UNIT 2

Angular

Part 6
Animations. Reactive Forms. Testing. Internationalization. Angular Universal.

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

Index

Angular animations

Angular Animations is a module that allows you to create animations with native performance based on the Web animations standard. To enable this feature, just import this module in your application module:

```
import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

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

Animations are based on **triggers** (change from one state to another). To attach a trigger to an HTML element use the @ character before the name of that trigger.

```
<!-- this animates when `someStateValue` changes -->
<div [@trigger]="someStateValue">
  ...
</div>

<!-- this can animate on enter/leave -->
<div @trigger *ngIf="exp">
  ...
</div>
```

In the component's decorator we'll establish animations for that trigger. If we want to animate some element when it's inserted in the DOM, we could do this:

```
@Component({
    ...
    animations: [
        trigger('animateList', [
            transition(':enter', [
                style({ opacity: 0, transform: 'translateX(-100px)' }),
                animate('500ms ease-out', style({ opacity: 1, transform: 'none' }))
            ])
        ])
    ]
})
```

This tells angular that when the item with the trigger '**@animateList**' is inserted it should go from invisible (opacity: 0) and from the right (translate: '-100px'), to be visible and on its true position in 500ms. We can also use the '**:leave**' trigger to animate an element when it's removed from the DOM.

Transitions between states

Apart from entering / leaving the DOM, we can establish custom states and transitions (animations) between those states. You can use the special states *wildcard* (*) to indicate any state and void to indicate "not inserted in the DOM". To specify transitions between states we use the syntax '**state1 => state2**'.

```
    <ap-product-item [@animateList]="prod.state" (click)="toggleSelect(prod)"
    ...></ap-product-item>

  toggleSelect(prod: IProduct) {
      prod.state = prod.state === 'selected' ? '' : 'selected';
  }

  animations: [
      trigger('animateList', [
          state('selected', style({ borderLeft: '40px lightgreen solid' })),
          transition('* => selected', animate('200ms ease-in')),
          transition('selected => *', animate('200ms ease-out')),
          ...
      ])
  ]
```

We've established 2 new transitions: from any state (*) to selected and from selected to any state (not selected).

If you want to animate a list of items with a delay between them, it has to be a parent element who controls that animation.

```
<div *ngIf="products.length" @productList>
    ...
  <product-item ...>
  </product-item>
</div>
```

Using the query function we can select any inner HTML element (or Angular component selector) an animate it. We can combine it with the stagger function to introduce a delay between children animations.

```
@Component({
  selector: 'product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css'],
  animations: [
    ...
    trigger('productList', [
      transition(':enter', [
        query('product-item', [
          style({ opacity: 0, transform: 'translateX(-100px)' }),
          stagger(100, animate('500ms ease-out', style({ opacity: 1, transform:
'none' })))
        ])
      ])
    ])
  ]
})
```

Animating routes

To animate transitions between pages, we'll add a trigger to the router's parent element. This trigger will get the state from a method in the app component.

```
<div class="container-fluid" [@routeAnimation]="getState(routerOutlet)">
  <router-outlet #routerOutlet="outlet"></router-outlet>
</div>
```

The **getState** method will receive the RouterOutlet object and examine the data of the current route to find if there's any animation state name.

```
export class AppComponent {
    getState(outlet: RouterOutlet) {
        // Returns the page animation name (or 'None' of it has no animation)
        return outlet.activatedRouteData.animation || 'None';
    }
}
```

And then, you just need to add the ‘animation’ state to the data attribute on every route you want to animate:

```
export const PRODUCT_ROUTES: Route[] = [
  {
    path: '',
    component: ProductListComponent,
    data: { animation: 'productListPage' }
  },
  {
    path: ':id',
    component: ProductDetailComponent,
    data: { animation: 'productDetailPage' },
    ...
  },
  ...
];
```

Finally, in the app component, we’ll add the necessary transitions between pages:

