



Tecnologie e applicazioni web

Angular

Filippo Bergamasco (filippo.bergamasco@unive.it)

<http://www.dais.unive.it/~bergamasco/>

DAIS - Università Ca'Foscari di Venezia

Academic year: 2021/2022

Angular

Angular (also known as Angular 2, currently at version 13) is a front-end framework for developing SPA web clients

- Developed by Google (Angular team)
- Open-source
- Based on TypeScript
- Modular



AngularJs vs. Angular

Angular has been rewritten almost entirely from the old Angular 1.x web framework. The stable version was released on September 14, 2016.

Angular 1.x still exists as an independent project under the name AngularJS, but has been largely surpassed by the more modern Angular

Versions

Version 2, known simply as Angular, is the one that led to the radical change of:

- Architecture (components vs. scope/controllers)
- Language (written entirely in TypeScript)
- Structure: classes, modules, etc (thanks to typescript)

Later versions, up to the current one, are backwards compatible with Angular 2

Angular vs. JQuery

The philosophy behind the two frameworks is completely different. JQuery provides tools for direct DOM manipulation.

Angular is more focused on a declarative description of the data that is associated with elements of the DOM through a process called **data binding**.

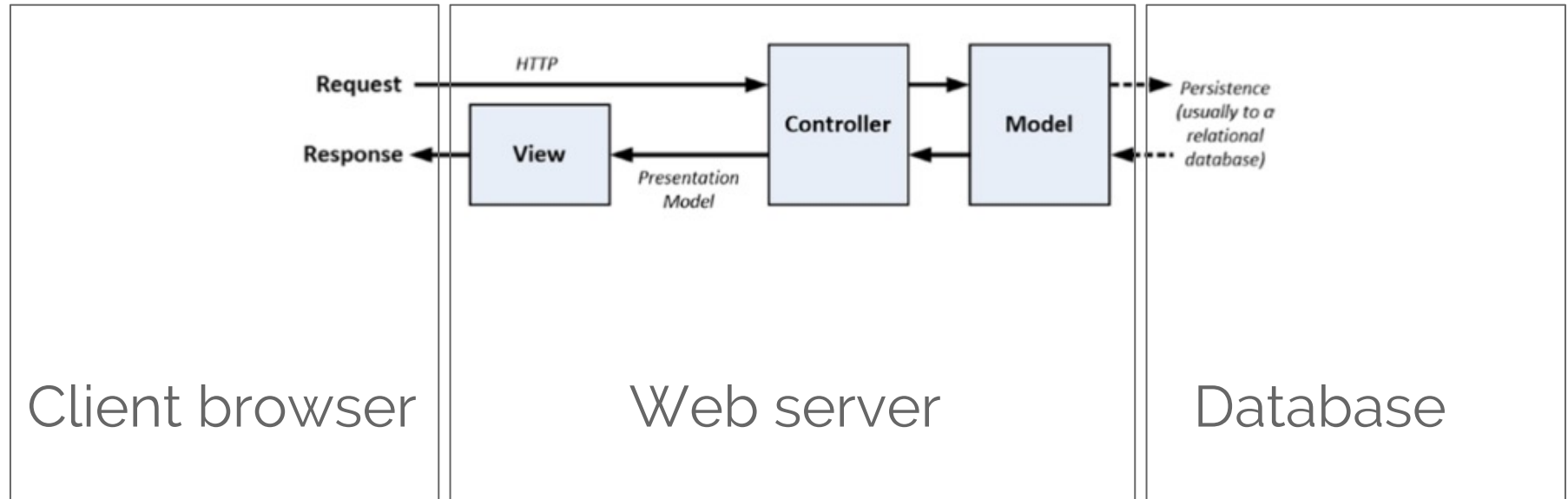
Angular is a complex framework but with an architecture designed for SPA development

