

UNIT 2

Angular



Part 1 **Introduction. Setup.** **Components. Templates.**

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

Index

Introduction.....	3
Setup an Angular project.....	4
Creating a new project with Angular CLI.....	4
How the Angular CLI works.....	4
Creating a new project.....	6
Application Bootstrap.....	7
Components.....	9
Templates.....	11
Components as directives.....	11
Creating a new component.....	11
Using a component selector in a template.....	13
Interpolation.....	13
Built-in directives (*ngFor, *ngIf).....	14
Data binding.....	18
Property binding.....	18
ngStyle and ngClass.....	19
Event binding.....	19
Two way binding [(ngModel)].....	20
Pipes.....	22

Introduction

Angular is a redesigned new framework for web application development. It's based on the popular AngularJS (version 1) framework, but it has changed a lot from that first version. Two of the most important advantages are that it's much faster than its predecessor and it's has a simplified API, there are fewer concepts to learn in Angular 2+ than there were in AngularJS.

Some of the most important features of Angular are:

- It makes our HTML code more expressive by using interpolation, data-binding, directives, etc.
- It's modular by design. We only use what we need from the framework at a given time. It also allows to design our application in a modular way, so the browser will only need to load part of it (Lazy Loading) and make loading faster.
- We can build reusable components in many applications easily.
- It makes very easy to integrate our application with the back-end (server), like a Restful API.
- Server side rendering with Angular Universal. This is a very useful feature to generate content on the server for search engines that don't understand Angular (none of them at this time).
- Powerful debugging tools. From TypeScript, to Chrome plugins like [Augury](#), and testing frameworks like [Karma](#) or [Jasmine](#).
- Integration with some design frameworks like Bootstrap, Angular Material (Google Material Design), and of course, Ionic.
- With angular, we build Single Page Applications (SPA), where the main page is loaded only once, and the content is refreshed by the framework.
- It's a modular framework, so we only include what we need. It's also well designed to build large applications.

To sum up, we are going to learn the new version of maybe the most popular JavaScript framework out there with more than 1 million developers. There's no doubt Angular will also become quickly popular an adopted as it's faster, smaller, cleaner and easier to learn than the first version!

Setup an Angular project

First of all, I recommend you install some extensions to improve the experience with Angular in Visual Studio Code. The Angular Essentials extension installs at the same time some useful extensions like snippets, template integration, etc.



Use the [Angular CLI](#). This is a command line tool that not only allows to generate Angular projects fast and easily, but also manages them. And most important, it has the necessary to generate a production application bundling and minifying your code, doing AoT (Ahead of Time) compilation, which is a new Angular 2+ feature, running automated testing, etc.

More information also in the [Angular quickstart guide](#).

Creating a new project with Angular CLI

First of all we'll need to install this tool with NPM:

```
npm i @angular/cli -g
```

If you want the latest (non-stable) version, which at the time of writing this is 9.0.0-rc.1 execute: **npm i @angular/cli@next**. You can check the installed version by running **ng --version**.

This will take a while as it has to download and install a lot of dependencies. Once it's installed we can create a new Angular project just by running:

```
ng new --strict PROJECT-NAME
```

This also will take time, because it has many NPM dependencies, but it will make our lives easier. The [Angular Strict Mode](#) activates [Typescript's Strict mode](#) and reduces final bundle size. If you don't want to generate test files, use the option **--skip-tests** (or **-st**). More information in the official [Angular CLI documentation](#).

You can now enter the new project's folder and execute **ng serve** to see if it works (open <http://localhost:4200> in your browser).

How the Angular CLI works

angular-cli is a new tool for creating and managing Angular 2+ projects. The new Angular framework brings a lot of improvements over the first version, but it

comes with a cost. It now depends on more libraries and tools and doing everything manually as you could do before it's really hard.

This tool makes our lives easier by automatizing all typical tasks and letting us focus only on coding, while Angular CLI does the rest.

Generating Components, Directives, Pipes and Services

Throughout this unit, we're going to explain the different parts that compose an Angular application. We can generate most of them using the Angular CLI. Just go to the folder where the project is, and execute the corresponding command (we still can generate our files and structure manually if we wanted to):

ng generate or **ng g** → more information in the [official documentation](#).