```
@Component({
    ...
    animations: [
        trigger('routeAnimation', [
            transition('productList => productDetail', [
                query(':enter, :leave', style({ position: 'absolute', width: '100%' })),
                query(':enter', style({ transform: 'translateX(100%)' })),
                group([
                    query(':leave', [
                        animate('0.5s', style({ transform: 'translateX(-100%)' })),
                    ]),
                    query(':enter', [
                        animate('0.5s', style({ transform: 'none' })),
                    ]),
                ])
            ]),
            transition('productDetail => productList', [
                query(':enter, :leave', style({ position: 'absolute', width: '100%' })),
                query(':enter', style({ transform: 'translateX(-100%)' })),
                group([
                    query(':leave', [
                        animate('0.5s', style({ transform: 'translateX(100%)' })),
                    ]),
                    query(':enter', [
                        animate('0.5s', style({ transform: 'none' })),
                    ]),
                ])
            ])
        ])
    ]
})
export class AppComponent {
    ...
}
```

More examples and info: http://animationsftw.in, https://www.linkedin.com/pulse/angular-5-animations-increment-decrement-eliran-elnasi, https://www.yearofmoo.com/2017/06/new-wave-of-animation-features.html

Reactive forms

While template-driven forms (which we've already seen) are easier to use and require less code, reactive forms are more flexible, dynamic and easier to test (unit testing). A form is represented internally by a **FormGroup** object, while any input is represented as a **FormControl** object.

To begin using Reactive forms, instead of the FormsModule imported for template forms, we'll need to import **ReactiveFormsModule**:

```
import { ReactiveFormsModule } from '@angular/forms';
...
@NgModule({
  ...
  imports: [
    ...
    ReactiveFormsModule
  ],
  ...
})
export class AppModule { }
```

Building a Reactive form

The first thing we need to do is create a **FormGroup** object that will represent the entire form. We can create more FormGroups inside that represent sections of that form. Every input element we need to track (validate, get its value, etc) needs to be represented by a **FormControl** object.

For example, if we want a simple form to register a new user with the inputs **name**, **email**, and **password**, we would create the form like this in the component:

```
export class AppComponent implements OnInit {
  userForm: FormGroup;
  title = '';

  ngOnInit() {
    this.userForm = new FormGroup({
      name: new FormControl(),
      email: new FormControl(),
      password: new FormControl()
    });
  }
```

To set a default value, we pass it to the FormControl constructor:

```
receiveInfo: new FormControl(true) // This will be a checkbox (checked = true)
```

In the template, we'll use the **formGroup** directive to bind the <form> element to the FormGroup created in the component, and **formControlName** to bind each input with the corresponding FormControl:

```
<form [formGroup]="userForm">
  <input type="text" placeholder="Name" formControlName="name">
  <input type="email" placeholder="email" formControlName="email">
  <input type="password" placeholder="Password" formControlName="password">
</form>
```

Having that FormGroup object in the component, we can access all its properties and children with no need to create any reference (#formRef=”ngForm”).

```
<p>{{userForm.value | json}}</p>
<p>Name modified: {{userForm.get('name').dirty}}</p>
```

We can set all values in a form using **setValue**. However, this implies setting values for **all** controls. To set values for some of them, use **patchValue**.

```
export class AppComponent implements OnInit {
  ...
  setDemoData() {
    this.userForm.setValue({
      name: 'Test user',
      email: 'test@test.com',
      password: 'test'
    });
  }
}
```

A simpler way to create a Reactive form is by using the **FormBuilder** service. This service simplifies the creation syntax.

```
export class AppComponent implements OnInit {
  ...
  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.userForm = this.fb.group({
      name: '', // Control name: initial value
      email: '',
      password: ''
    });
  }
  ...
}
```

Validation

As we said at the beginning, Reactive forms give more flexibility to control everything than template forms. This is specially true for validation. Instead of a fixed validation HTML attributes, we could set different validation depending on the user (normal user / admin) or depending of the value set on other input, etc..

Using the FormBuilder service, we specify the validators after the initial value (inside of an array). If you need to set more than one validator, pass an array of validators. Use the **Validators** class to access all possibilities.

```
  ngOnInit() {
    this.userForm = this.fb.group({
      name: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
      password: ['', [Validators.required, Validators.minLength(5)]]
    });
  }
```

Anytime we want, we can modify or remove validators for any FormControl element using methods **setValidators** and **clearValidators**. This will not cause immediate re-evaluation of the input, to force that, call **updateValueAndValidity**.