MVC from server to client

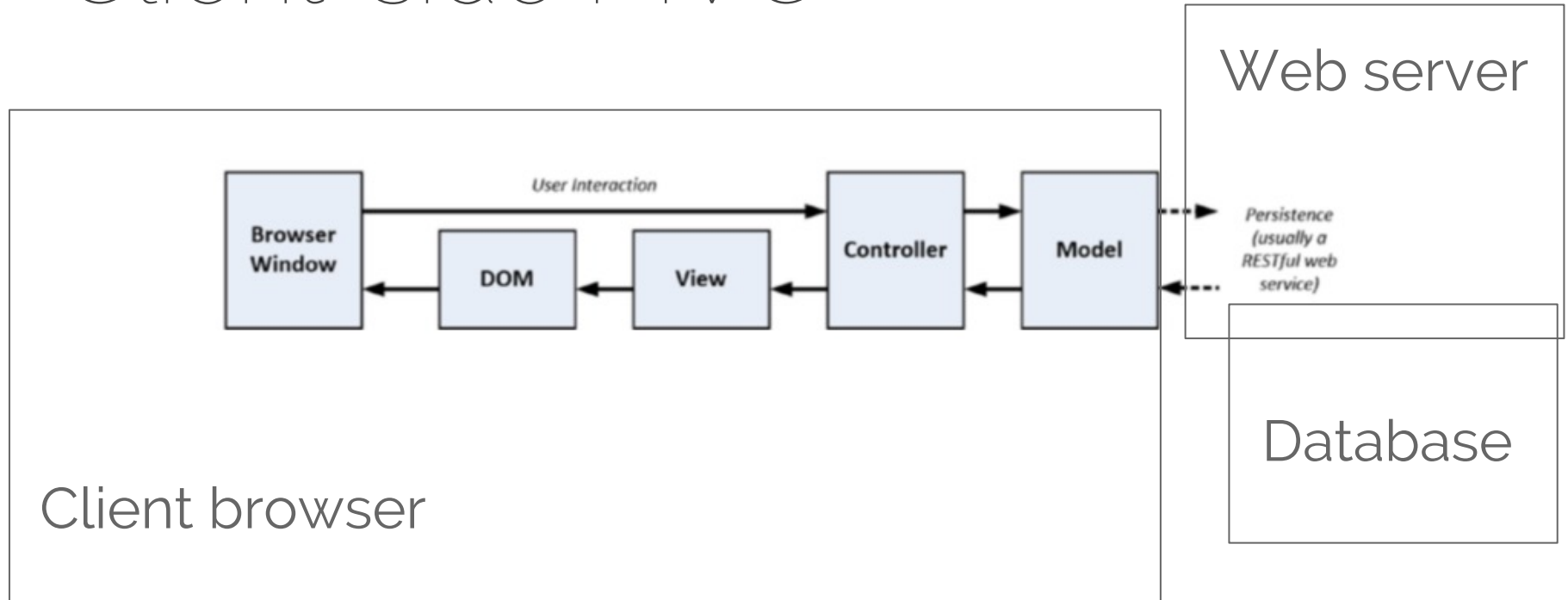
Model-View-Controller is a design pattern typically used for the development of web application.

It allows the "separation of concerns" in which the **data model** is separated from the **business** and **presentation** logic.