Component	ng g component my-new-component
Directive	ng g directive my-new-directive
Pipe	ng g pipe my-new-pipe
Service	ng g service my-new-service
Class	ng g class my-new-class
Interface	ng g interface my-new-interface
Enum	ng g enum my-new-enum
Module	ng g module my-module

Installing libraries

The **ng add** command installs an external dependency, like npm install, but with a difference: if a library is integrated with Angular by using [Schematics](#), it will configure automatically our project to work with that library. For example:

ng add @angular/material → Installs and configures Angular Material in our project.

Updating Angular version in a project

The ng update command will show us if there are new Angular versions and can update our project to the latest version. You can update Angular CLI executing **ng update @angular/cli** for example, or try to update all modules using **ng update --all**.

```
We analyzed your package.json, there are some packages to update:
```

Name	Version	Command to update
@angular/cli	7.1.3 -> 8.3.2	ng update @angular/cli
@angular/core	7.1.3 -> 8.2.4	ng update @angular/core
@angular/core	7.1.3 -> 7.2.15	ng update @angular/core
rxjs	6.3.3 -> 6.5.2	ng update rxjs

```
There might be additional packages that are outdated.  
Run "ng update --all" to try to update all at the same time.
```

The page update.angular.io can help you to update your project.

Building our project

Uploading all the files we have in our project to a web server is unnecessary and time consuming (see the size of the project). This is because there are a lot of development dependencies and other files (configuration for NPM, TypeScript, testing, libraries that are not imported in our code) that aren't necessary to run our application (just for developing).

If we build our application, the Angular CLI will use [WebPack](#) to bundle all our CSS and JavaScript into a few files (minified) resolving the necessary dependencies and leaving only the strictly necessary code to run our application. These files, ready to upload to a server, are placed in a directory called **dist/**.

The command used to build our application is: **ng build**. By default it builds a development environment application (it will be heavier but have additional information for debugging when an error appears). For production mode (once tested) you should run: **ng build --prod**.

The production flag (**--prod**) will generate a much smaller code (uglified) with no additional information and optimized code. So, when building for a production environment, always use **--prod**.

For more information: <https://github.com/angular/angular-cli>

Updating the Angular version

If you want to update the CLI for creating Angular projects with the newest version: **npm install -g @angular/cli**.

Updating the Angular version inside a project (visit: <https://update.angular.io/>), but generally, you'll have to execute: **ng update @angular/cli @angular/core**

Creating a new project

We're going to create a new project called **angular-products**:

```
arturo@arturo-desktop:~/Documentos/angular$ ng new angular-products
? Would you like to add Angular routing? No
```

When creating a new project, Angular CLI asks if we want to use Angular's router. We are going to say no (for now). We'll add that functionality later manually.

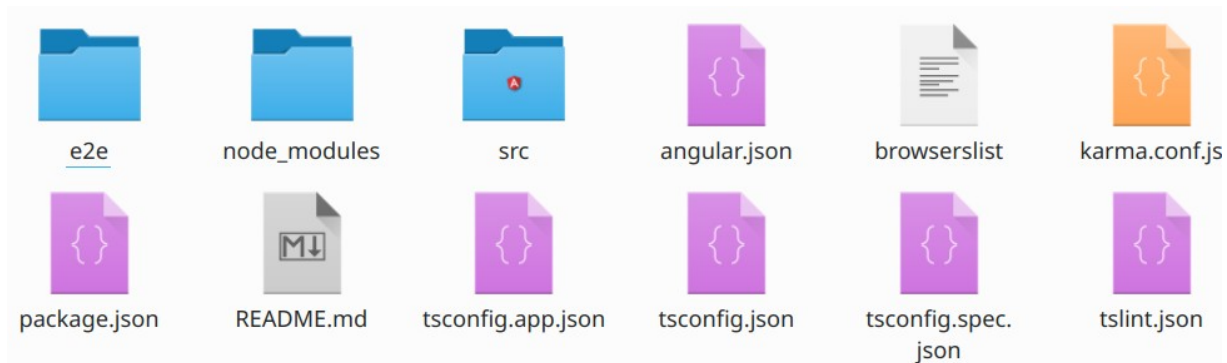
```
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
SCSS [ https://sass-lang.com/documentation/syntax#scss ]
Sass [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
Less [ http://lesscss.org ]
Stylus [ http://stylus-lang.com ]
```

The second question is about stylesheets. If we want to use plain CSS or a

preprocessor like SASS. Angular (using webpack) will compile automatically everything, so we won't have to worry.