```
<form [formGroup]="userForm">
  <p><input type="text" placeholder="Name" formControlName="name"></p>
  <p><input type="email" placeholder="Email" formControlName="email"></p>
  <p><input type="tel" placeholder="Phone" formControlName="phone"></p>
  <p><input type="password" placeholder="Password"
formControlName="password"></p>
  <p>Notify by:
    <input type="radio" formControlName="notifications" value="email"
(change)="updateNotifMethod()"> Email
    <input type="radio" formControlName="notifications" value="phone"
(change)="updateNotifMethod()"> Phone
  </p>
</form>

export class AppComponent implements OnInit {
  userForm: FormGroup;
  title = '';

  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.userForm = this.fb.group({
      name: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
      phone: ['', Validators.pattern(/[0-9]{9,}/)],
      notifications: 'email',
      password: ['', [Validators.required, Validators.minLength(5)]]
    });
  }

  updateNotifMethod() {
    const notif: string = this.userForm.get('notifications').value;
    const phoneControl = this.userForm.get('phone');
    if (notif === 'phone') {
      phoneControl.setValidators([Validators.required,
                                  Validators.pattern(/[0-9]{9,}/)]);
    } else { // email (Phone not required)
      phoneControl.setValidators([Validators.pattern(/[0-9]{9,}/)]);
    }
    phoneControl.updateValueAndValidity();
  }
}
```

Custom validators

Writing custom validators is easier with Reactive forms, or at least doesn't require a directive, just a function (that we can place in a separate file and export). This is what the **minDate** validator we created in the template forms examples looks like as a function:

```
import { AbstractControl, ValidatorFn } from '@angular/forms';

export function minDateValidator(minInputDate: string): ValidatorFn {
  return (c: AbstractControl): { [key: string]: any } => {
    if (c.value) {
      const minDate   = new Date(minInputDate);
      const inputDate = new Date(c.value);
      console.log(minDate, inputDate);
      return minDate <= inputDate ? null : {'minDate': minDate.toLocaleDateString()};
    }
    return null;
  };
}
```

And how to use it when we create a form:

```
  ngOnInit() {
    this.userForm = this.fb.group({
      ...
      birthDate: ['', minDateValidator('1900-01-01')]
    });
  }
```

Grouping fields

You've already seen that the form is represented by a FormGroup object. Inside that form we can group controls into nested FormGroups to perform cross-field validation such as checking if a confirmation email field is equal to the original email.

```
export class AppComponent implements OnInit {
  userForm: FormGroup;
  title = '';

  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.userForm = this.fb.group({
      name: ['', Validators.required],
      emailGroup: this.fb.group({
        email: ['', [Validators.required, Validators.email]],
        emailConfirm: ['', [Validators.required, Validators.email]]
      }, { validator: matchEmail }),
      ...
    });
  }
  ...
}

export function matchEmail(c: AbstractControl): { [key: string]: any } {
  const email = c.get('email').value;
  const email2 = c.get('emailConfirm').value;
  return email === email2 ? null : { match: true };
}

<form [formGroup]="userForm">
  <p><input type="text" placeholder="Name" formControlName="name"></p>
  <div formGroupName="emailGroup">
    <p><input type="email" placeholder="Email" formControlName="email"></p>
    <p><input type="email" placeholder="Repeat Email"
formControlName="emailConfirm"></p>
  </div>
  ...
</form>
```

Reacting to changes

Any FormGroup or FormControl has an observable called **valueChanges** that returns the new value (or values if it's a FormGroup) every time the value changes. For example, instead of using the **(change)** event in the HTML (notifications radio button), we can subscribe to that observable:

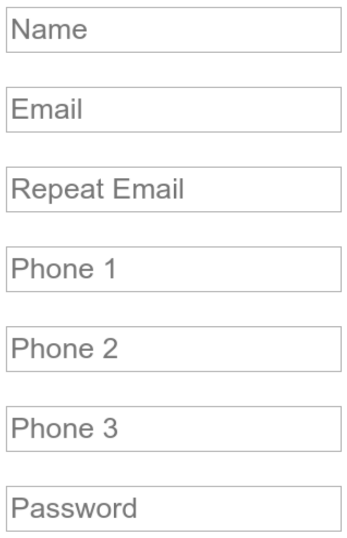
```
export class AppComponent implements OnInit {
  ...
  ngOnInit() {
    ...
    this.userForm.get('notifications').valueChanges
        .subscribe(notif => this.updateNotifMethod(notif));
  }
}
```

Duplicating input elements (FormArray)

A FormArray lets us duplicate any FormControl or FormGroup as many times as we want. For example, if we want a user to introduce multiple phones, addresses, etc.