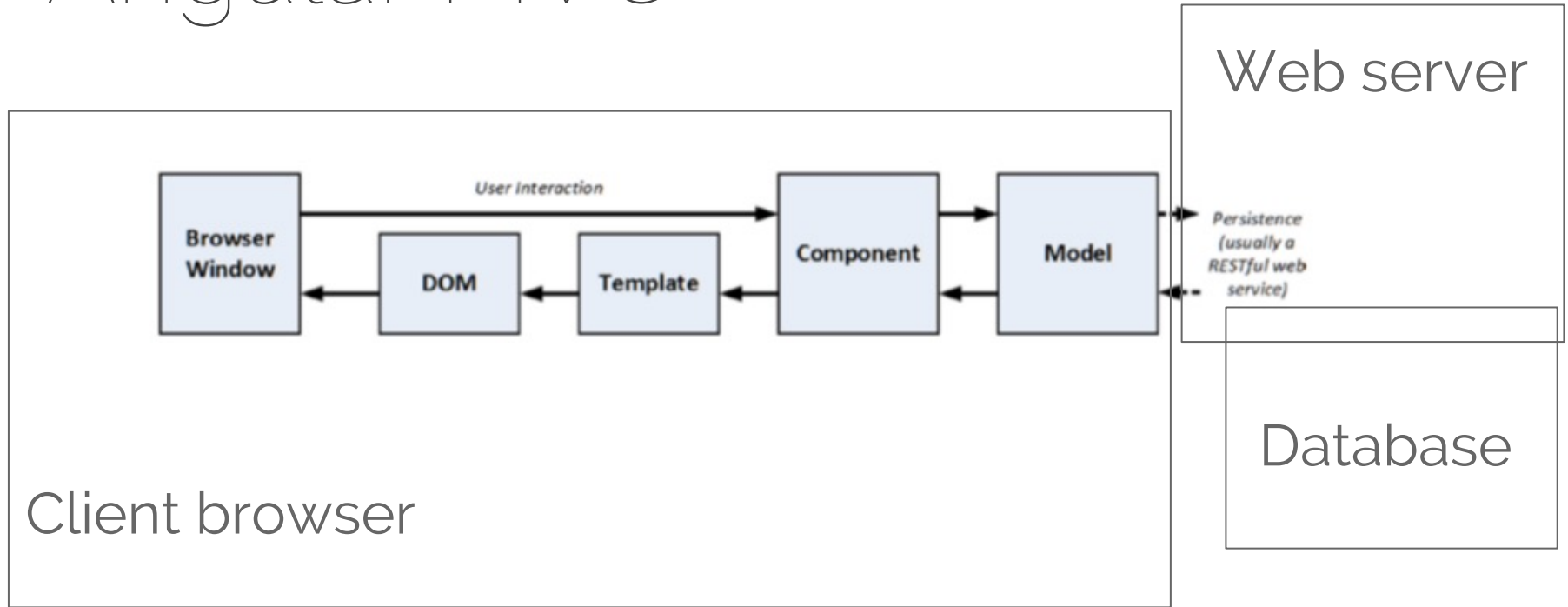
Server-side MVC



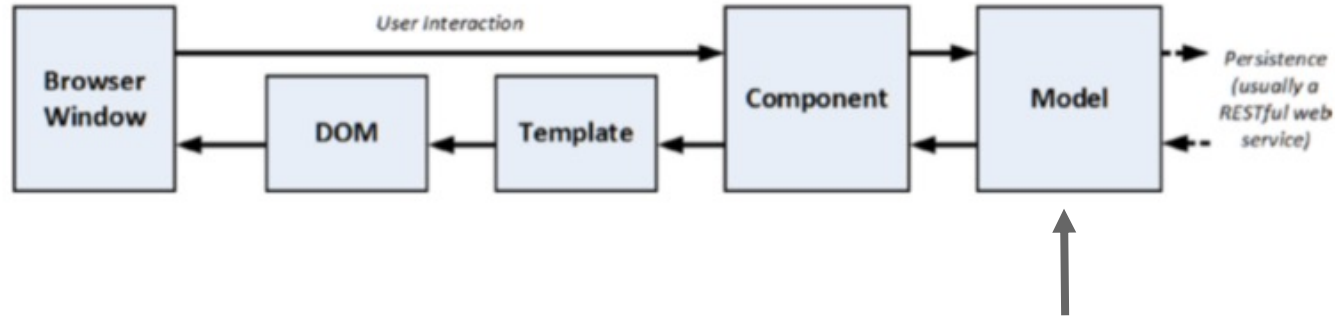
Client-side MVC



Angular MVC

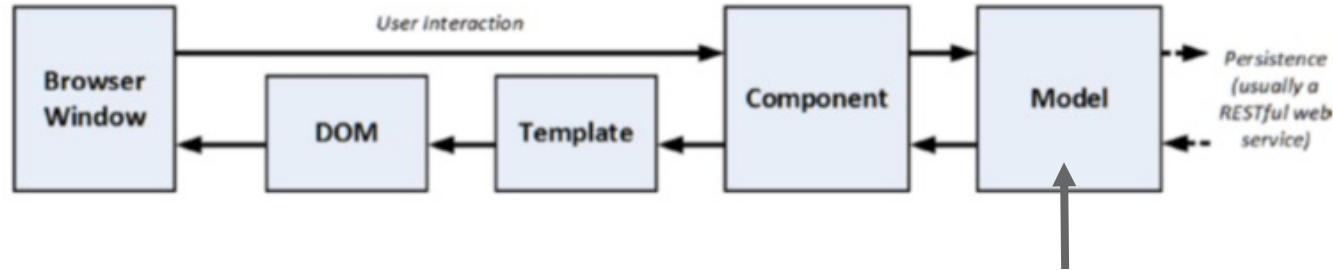


Model



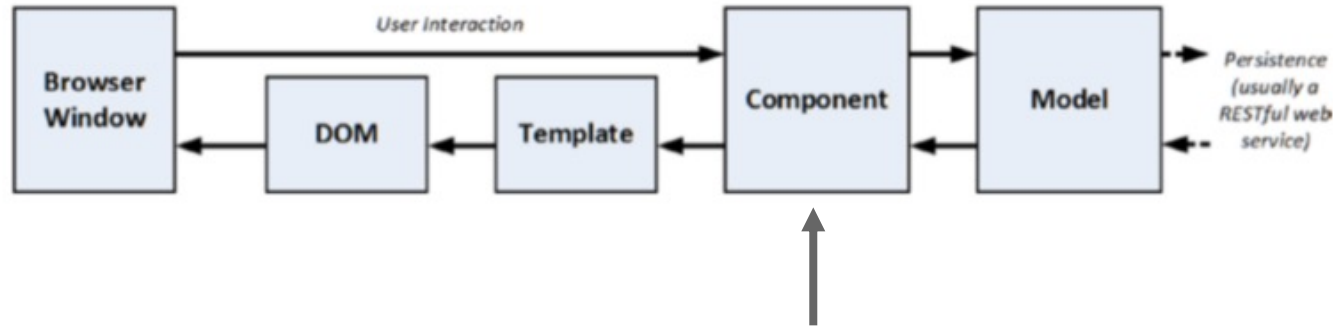
- It contains the data of a certain domain
- It contains the logic to create, modify and manage data
- It provides APIs that expose data and operations on it

Model



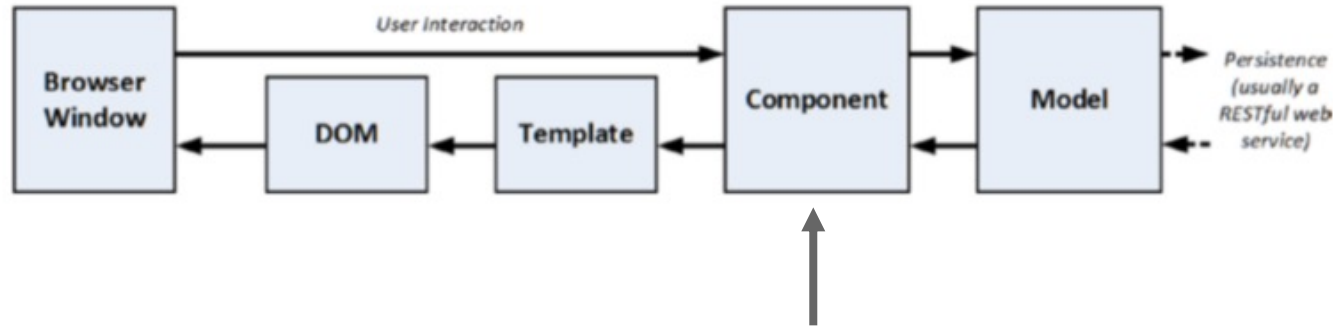
- It must NOT expose details on how the data is obtained or managed (the web service should not be exposed to the controller and views)
- It must NOT contain the logic that transforms the data with respect to interactions with the user (the controller's task)
- It must NOT contain the data visualization logic (task of the view)