Project's structure

Let's analyze a little the folders and files created by the Angular CLI and what were they created for.



In the main directory we can see the configuration files for **NPM** (package.json), **Angular CLI** (angular.json) and testing frameworks and tools like **Karma** (karma.conf.js): A tool for testing in the browser with frameworks like **Jasmine**. **Protractor** (e2e/protractor.conf.js): and End to End (E2E) testing framework. **TSLint** (tslint.json): A linter for debugging TypeScript code like ESLint for JavaScript.

We also have these subdirectories:

- **e2e**: This is where the End to End test are (Protractor).
- **node_modules**: NPM dependencies.
- **src**: This is where our Angular application is.

Application Bootstrap

An Angular application must have a main module (**@NgModule**), and can have also other submodules as we'll see when we arrive to **routing**. This will be very useful in order to load only the parts needed from our application in the browser when running the app for the first time (lazy loading).

Our main application module is created in the file **src/app/app.module.ts**.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
```

```

})
export class AppModule { }

```

First, let's analyze what the imports are for:

- **BrowserModule:** This Angular module has the necessary functionality to work with the browser (DOM manipulation, events, animations, ...). From version 2, Angular is not integrated with the DOM (it's a separated module), so it can work in other different environments (see [Angular + NativeScript](#)).
- **NgModule:** This imports the **@NgModule** decorator. Decorators are just functions (similar to annotations) which allow to specify class metadata Angular uses to configure a behavior for that class. They also can be used for methods. Common decorators we'll see are **@Component** or **@Injectable**.
- **AppComponent:** This is the main component of our application. All our app will be running inside this component as we'll soon see.

Now, we'll see what means the metadata in @NgModule

- **declarations** → **Components, directives** and **pipes** that are used in this module. As we'll soon see, when we create a new component class for example, we need to declare it here in order to use it.
- **imports** → Other modules whose (exported) classes will be used by this module. Here, we'll usually put the Angular modules we need in our application.
- **exports** → It's not present in this example. But, we can use this property to export classes (components, ...) from this module to be used by other modules.
- **providers** → Here we put global values or objects (like services) to be used globally in the app using dependency injection.
- **bootstrap** → Here we must put the main application view or component. This is the first thing our application loads and shows.

But how do we tell Angular that this is our main module and has to load it when the application starts?. This is done in the file **src/main.ts**:

```

import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

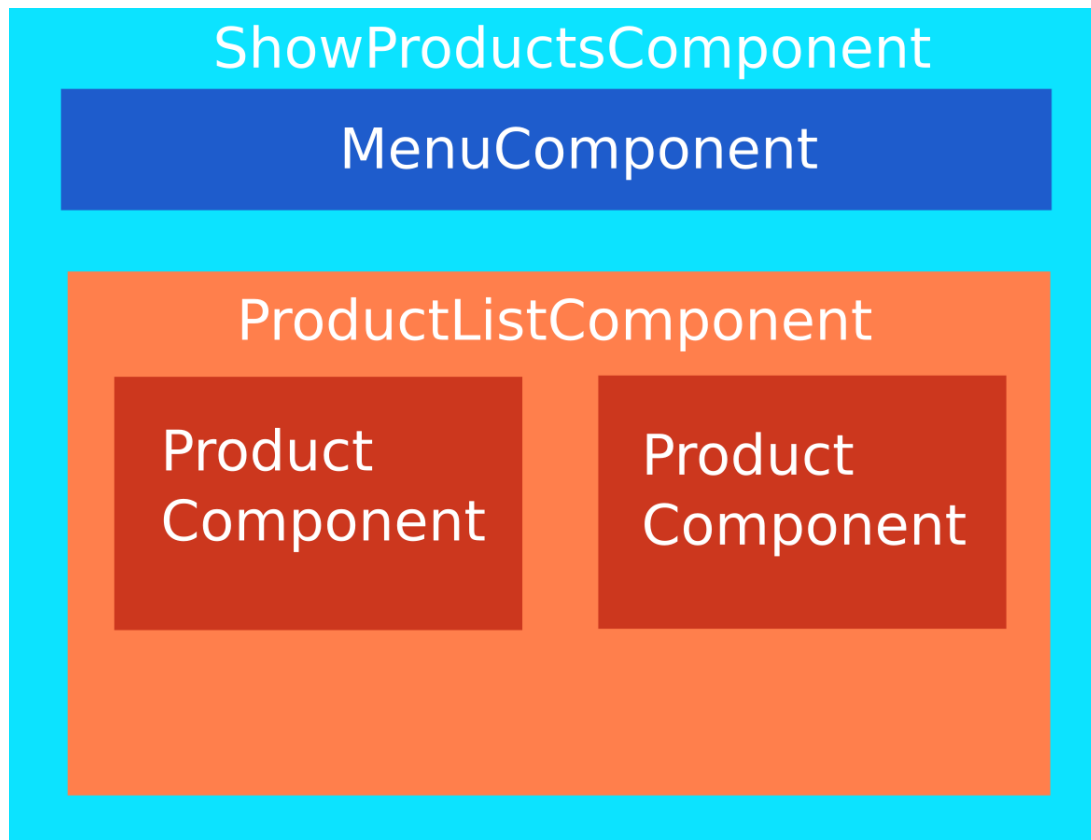
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(err));

```

We'll soon see that **index.html** is the main document, where all the web application will be loaded. This document will only be loaded once, but when using Angular routing service, users will feel that different web pages are being loaded.

Components

In Angular, a component can be understood as a controller class for a view (if we compare it to other MVC frameworks). There will be a main component (AppComponent), and also, there can be nested components (reusable in other parts of our application) controlling specific parts of the app:



When we created our project, a controller for the main view of our application was created (**src/app/app.component.ts**). Let's analyze it:

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

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

As we can see, a **component** is just a class with some metadata, like:

- **selector** → This is the **tag name** to be used in the view (HTML) where this component's template will be inserted inside. It's just a custom name. As HTML is case insensitive, uses '-' to separate words. Look at **src/index.html**, you will see the **<app-root>** tag. When Angular is loaded, it will replace the contents of

that tag with the component template.

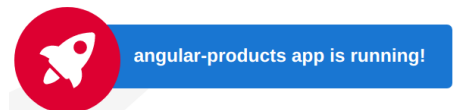
- **Note:** Custom selectors are just ignored by default by the browser (but not its contents), however, CSS styles can be applied to them.
- By default, every component is created with a selector prefixed with **app**. You can change this prefix (or delete it) when creating the project → **ng new my-project --prefix mp**. But you can do it later in **angular.json** (search for “**prefix**” attribute).
- **templateUrl** → This is the file where the view controlled by this component is. It will contain some HTML code (template) which will be inserted inside the component’s selector when it’s loaded.
- **styleUrls** → Angular allows you to assign one or more CSS stylesheets to a component. Those styles will be applied only to the component template (and everything inside of it). It allows us to create modular and reusable components with their own style.
 - Note: General styles for all the application can be included in the file **src/styles.css**.

Also, this class has a property called **title**, which is public (default) and its type is string (remember that if the type’s not declared, when you assign a value in the declaration, TypeScript binds that value’s type to the variable).

If you go to the template (**src/app/app.component.html**), you will see a lot of example content (which you can remove). Search for this):

```
<span>{{ title }} app is running!</span>
```

This is called **interpolation**, as we’ll see in the next section. When the component loads, it will change **{{title}}** for the ‘**app**’ text. Change the title property in the component class to see that’s true.



Instead of using an external template (HTML file), we could define the template in the component decorator as a string by using **template** instead of **templateUrl**:

```
@Component ({
  selector: 'app-root',
  template: `
    <h1>
      {{title}}
    </h1>`,
  styleUrls: ['./app.component.css']
})
```

Templates

If we consider our components to be controller classes, templates can be considered their views. A component is in charge of manipulate and interact with its template, and the way to do this is by using **interpolation** and **data binding**.

Components as directives

As we saw before, components have a metadata attribute called **selector**. This selector is placed where we want to load the template in our app. This selector can be used multiple times (for example, a product inside a product list), and each time, a different instance of the component class will be created. This is called **component directive**.

We saw that our **AppComponent** (its template) was included in the **index.html** file using the **<app-root>** tag. Let's create a new component class called **ProductListComponent** which will manage a list of products and include it in our **AppComponent** template. We can do it manually or using the Angular CLI tool.

To add style easily to our components, we are going to install bootstrap via NPM:

```
npm i bootstrap
```

Now, we only need (for now) to include the Bootstrap CSS file in our project. Instead of including it directly in the **index.html** file, we'll include it in our **src/styles.css** file. When we run **ng serve** or **ng build**, Webpack will include all in a single CSS file for us.

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

We can also include a CSS file in the **angular.json** configuration file, inside the "styles" array:

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

Creating a new component

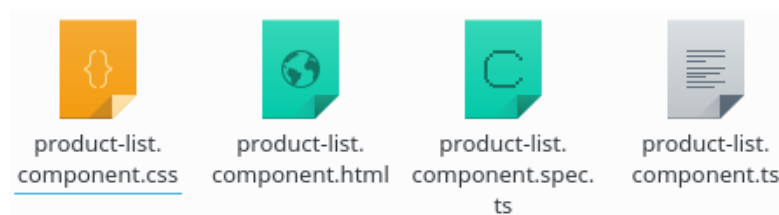
By default all components (and directives) selectors begin with the prefix '**app**'. If we're going to create reusable componentes, it's better to choose an appropriate prefix related to our components. If components are just for the application we're building, we can remove the prefix. To update/remove the prefix, in the **angular.cli.json** file we're going to establish an empty prefix for the component's selectors (by default: **app**).

```
"prefix": "",
```

Also edit **tslint.json** and remove the “**app**” prefix there for components and directives or VS Code will complain about it:

```
"directive-selector": [
  true,
  "attribute",
  "",
  "camelCase"
],
"component-selector": [
  true,
  "element",
  "",
  "kebab-case"
],
```

We can create components manually or by using **Angular CLI**. If we go to the **src/app** directory and execute **ng g component product-list**, it will generate a new directory there named **product-list** with these files in it:



This will also update the **app.module.ts** file to include this new component in our application:

```
...
import { AppComponent } from './app.component';
import { ProductListComponent } from './product-list/product-list.component';

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

The spec files (**product-list.component.spec.ts**) are created for unit testing purposes (karma, Jasmine).

This is what the component file (**product-list.component.ts**) looks like:

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

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

  constructor() { }

  ngOnInit() {
  }
}
```

We'll take a look at what **OnInit** means when we review **components life cycle**. For now, let's focus on other aspects.

We have a class that represents our component with a **@Component** decorator and its properties. Let's build a template (product-list.component.html) for our new component. We'll display a table (only header for now) where our future products will be listed:

```
<div class="card">
  <div class="card-header bg-primary text-white">
    My product's list
  </div>
  <div class="card-block">
    <div class="table-responsive">
      <table class="table table-striped">
        <thead>
          <tr>
            <th>Product</th>
            <th>Price</th>
            <th>Available</th>
          </tr>
        </thead>
        <tbody>
          <!-- Leave empty now -->
        </tbody>
      </table>
    </div>
  </div>
</div>
```

Using a component selector in a template

Now, we just have to include this **ProductListComponent** selector inside our **AppComponent** template (app.component.html) to display it (also change the **title** of the application to "Angular Products").

```
<div class="container">
  <h1>
    {{title}}
  </h1>
  <product-list></product-list>
</div>
```

If we execute **ng serve**, we should see this result:

Angular Products

My product's list		
Product	Price	Available

Interpolation

As we saw in the "[Components](#)" section, we can display the properties of our component class using interpolation `{{double curly brackets}}` in the template, and they'll be replaced with that property's value.

When doing interpolation, we can print any value: Concatenate values, properties, calling methods (from the component class), math calculations, etc.:

```
{{title}}
{{'Title: ' + title}}
{{'Title: ' + getTitle()}}
{{4*54+6/2}}
```

Let's put the Product's list title and table headings inside properties and display them in the template so we can change them easily, translate, etc.:

```
export class ProductListComponent implements OnInit {
  title = "My product's list";
  headers = {description: 'Product', price: 'Price', available: 'Available'};

  constructor() { }

  ngOnInit() { }
}
```

Now, using interpolation, we'll bind them in the template:

```
<div class="card">
  <div class="card-header bg-primary text-white">
    {{title}}
  </div>
  <div class="card-block">
    <div class="table-responsive">
      <table class="table table-striped">
        <thead>
          <tr>
            <th>{{headers.description}}</th>
            <th>{{headers.price}}</th>
            <th>{{headers.available}}</th>
          </tr>
        </thead>
        <tbody>
          <!-- Leave empty now -->
        </tbody>
      </table>
    </div>
  </div>
</div>
```

Built-in directives (*ngFor, *ngIf)

Now, we are going to add some products to our list. First, inside a different directory (src/app/interfaces/ for example), we create an interface for our products called **IProduct**, so we can type and use intellisense (auto-complete):

```
ng g interface interfaces/i-product
```

```
export interface IProduct {
  id?: number;
  description: string;
  price: number;
  available: Date;
  imageUrl: string;
  rating: number;
}
```

First, we are going to make the product's list table only appear when there are

actually products to show. If we have no products array or if it's empty, the table won't appear in the HTML (removed from the DOM). This is achieved using the ***ngIf** Angular built-in directive:

```
<div class="table-responsive" *ngIf="products && products.length">
  <table class="table table-striped">
    <thead>
      <tr>
        <th>{{headers.description}}</th>
        <th>{{headers.price}}</th>
        <th>{{headers.available}}</th>
      </tr>
    </thead>
    <tbody>
      <!-- Leave empty now -->
    </tbody>
  </table>
</div>
```

Try creating a property named products (IProduct[]), with no products to check that the table doesn't appear.

Finally, we are going to add some products. Let's create an array of **IProduct** inside **ProductListComponent**. Don't forget to in also include the interface created (extension is not needed). Images should be placed in the **assets** directory.

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

@Component({
  selector: 'product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent implements OnInit {
  title = "My product's list";
  headers = {description: 'Product', price: 'Price', available: 'Available'};

  products: IProduct[] = [{
    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
  }
  ];

  constructor() { }

  ngOnInit() { }
}
```

The directive ***ngFor** allows us to iterate through this **products** collection and generate the necessary HTML (table row) to display them.

```

<tbody>
  <tr *ngFor="let product of products">
    <td>{{product.description}}</td>
    <td>{{product.price}}</td>
    <td>{{product.available}}</td>
  </tr>
</tbody>

```

This is equivalent to a **foreach** statement. For each product in the list, a local variable called **product** will be created and have a product object assigned to it. The same HTML `<tr>` will be generated sequentially for every product.

Angular Products

My product's list		
Product	Price	Available
SSD hard drive	75	Mon Oct 03 2016 02:00:00 GMT+0200 (hora de verano de Europa central)
LGA1151 Motherboard	96.95	Thu Sep 15 2016 02:00:00 GMT+0200 (hora de verano de Europa central)

```

▼ <tbody _ngcontent-mew-2>
  <!--template bindings={
    "ng-reflect-ng-for-of": "[object Object],[object Object]"
  }-->
  ▼ <tr _ngcontent-mew-2>
    <td _ngcontent-mew-2>SSD hard drive</td>
    <td _ngcontent-mew-2>75</td>
    <td _ngcontent-mew-2>Mon Oct 03 2016 02:00:00 GMT+0200 (CEST)</td>
  </tr>
  ▼ <tr _ngcontent-mew-2>
    <td _ngcontent-mew-2>LGA1151 Motherboard</td>
    <td _ngcontent-mew-2>96.95</td>
    <td _ngcontent-mew-2>Thu Sep 15 2016 02:00:00 GMT+0200 (CEST)</td>
  </tr>
</tbody>

```

Dates and prices are still not well formatted (dd/mm/yyyy and 0.00€). When we arrive to the **Pipes** section, we'll see how Angular has some built-in functionality for this type of formatting.