```
export class AppComponent implements OnInit {
  userForm: FormGroup;
  phones: FormArray;
  ...

  ngOnInit() {
    this.userForm = this.fb.group({
      ...
      phones: this.fb.array([this.getPhoneControl()]),
      ...
    });

    this.phones = <FormArray>this.userForm.get('phones');
    ...
  }

  getPhoneControl(): FormControl {
    const control = this.fb.control('');
    control.setValidators(Validators.pattern(/[0-9]{9,}/));
    return control;
  }

  addPhone() {
    (<FormArray>this.userForm.get('phones')).push(this.getPhoneControl());
  }
  ...
}
```

```
<form [formGroup]="userForm">
  ...
  <p formArrayName="phones"
     *ngFor="let phone of phones.controls; let i = index">
    <input type="tel" placeholder="Phone {{i + 1}}" [formControlName]="i">
  </p>
  ...
</form>
<p><button (click)="addPhone()">Add Phone</button></p>
<p>{{userForm.value | json}}</p>
<p>Phone 1: {{phones.get('0').value}}</p>
```



Unit testing Angular apps

Unit testing usually refers to testing a class individually (in Angular, a class could be a component, a service, …). Unit tests are usually simple, fast and don't require the entire application to be working, as opposed to end to end tests.

Also unit tests are based on assertions (results you should expect when you do a certain operation). In fact, a unit test, checks only one assertion, so for every assertion you want to test, you should prepare a different unit test.

Finally, unit tests don't make external calls (Http services, databases, etc..).

When thinking about the structure of a unit test, keep in mind the letters 'AAA':

- **Arrange** → Create the initial state of the unit test (creating an object, establish initial data values).
- **Act** → Change something in the initial state (call a method, change a variable).
- **Assert** → Check that the expected result happens after you do any change.

Angular uses Jasmine as its unit testing framework. It uses 4 main functions to create a test (although there are more) :

- **describe** → Defines a suite (group) of unit tests.
- **beforeEach** → Common setup code for every test.
- **it** → Creates an individual unit test.
- **expect** → Function that checks if a test has passed. It uses matchers such as:
  - **toBe** → The value we expect: **expect(result).toBe(value)**.
  - **toBeDefined** → Expects a value not to be undefined.
  - **toContain, toBeGreaterThan, …**

Karma is the tool used to run those test. It launches a browsers and runs the tests on it and report the corresponding results. This tool is installed in

Isolated testing (testing a service)

Isolated tests just test a single class code, this means that they don't use any template, so these are the best to test services and pipes for example. You can test components and directives with isolated test, but only when you want to check the code and not what they produce on a template. In isolated test, you call the constructor directly, and they usually are very simple.

Angular CLI already creates a unit test file for every class in our project. Test files have the name **my.service.spec.ts** or **my.component.spect.ts** (note the **spec** suffix). It also adds some tools to instantiate a class object and a test that checks if the service/component/etc. has been correctly created. For example:

```
import { TestBed, inject } from '@angular/core/testing';
import { ProductService } from './product.service';

describe('ProductService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [ProductService]
    });
  });

  it('should be created', inject([ProductService], (service: ProductService) => {
    expect(service).toBeTruthy();
  }));
});
```

This file has only one test that checks if a service has been created and can be injected. The method **toBeThruthy** tests that a value is equivalent to true (not empty, or undefined, or null, etc.).

If our class needs to use other classes like services (remember that we're running unit tests an should only test one class), we can mock them, which means creating testing classes that return fixed values. Luckily, for some services like HttpClient, Angular already provides a testing service in **HttpClientTestingModule**.

If you look at the beginning o the test file, it creates a testing module using **Testbed.configureTestingModule**. Our class ProductService, needs the HttpClient service available. Usually this would be done by importing HttpClientModule, but we can't use the real service, so we'll import the testing module instead.

