>
    <input type="date" [(ngModel)]="newProd.available"
           class="form-control" #availModel="ngModel"
           name="available" placeholder="Available date"
           required apMinDate="2017-09-01">
    {{availModel.status}} - Errors: {{availModel.errors | json}}
 </div>
```

Available:

dd/mm/aaaa

INVALID - Errors: { "required": true }

Available:

02/02/2014

INVALID - Errors: { " minDate ": true }

Available:

02/12/2018

VALID - Errors: null

Custom validators on control groups

If we need to create a validator that needs to access more than one input or form control, we can use the **ngModelGroup** directive in a parent element. This will create a FormGroup object with all the inputs inside. Those inputs must have different names. Then we create a validator named **fsOneChecked** (ng g directive one-checked) that checks if there's at least one checkbox checked.

```
<div class="form-group" ngModelGroup="daysGroup" #daysModel="ngModelGroup"
fsOneChecked>
  <div class="custom-control custom-control-inline custom-checkbox" *ngFor="let
day of days; let i = index">
    <!-- Inputs must have DIFFERENT names -->
    <input type="checkbox" class="custom-control-input" name="days{{i}}"
[(ngModel)]="daysOpen[i]">
    <label class="custom-control-label" for="checkDay{{i}}">{{day}}</label>
  </div>
  <div *ngIf="daysModel.invalid" class="alert alert-danger">
    You must select at least one input
  </div>
</div>
```

Inside the validation directive, we access its value. It will be an object with all the input values (a checkbox value is true or false) → Example: {days0: true, days1: true, days2: false, … }. We just have to check if every value is false (no checkbox is checked), and return an error.

```
export class OneCheckedDirective implements Validator {

  constructor() { }

  validate(group: AbstractControl): { [key: string]: any } {
```

```
    if (group instanceof FormGroup) {
      if (Object.values(group.value).every(v => v === false)) { // No checked
        return { 'oneChecked': true };
      }
    }

    return null; // No errors
  }
}
```

Sending a form

When sending a form after validating everything, we just need to send the properties binded to that form. We can also validate anything left there, include more data in the object we are going to send, and call the correspondent service's method.

```
export class ProductEditComponent implements OnInit, CanComponentDeactivate {
  ...
  submit() {
    // Other validations, etc... (call return if you want to cancel the submit)
    this.productService.updateProduct(this.product)
      .subscribe(
        ok => this.router.navigate([`/products/${this.product.id}`]),
        error => console.error(error)
      );
  }
}

export class ProductService {
  ...
  updateProduct(product: IProduct): Observable<boolean> {
    return this.http.put(this.productsURL + '/' + product.id, product)
      .pipe(map((response: Response) => {
        let respObj: { ok: boolean, error?: string } = response.json();
        if (!respObj.ok) throw respObj.error;
        return respObj.ok;
      }),
      catchError((response: Response) =>
              Observable.throw(`Error updating product ${product.id}!`)));
  }
}
```

Checking if the form is valid before sending

From the component

To access the form and/or any input validation state, we can reference their NgModel and NgForm objects using the decorator **@ViewChild()** and the name of the reference created with '#'.

```
<form class="form m-3" (ngSubmit)="addProduct()" #addForm="ngForm">

export class ProductAddComponent implements OnInit {
    @ViewChild('addForm') addForm: NgForm;
    ...

    addProduct() {
        if(this.addForm.valid) {
            this.productService.addProduct(this.newProd).subscribe(
                ok => {
                    this.router.navigate(['/products']);
                },
                error => console.error(error)
```

```
            );
        }
    }
    ...
}
```

From the template

We could use the same reference to the NgForm object in the HTML template and disable the submit button until the form is valid.

```
<button type="submit" class="btn btn-primary mr-3"
        [disabled]="addForm.invalid">Add Product</button>
```

Uploading a image in the JSON object

We can't use Angular's forms module with file inputs right now (there are some plugins to do this we'll review in the next part of this unit). However, we can transform a file into base64 encoding and send it in the uploaded JSON object. When using the element **<input type="file">**, the event we want to watch is **change**.

We need to pass the event method a reference to the input's object. To create a reference, just create an attribute with its name starting with '#' (don't assign any value this time).

```
<div class="form-group row">
  <label for="inputEmail3" class="col-sm-2 col-form-label text-sm-right">
      Image</label>
  <div class="col-sm-10">
    <input type="file" #fileImage class="form-control" accept="image/*"
           (change)="changeImage(fileImage)">
  </div>
</div>
<div class="row">
  <div class="col-sm-10 offset-sm-2">
    <img class="product-img" [src]="product.imageUrl">
  </div>
</div>
```

This is the implemented method. We just have to change **product.imageUrl** property once the file processing finishes.

```
changeImage(fileInput: HTMLInputElement) {
  if(!fileInput.files || fileInput.files.length === 0) return;

  let reader:FileReader = new FileReader();
  reader.readAsDataURL(fileInput.files[0]);
  reader.addEventListener('loadend', e => {
    this.product.imageUrl = reader.result;
  });
}
```

If you want to check if any input with **type="file"** has a file selected, you can't use a reference to the ngModel object (these inputs have no ngModel object associated). You can use a normal reference (#fileImage) to the HTML object and check the **files** array length.

```
<button type="submit" class="btn btn-primary mr-3" [disabled]="addForm.invalid
|| !fileImage.files.length">Add Product</button>
```