There are some implicit values you can assign when implementing the ***ngFor** directive. And these are:

- **index: number:** The index of the current item.
- **first: boolean:** True when the item is the first item.
- **last: boolean:** True when the item is the last item.
- **even: boolean:** True when the item has an even index.
- **odd: boolean:** True when the item has an odd index.

To use one or more of these properties, you have to assign them to a variable like

in this example:

```
<tr *ngFor="let product of products; let i = index; let isEven = even">
  <td [ngClass]="{'even': isEven}">{{'Index: ' + i}}</td>
  ...
</tr>
```

Other way to assign those variables is using the “as” syntax:

```
<tr *ngFor="let product of products; index as i; even as isEven">
```

The truth is ***ngFor** (like all directives beginning with *) is a simpler syntax that allows us to use the **ngFor** directive in a more friendly way. What we just used would be translated internally into this (this syntax is equivalent):

```
<ng-template ngFor let-product [ngForOf]="products" let-i="index"
  let-isEven="even">
  <tr>
    <td [ngClass]="{'even': isEven}">{{'Index: ' + i}}</td>
    ...
  </tr>
</ng-template>
```

Data binding

Property binding

Property binding is the same concept as interpolation but applied to HTML elements properties. It will bind (one way) the component's property to an HTML element's attribute. To achieve this, we must enclose the attribute name between square brackets (ex: **[src]**) and put the component's property name as the value.

We are going to display our product's image in the first column of the table (Don't forget to add a new header cell, and include the images in the **assets** directory) using property binding for the **src** attribute:

```
<table class="table table-striped">
  <thead>
    <tr>
      <th>{{headers.image}}</th><th>{{headers.description}}</th>
      <th>{{headers.price}}</th><th>{{headers.available}}</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let product of products">
      <td><img [src]="product.imageUrl" alt=""></td>
      <td>{{product.description}}</td>
      <td>{{product.price}}</td>
      <td>{{product.available}}</td>
    </tr>
  </tbody>
</table>
```

In fact, you can use the interpolation syntax to bind a property. However, using the square brackets syntax is preferred.



```
<img [src]="product.imageUrl" alt=""> → Better!
<img src={{product.imageUrl}} alt="">
```

If our images are too big, we can limit their height for example. We can do it by using **style binding** like **[style.height.px]="imageHeight"** (Sets the image's size in pixels getting the value from a component's attribute → height: ...px;). Or better, using the component's CSS file → **src/app/product-list/product-list.component.css**:

```
td {
  vertical-align: middle;
}

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

Angular Products

My product's list			
Image	Product	Price	Available
	SSD hard drive	75	Mon Oct 03 2016 02:00:00 GMT+0200 (hora de verano de Europa central)
	LGA1151 Motherboard	96.95	Thu Sep 15 2016 02:00:00 GMT+0200 (hora de verano de Europa central)

ngStyle and ngClass

If you want to dynamically style a component based on certain values you can use the built-in **ngStyle** and **ngClass** directives.

The **ngStyle** directive receives an object as a value. This object consists of css properties (keys) and their values (which usually point to a variable or component property). This object can be specified inside the component or directly in the HTML.

```
<td [ngStyle]='{"background-color": isEven?'red':'green'}">...</td>
```

If we want to specify the units (px, em, mm, ...) we can add them with a suffix to the property name → property.units:

```
<td [ngStyle]='{"width.px": width}'>...</td> → width should have a numeric value
```

If the style object is big, we can store it inside the component and reference it from the **ngStyle** directive:

```
<td [ngStyle]="styleObject">...</td>
```

The **ngClass** directive is similar but simpler. It allows an element to have a certain CSS class (or not) based on a **boolean** property.

```
<td [ngClass]='{"even": isEven, 'last': isLast}'> → isEven and isLast: boolean
```

You can assign more than 1 class based on the value of a property:

```
<td [ngClass]='{"even": isEven, 'last active': isLast}'>
```

Like **ngStyle**, you can store the object as a component property and reference it from the directive:

```
<td [ngClass]="classObject">
```

```
@Component({
  ...
})
export class SomeComponent {
  ...
  classObject = {
    'class1': true,
    'class2': false
  }
  ...
}
```

Event binding



The opposite of property binding could be **event binding**. It can be considered the opposite because now the direction goes from the template (HTML element event) to the component class (calling a method).

To bind an event, we must use the event's name between parenthesis as an attribute like **(click)**, **(mouseenter)**, **(keypress)**, etc. In the example, we are going to create a button and when you click on it, images will hide and show (depending the current state).