```
describe('ProductService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [ProductService],
      imports: [
        HttpClientTestingModule
      ]
    });
  });
  ...
});
```

Now, lets see how we can test a method of our service. In this case, we'll test the getProduct(id) method which, and see that it calls the right url with the right HTTP method, and also, if we return a product, there's no error in the observable returned.

```
  it('should get a product', inject([ProductService, HttpTestingController],
    (service: ProductService, backend: HttpTestingController) => {

    service.getProduct(1).subscribe(
        (prod) => expect(prod).toBeTruthy() // Gets something
    );

    backend.expectOne({ // Expects a GET call to this URL
            url: 'http://arturober.com/products-angular/products/1',
            method: 'GET'
        }).flush({ // Returns (fake) data to the service (OK response)
            product: {id: 1, name: 'Product', price: 12}
        }, {
            status: 200,
            statusText: 'Ok'
        });

    backend.verify(); // Needs to be called at the end
  }));
```

Integrated testing (testing a component)

These tests use the template when you're testing a component or a directive. Also, objects are constructed by the framework (you don't call the constructor). Integrated tests can be **deep** (testing multiple components like father and child components) or **shallow** (single component).

Lets see what's created by default by the Angular CLI to test a component:

```
import { DebugElement } from '@angular/core';
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { ProductItemComponent } from './product-item.component';

describe('ProductItemComponent', () => {
  let component: ProductItemComponent;
  let fixture: ComponentFixture<ProductItemComponent>;
  let debugElement: DebugElement;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ ProductItemComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(ProductItemComponent);
    component = fixture.componentInstance;
    debugElement = fixture.debugElement;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

First of all we're creating a new test module with **TestBed** like we did with our service, but in this case we're declaring a component. Second, we get a component **fixture**, from where we can get the component object (code) and the DebugElement object (template).

By default, Angular tests don't update the template when you change any variable on the component being tested. You have to call **fixture.detectChanges()** manually every time you change a value.

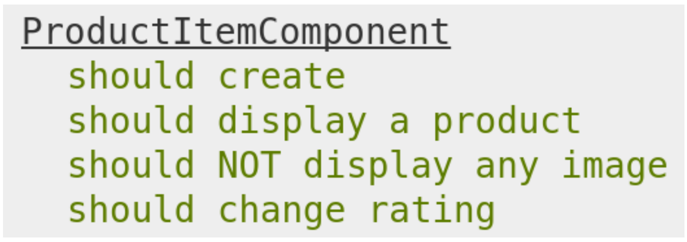
Lets create 2 tests. One that assigns a **product** an check that the description is in the HTML template, and other that assigns a product but sets the **showImage** property to false, so it shouldn't have the <img> element in the DOM.

```
describe('ProductItemComponent', () => {
  let component: ProductItemComponent;
  let fixture: ComponentFixture<ProductItemComponent>;
  let debugElement: DebugElement;

  const prod: IProduct = {
    id: 1,
    description: 'Product',
    available: '2019-01-01',
    price: 10,
    rating: 3,
    imageUrl: 'image.jpg'
  };

  beforeEach(
    ...
  );

  beforeEach(() => {
    ...
    component.product = prod; // Always set a product
    fixture.detectChanges(); // Update the template
    ...
  });

  ...

  it('should display a product', () => {
    const descAnchor: HTMLAnchorElement = debugElement.query(By.css('a'))
      .nativeElement;
    expect(descAnchor.innerText).toBe('Product');
  });

  it('should NOT display any image', () => {
    component.showImage = false;
    const img = debugElement.query(By.css('img'));
    expect(img).toBeFalsy();
  });
});
```

However, this still doesn't work. It doesn't recognize the **routerLink** directive (RouterModule) or the **StarRating** child component. The first problem is easy to solve by importing the **RouterTestingModule**. The second is also easy to solve by declaring the needed child component in the module.