Component (controller)



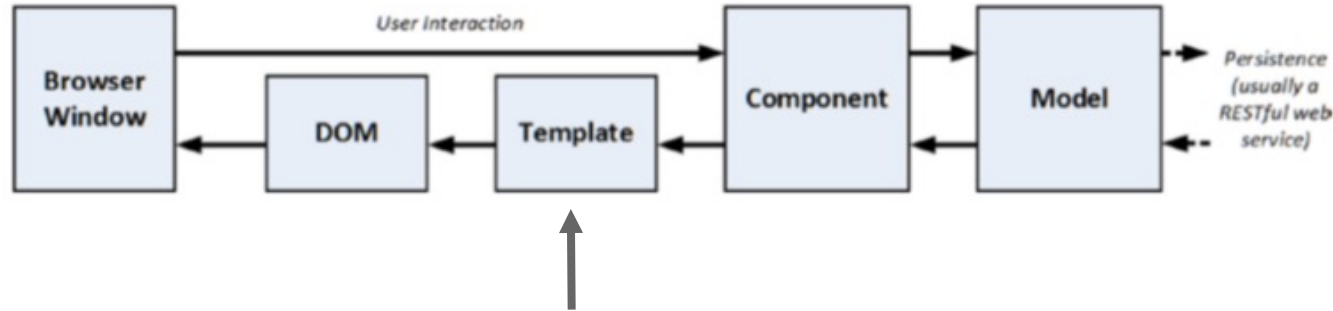
- It contains the logic that allows you to modify the model based on the interaction with the user
- It contains the logic to set the initial state of the view (template)
- It contains the features required by the template to access the data

Component (controller)



- It must NOT manage the display of data (it must not modify the DOM)
- It must NOT manage the data persistence logic (task of the model through the web service)

Template (view)



- It contains the markup logic necessary to present the data to the user
- It must NOT contain any data modification, creation and management logic
- It must NOT contain the application logic

Decorators

Angular uses decorators to "annotate" a class, method or property and add additional metadata and features at run-time.

For example a class decorator is applied to the class constructor and can be used to observe, modify or replace the class definition.

A decorator can in turn depend on parameters evaluated at runtime.

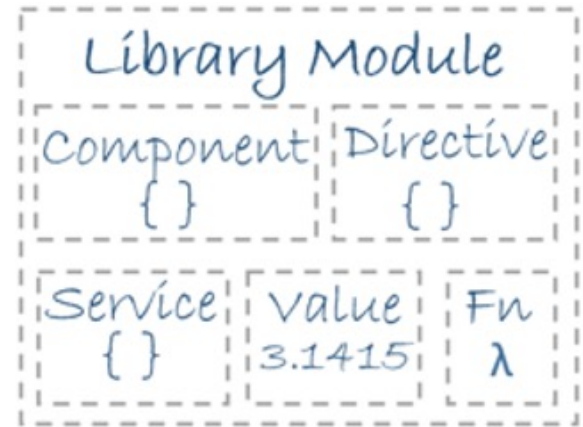
Decorators

```
function sealed(constructor: Function) {  
    Object.seal(constructor);  
    Object.seal(constructor.prototype);  
}
```

```
@sealed  
class Greeter {  
    greeting: string;  
    constructor(message: string) {  
        this.greeting = message;  
    }  
    greet() {  
        return "Hello, " + this.greeting;  
    }  
}
```

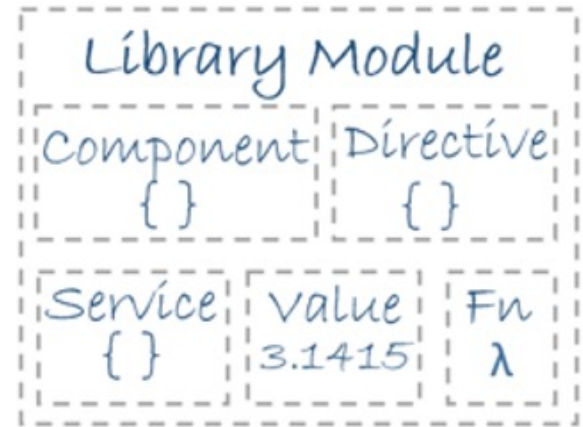

Architecture: Modules

The building blocks of an Angular app are called **NgModules** which act as containers for components, service providers, and all the code under a certain domain, workflow or closely related features