```
<thead>
  <tr>
    <th>
      <button class="btn btn-sm"
        [ngClass]="{'btn-danger': showImage, 'btn-primary': !showImage}"
        (click)="toggleImage()" ">
        {{showImage?'Hide':'Show'}} images
      </button>
    </th>
    <th>{{headers.description}}</th>
    <th>{{headers.price}}</th>
    <th>{{headers.available}}</th>
  </tr>
</thead>
<tbody>
  <tr *ngFor="let product of products">
    <td>
      <img [src]="product.imageUrl" *ngIf="showImage" alt=""
        [title]="product.description">
    </td>
    <td>{{product.description}}</td>
    <td>{{product.price}}</td>
    <td>{{product.available}}</td>
  </tr>
</tbody>
```

The **[ngClass]** directive allows us to assign different classes to an element based on boolean attributes or conditionals. The click event on the button will call a method named **toggleImage()**.

```
showImage = true;
...
toggleImage() {
  this.showImage = !this.showImage;
}
```

Show images	Product	Price	Available
	SSD hard drive	75	Mon Oct 03 2016 02:00:00 GMT+0200 (CEST)
	LGA1151 Motherboard	96.95	Thu Sep 15 2016 02:00:00 GMT+0200 (CEST)
Hide images	Product	Price	Available
	SSD hard drive	75	Mon Oct 03 2016 02:00:00 GMT+0200 (CEST)
	LGA1151 Motherboard	96.95	Thu Sep 15 2016 02:00:00 GMT+0200 (CEST)

Two way binding [(ngModel)]

We've seen how to bind properties from the component to the template and bind events which go from the template to the component. There's also other option which goes both ways. It's a directive called **ngModel**, and it's usually used with input elements. To tell Angular that it goes in both directions, we must use square brackets and parenthesis at the same time → **[(ngModel)]**.

First of all, this directive is part of Angular Forms Module, so we'll have to include that module in our main App module:

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

This directive will allow you to modify a property's value in the component, but also, it will be read the first time to get the default value.

```
export class ProductListComponent implements OnInit {
  ...
  filterSearch = ''; // We can set a default value here
  ...
}
```

Filter: Filtered by: asdf



```
<div class="card-block">
  <form class="form mt-3">
    <div class="form-group row">
      <label class="col-form-label col-sm-2 text-sm-right"
        for="filterDesc">Filter:</label>
      <div class="col-sm-5">
        <input type="text" [(ngModel)]="filterSearch" class="form-control"
          name="filterDesc" id="filterDesc" placeholder="Filter...">
      </div>
      <label class="col-form-label col-sm-5">
        Filtered by: {{filterSearch}}</label>
      </div>
    </form>
    ...
  </div>
```

Pipes

Pipes are usually used with interpolation and data binding to transform the data before it's displayed. They can be considered as **filters**. Angular provides some [built-in pipes](#) for operations with strings, arrays, dates, JSON, currency, locales, ...



```
<tr *ngFor="let product of products">
  <td>
    <img [src]="product.imageUrl" *ngIf="showImage" alt=""
        [title]="product.description | uppercase">
    </td>
    <td>{{product.description}}</td>
    <td>{{product.price | currency}}</td>
    <td>{{product.available | date}}</td>
</tr>
```

Now, we are filtering the image title's to be displayed in uppercase. We are also applying a standard currency filter to the price and transforming the date object (**product.available**) into a more readable format.

	SSD HARD DRIVE	\$75.00	Oct 3, 2016
	LGA1151 MOTHERBOARD	\$96.95	Sep 15, 2016

Filters have also arguments (options) which can be passed to them concatenated (in order) using colons ':'. For example, we are going to tell the currency filter that our price is in **Euros (EUR)**, and it must display the coin symbol (€) → 'symbol' ([more info](#)). Also we are going to format the date in a dd/mm/yyyy format ([more info](#)).

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

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