One more step to go. Our component needs a service to be injected (**ProductService**), we're going to provide and inject it, but we need to replace the methods we're going to call (it's not recommended to make real HTTP calls during testing). Also, we're going to fake **changeRating** method with the **spyOn** function and return a simulated value instead of making a real HTTP call.

```
describe('ProductItemComponent', () => {
  let component: ProductItemComponent;
  let fixture: ComponentFixture<ProductItemComponent>;
  let debugElement: DebugElement;

  const prod: IProduct = {
    id: 1,
    description: 'Product',
    available: '2019-01-01',
    price: 10,
    rating: 3,
    imageUrl: 'image.jpg'
  };

  beforeEach(
    async(() => {
      TestBed.configureTestingModule({
        declarations: [ProductItemComponent, StarRatingComponent],
        providers: [
            ProductService,
        ],
        imports: [RouterTestingModule, HttpClientTestingModule]
      }).compileComponents();
    })
  );

  beforeEach(() => {
    fixture = TestBed.createComponent(ProductItemComponent);
    component = fixture.componentInstance;
    debugElement = fixture.debugElement;
    component.product = prod;
    fixture.detectChanges();
    const productService: ProductService =
debugElement.injector.get(ProductService);
    spyOn(productService, 'changeRating').and.returnValue(Observable.of(true));
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should display a product', () => {
    const descAnchor: HTMLAnchorElement = debugElement.query(By.css('a'))
      .nativeElement;
    expect(descAnchor.innerText).toBe('Product');
  });

  it('should NOT display any image', () => {
    component.showImage = false;
    const img = debugElement.query(By.css('img'));
    expect(img).toBeFalsy();
  });

  it('should change rating', async(() => { // Needs to be async (Observable)
    component.changeRating(5);
    fixture.whenStable().then(() => expect(component.product.rating).toBe(5));
  }));
});
```



```
ProductItemComponent
   should create
   should display a product
   should NOT display any image
   should change rating
```

More information: https://angular.io/guide/testing

Internationalization

Angular has built-in internationalization features that allows us to write multi-language apps. Internationalization is not only translating text but also handling date and currency formats or plural / genre of a word.

Locales in Angular are based on the standard BCP47. By default, Angular uses the “en-US” locale (United States English). To make Angular use other locale configuration, run it with the --locale option:

```
ng serve --aot --locale es-ES
```

This uses the specified locale configuration by default for your app. However, if you want to use more than one locale at the same time (real time language change for example), you’ll have to load it manually (in the app module file).

```
import { registerLocaleData } from '@angular/common';
import localeFr from '@angular/common/locales/fr';
import localeFrExtra from '@angular/common/locales/extra/fr';

registerLocaleData(localeFr, 'fr-FR', localeFrExtra);
```

The translation process for a template has 4 steps:

1. Mark static messages (i18n attribute) in your templates for translation

2. The Angular i18n tool extracts those messages into a standard translation source file.

3. A translator (person) edits that file, and translates the messages.

4. The Angular compiler imports the translated files, and replaces original messages generating a new version of the app in the target language.

Using the i18n attribute

First of all, we need to mark any element that contains text to be translated (you can create a span element to isolate text without any effect on the displayed view). You can do this with the **i18n** attribute.

```
<p><span i18n>Product description:</span> {{prod.description}}</p>
<!-- You can provide a description for the translator -->
<p><span  i18n="Label to describe a product">Product description:</span>
{{prod.description}}</p>
<!-- And also a meaning (meaning|description)  -->
<p><span i18n="product description|Label to describe a product">Product
description:</span> {{prod.description}}</p>
```

All text messages that have the same meaning, will have the same translation. If you repeat a text, but specify a different meaning, it could have different translations.

By default the i18n creates and assigns a different id for each translation unit (text message with the same meaning). However, you can specify a custom id for any i18n element using the prefix **@@my_id**. Don't use the same id on texts with different meanings because they will both be replaced with the same translation.

```
<p><span i18n="@@prodDesc">Product description:</span> {{prod.description}}</p>
<p><span  i18n="product label|Label to describe a product@@prodDesc">
Product description:</span> {{prod.description}}</p>
```

You can also generate translation for attributes like title using **i18n-attribute**:

```
<img title="Product image" i18n-title [src]="prod.image">
<img title="Product image" i18n-title="meaning|description@id"
[src]="prod.image">
```

Singular / plural

Angular uses the ICU Message Format syntax to define pluralization rules. This message is enclosed into single curly braces and has the following syntax:

{key, type, pattern}

- **key** is the component's property to bind
- **type** is the translation type (plural, select)
- **pattern** are the options provided.

```
<span i18n>Updated {minutes, plural, =0 {just now} =1 {one minute ago} other
{{{minutes}} minutes ago}}</span>
```

This example will show "just now" when minutes === 0, "one minute ago" when minutes === 1 or "n minutes ago" when minutes === n.

```
<span i18n>The author is {gender, select, m {male} f {female} o {other}}</span>
```

The example above will show "male" when gender === 'm', "female" when gender === 'f', or "other" when gender === 'o'.