Architecture: Modules

Each Angular app has at least one **NgModule** called "root module" (by convention it is called **AppModule**) which defines how to "bootstrap" the application (which "root" components to insert within the page)



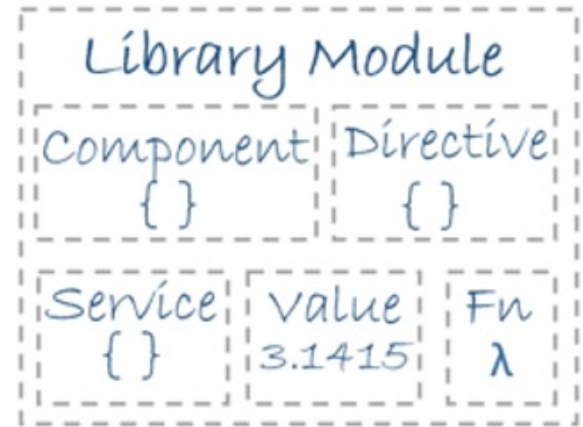
Architecture: Modules

- A module can import and export functionality to / from other modules
- Angular's built-in libraries are contained in **NgModules**:

Example:

HttpModule from *@angular/http*

FormsModule from *@angular/core*



Modules

A module in Angular is defined through a class decorated with the `@NgModule` decorator. The decorator takes as a parameter an object that describes the module's metadata:

- **Declarations:** components, services, etc contained in the module
- **Exports:** components to export
- **Imports:** modules whose exported classes are needed by the module components
- **Providers:** service providers exported by the module

Modules

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

```
@NgModule({  
  imports:      [ BrowserModule ],  
  providers:    [ Logger ],  
  declarations: [ AppComponent ],  
  exports:      [ AppComponent ],  
  bootstrap:    [ AppComponent ]  
})  
export class AppModule { }
```

Angular modules and JavaScript

In Angular a module is a class annotated with the decorator `@NgModule`.

However, the browser must know where the code contained in the imported modules is.

To do this, TypeScript modules are used. They describe the association between the exported objects and the files in which the code is defined

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

Modules

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

```
@NgModule({  
  imports: [ BrowserModule ],  
  providers: [ Logger ],  
  declarations: [ AppComponent ],  
  exports: [ AppComponent ],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule { }
```



TypeScript modules

Modules

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

```
@NgModule({  
  imports:      [ BrowserModule ],  
  providers:    [ Logger ],  
  declarations: [ AppComponent ],  
  exports:      [ AppComponent ],  
  bootstrap:    [ AppComponent ]  
})
```

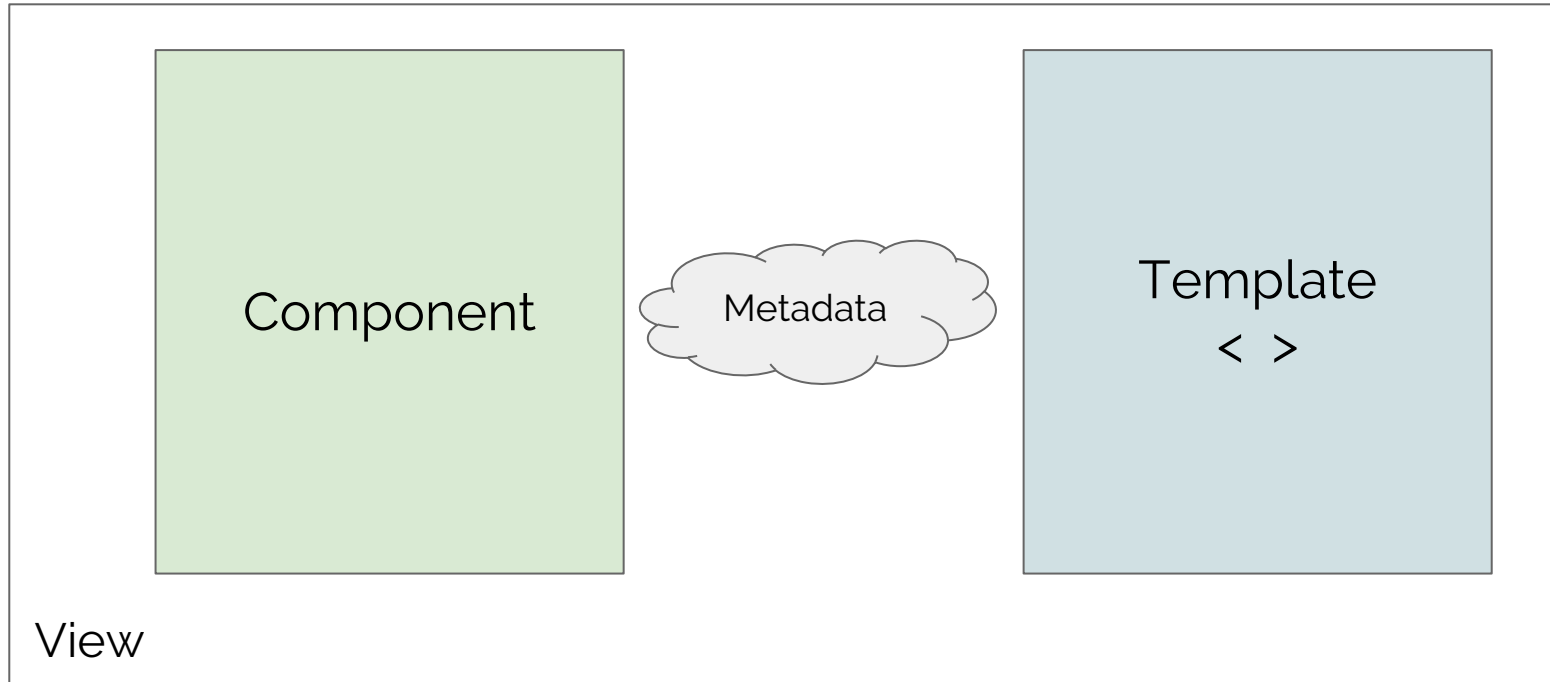
```
export class AppModule { } ←———— Angular module
```


Architecture: Components

Components allow the user to view the application data by controlling a portion of the visible screen (DOM) called "view".

A **component** consists of a **class** that contains the data and logic associated with it. Through the metadata defined by the `@Component` decorator an HTML **template** is associated, which defines the view to be displayed

Architecture: Components



Components

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

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

app.component.ts

```
<!doctype html>
<html lang="en">
<head><meta charset="utf-8"><title>Testng1</title>
  <base href="/">
<meta name="viewport" content="width=device-width, initial-
scale=1">
<link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

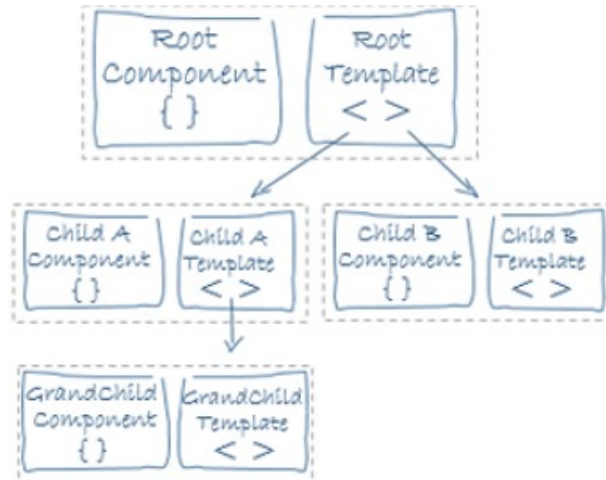
index.html

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

app.component.html

Hierarchy between views

Views can be structured in a hierarchy (a view can include other views). A child-view can belong to the same module or to another NgModule

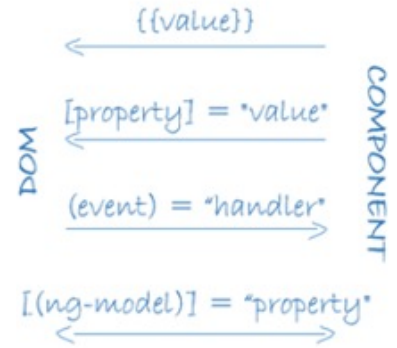


Data binding

How to synchronize the data in the component hierarchy with the template DOM?

Through the binding markup mechanism:

- **Interpolation:** Inserts the string evaluated in an *expression* into the HTML code
- **Property binding:** Sets the property of a view element (DOM or Component) based on an expression
- **Event binding:** Allows a component to respond to a DOM event



Data binding

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

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

```
export class AppComponent {  
  title = 'app';  
  isDisabled = true;  
  
  disableToggle() {  
    this.isDisabled = !this.isDisabled;  
  }  
}
```

```
<div style="text-align:center">
```

```
<h1>
```

```
  Welcome to {{ title }}
```

```
</h1>
```

```
  <button [disabled]="isDisabled" >Cancel is {{  
isDisabled ? "disabled" : "enabled" }}
```

```
</button>
```

```
  <button (click)="disableToggle()" >  
    {{isDisabled ? "enable" : "disable"}} the other  
  button</button>
```

```
</div>
```

Data binding

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

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

```
export class AppComponent {
```

```
  title = 'app';  
  isDisabled = true;
```

```
  disableToggle() {  
    this.isDisabled = !this.isDisabled;  
  }  
}
```

Interpolation



```
<div style="text-align:center">
```

```
<h1>
```

```
  Welcome to {{ title }}
```

```
</h1>
```

```
<button [disabled]="isDisabled" >Cancel is {{  
isDisabled ? "disabled" : "enabled" }}  
</button>
```

```
<button (click)="disableToggle()" >  
  {{isDisabled ? "enable" : "disable"}} the other  
button</button>
```

```
</div>
```

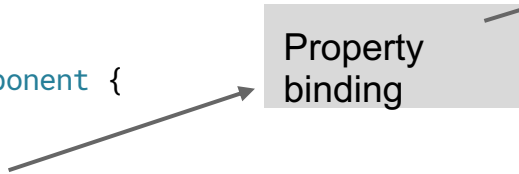
Data binding

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

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