Translating messages

To generate a translation file using the messages extracted with i18n run:

```
ng x18n
```

This will create a file named **src/messages.xlf**. This file is in XML format and consists of translations units. For example, the following element:

```
<label for="filterDesc" i18n="Product's description
label@@descLabel">Description:</label>
```

Generates this translation unit:

```
<trans-unit id="descLabel" datatype="html">
    <source>Description:</source>
    <context-group purpose="location">
       <context context-type="sourcefile">
```

```
          app/products/product-add/product-add.component.ts
       </context>
       <context context-type="linenumber">4</context>
    </context-group>
    <note priority="1" from="description">
        Product&apos;s description label
    </note>
</trans-unit>
```

Next, you should create the directory **src/locale** where we'll put the translated files. Copy the **messages.xlf** file there (you should have one copy for every language) and rename it to **messages.es.xlf** (Spanish translation). For every translation unit, generate a <target> element with the translation next to the <source> element:

```
<trans-unit id="descLabel" datatype="html">
    <source>Description:</source>
    <target>Descripción:</target>
    ...
</trans-unit>
```

Generating a translated app

To compile or serve a translated app, use the following options with **ng serve** or **ng build** (note that ng serve needs **--aot** flag. This won't be necessary in the future):

- **--i18nFile** → The path to the translation file.
- **--i18nFormat** → The format of the translation file.
- **--locale** → The locale id.

```
    ng serve --aot --i18nFile=src/locale/messages.es.xlf --
i18nFormat=xlf -locale=es-ES
```

Angular Server-side-rendering (Universal)

Angular Universal is a technology that allows to run your Angular application on the server (usually a NodeJS + Express server). These are the main reasons to create a Universal version of your application:

- Allows web crawlers to index your pages
    - Searching engines (Google, Bing, etc.) robots have very limited JavaScript capabilities, if any, so they can't execute Angular. With Universal, a rendered page with all content is served to these searching engines so they can index your contents.
- Improves performance on some mobile or low performance devices
    - For the same reason as above. Some of these devices have very limited JavaScript capabilities. The experience won't be complete, but it's the only solution.
- Show first page quickly
    - You can serve HTML pages already processed by Angular. These pages are static at the beginning until Angular loads and takes control. This is a better approach than showing an empty page or a loading screen for a few seconds (slow device) until Angular starts, because the user can see something immediately.

Creating a Universal application

Also, we need to install this module in our existing Angular project (--clientProject is the name of the current project):

```
ng add @nguniversal/express-engine --clientProject angular-products
```

Important: If you're running a non-stable version of Angular (@next), do the same with Universal:

```
ng add @nguniversal/express-engine@next --clientProject angular-
products
```

This will create the following files:

- **src/main.server.ts** → Starts the Angular app (server)
- **src/app/server.module.ts** → Main Angular module that loads in the server (This module imports the usual AppModule).
- **server.ts** → This file starts the express server. You can write this code in **TypeScript**.
- **tsconfig.server.json** → Configuration for the express TypeScript compiler.

- **webpack.server.config.js** → Webpack configuration for the express app.

Angular integrates very well with Express (however it's not the only framework we can use to serve Universal apps).

Running a Universal application

Run these commands and go to http://localhost:4000

```
npm run build:ssr && npm run serve:ssr
```

When running from localhost, you won't notice much difference, but if you upload your Universal app to an external server, specially if your network is slow, you should notice a lot of speed improvement. At least apparently to a user, because it will show you a full page rendered, but Angular will still take a few moments to fully load in the client.

Anywhere in your Angular code, you can check if it's running on server or client side. For example, when you're on server side, you don't have access to LocalStorage, but since you're running on Node, you can access other types of storage, for example:

```
constructor(
  @Inject(PLATFORM_ID) private platformId: Object,
  @Inject(APP_ID) private appId: string) {
  const platform = isPlatformBrowser(platformId) ?
    'on the server' : 'in the browser';
  console.log(`Running ${platform} with appId=${appId}`);
}
```