```
export class AppComponent {  
  title = 'app';  
  isDisabled = true;  
  
  disableToggle() {  
    this.isDisabled = !this.isDisabled;  
  }  
}
```

Property
binding



```
<div style="text-align:center">
```

```
<h1>
```

```
  Welcome to {{ title }}
```

```
</h1>
```

```
<button [disabled]="isDisabled" >Cancel is {{  
isDisabled ? "disabled" : "enabled" }}
```

```
</button>
```

```
<button (click)="disableToggle()" >  
{{isDisabled ? "enable" : "disable"}} the other  
button</button>
```

```
</div>
```

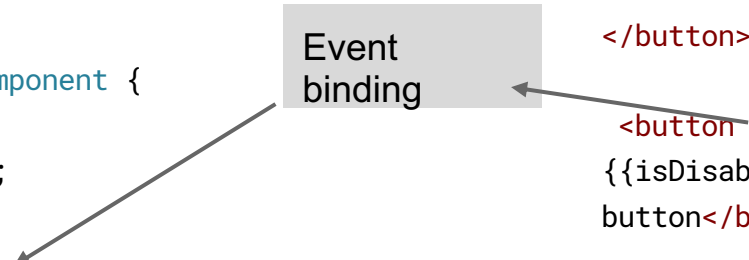

Data binding

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

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

```
export class AppComponent {  
  title = 'app';  
  isDisabled = true;  
  
  disableToggle() {  
    this.isDisabled = !this.isDisabled;  
  }  
}
```

Event
binding



```
<div style="text-align:center">
```

```
<h1>
```

```
  Welcome to {{ title }}
```

```
</h1>
```

```
  <button [disabled]="isDisabled" >Cancel is {{  
isDisabled ? "disabled" : "enabled" }}
```

```
</button>
```

```
  <button (click)="disableToggle()" >  
    {{isDisabled ? "Enable" : "Disable"}} the other  
  button</button>
```

```
</div>
```

Restrictions

1. The expressions used in interpolation and property binding must be **idempotent**: they must be able to be evaluated several times without changing the state of the application

Ex: `{{ counter = counter + 1 }}` is not idempotent and therefore generates an error

Restrictions

2. It is not possible to access objects that are outside the context defined by the template component. Each method or property is only the one defined within the component

Ex: `{{ Math.floor(0.1) }}` it is not a valid expression because `Math` is part of the global object and in general it is not a component property

Property binding or interpolation?

When the mapped property is a string it is possible to alternatively use interpolation or property binding (the result is the same)

`<p>` is the *interpolated* image.
`<p>` is the *property bound* image.

Warning: the binding refers to a property of the DOM and not to an HTML attribute (although often the mapping is 1:1). We can for example write:

```
<span [innerHTML]="title"></span>
```

Pipes

They are special classes that allow you to transform input data into a desired output. They can be used in data interpolation with the following syntax:

```
{{ expression | pipe:parameter }}
```

There are multiple built-in pipes such as **date**, **lowercase**, **uppercase**, **async**, etc.

They can be personalized by implementing the **PipeTransform** class annotated with the **@Pipe** decorator

Pipes

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

@Component({
  selector: 'app-hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})

export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15);
}

// Prints April 15, 1988
// Instead of Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time)
```

Pipes

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 *   formats to: 1024
 */
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements
PipeTransform {
  transform(value: number, exponent: string): number
  {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

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

@Component({
  selector: 'app-power-booster',
  template: `
    <h2>Power Booster</h2>
    <p>Super power boost: {{2 |
exponentialStrength: 10}}</p>
`
})
export class PowerBoosterComponent { }
```

Built-in directives

Directives are special constructs applicable to HTML elements of templates that allow you to:

- Include content selectively
- Select from content snippets
- Repeat a certain fragment for each element of an array
- etc...

*ngIf

The directive ***ngIf** includes an element and its content only if the expression is evaluated as true

```
<div *ngIf="expr"></div>
```

[ngSwitch] *ngSwitchCase

This directive allows you to choose different fragments to include in the HTML document based on the value of the expression

```
<div [ngSwitch]="num_products()">  
  <span *ngSwitchCase="1">Only one element</span>  
  <span *ngSwitchCase="2">Only two elements</span>  
  <span *ngSwitchDefault>More than 2 elements</span>  
</div>
```

*ngFor

This directive allows you to repeat a certain fragment several times, iterating over a certain collection

```
<ul>  
  <li *ngFor="let product of get_products()">  
    {{ product }}  
  </li>  
</ul>
```

Template reference variable

It is possible to create a variable as a reference to an element of the DOM within a certain template




```
<input #phone placeholder="phone number">

<!-- lots of other elements -->

<!-- phone refers to the input element;
pass its `value` to an event handler -->
<button (click)="callPhone(phone.value)">Call</button>
```

Template reference variable

Special directives can change the type of the object pointed to by the reference variable



```
<input type="text" #username="ngModel" required name="username" [(ngModel)]="user.username" class="form-control" id="inputUsername" placeholder="Enter username">
```

```
<div [hidden]="username.valid || username.pristine"
  class="alert alert-danger">
  Username is required
</div>
```



Valid and pristine are properties of ngModel and not of HTMLFormElement.

First example application

npm install -g @angular/cli

Angular-cli is a utility that allows in a simple way:

- To generate the structure of a new app
- To generate and add components, services and modules
- To launch the application from a local webserver, dynamically updating the code when the source files are modified

Create a new app

```
$ ng new <app_name>
```

```
$ ls <app_name>
```

README.md	karma.conf.js	package-
lock.json	protractor.conf.js	tsconfig.json
e2e	node_modules	package.json
src	tslint.json	

Add a component

\$ ng generate component comp-name

```
create src/app/comp-name/comp-name.component.css (0 bytes)
create src/app/comp-name/comp-name.component.html (28 bytes)
create src/app/comp-name/comp-name.component.spec.ts (643 bytes)
create src/app/comp-name/comp-name.component.ts (280 bytes)
update src/app/app.module.ts (486 bytes)
```

Local webserver

\$ ng serve --open

```
** NG Live Development Server is listening on localhost:4200, open your browser on
http://localhost:4200/ **
 15% building modules 45/48 modules 3 active ...form-browser/esm5/platform-browser.jswebpack:
wait until bundle finished: /
Date: 2018-03-30T20:12:14.820Z
Hash: f8743c5298a26dc080f5
Time: 6669ms
chunk {inline} inline.bundle.js (inline) 3.85 kB [entry] [rendered]
chunk {main} main.bundle.js (main) 32 kB [initial] [rendered]
chunk {polyfills} polyfills.bundle.js (polyfills) 549 kB [initial] [rendered]
chunk {styles} styles.bundle.js (styles) 41.5 kB [initial] [rendered]
chunk {vendor} vendor.bundle.js (vendor) 7.42 MB [initial] [rendered]

webpack: Compiled successfully.
```

Advanced concepts

Dependency Injection

DI is a very important pattern, used in many contexts in the Angular framework to manage dependencies between components

Goal: to make a class get the dependencies it needs from an external source instead of instantiating them by itself

Example

```
export class Car {  
  
  public engine: Engine;  
  public tires: Tires;  
  
  constructor() {  
    this.engine = new Engine();  
    this.tires = new Tires();  
  }  
  
  // Method using the engine and tires  
  drive() {  
    return `${this.description} car with  
    +      `${this.engine.cylinders} cylinders  
    and ${this.tires.make} tires.`;  
  }  
}
```

The **Car** class needs **Engine** and **Tires** to work

- **Car** instantiates an **Engine** object and a **Tires** object in its constructor
- The **Car** class is “fragile” because it strongly depends not only on the dependency interface but also on their implementation

Problems

```
export class Car {  
  
  public engine: Engine;  
  public tires: Tires;  
  
  constructor() {  
    this.engine = new Engine();  
    this.tires = new Tires();  
  }  
  
  // Method using the engine and tires  
  drive() {  
    return `${this.description} car with ` +  
      `${this.engine.cylinders} cylinders and  
      `${this.tires.make} tires.`;  
  }  
}
```

- What happens if the **Engine** class is modified so that the constructor requires parameters?
- What happens if a subclass of **Tires** is created in the future and we would like to use it in our **Car** class?
- What happens if we want to add to the class **Car** another class (service) shared with other instances of **Car**?

Problems

```
export class Car {  
  
  public engine: Engine;  
  public tires: Tires;  
  
  constructor() {  
    this.engine = new Engine();  
    this.tires = new Tires();  
  }  
  
  // Method using the engine and tires  
  drive() {  
    return `${this.description} car with ` +  
      `${this.engine.cylinders} cylinders and  
      `${this.tires.make} tires.`;  
  }  
}
```

- Testing the class **Car** is complex because we need to be able to instantiate all of its dependencies
 - The dependencies might be incompatible with our test environment (what if **Engine** makes an asynchronous call to a server?)

Solution

```
export class Car {  
  
  constructor(public engine: Engine,  
    public tires: Tires) {}  
  
  // Method using the engine and tires  
  drive() {  
    return `${this.description} car with  
    +  
    `${this.engine.cylinders} cylinders  
    and ${this.tires.make} tires.`;  
  }  
}
```

In the DI pattern the dependency definition is specified in the constructor.

- The **Car** class expects to receive the dependencies it needs from the outside ready for use
- The **Car** class no longer automatically instantiates its dependencies, since they are "injected" from the outside

D|

```
// Simple car with 4 cylinders let car  
= new Car(new Engine(), new Tires());
```

```
class Engine2 {  
  constructor(public cylinders:  
number) {}  
}
```

```
// Super car with 12 cylinders  
let bigCylinders = 12;  
let car = new Car(new  
Engine2(bigCylinders), new Tires());
```

- The Car class is now dependent only on the interface of the objects it uses
- The problem of instantiating objects has been moved **from the producer to the consumer** of the Car object

Injector

To ensure that the consumer is not forced to manually manage the creation of dependencies, a DI framework usually provides a component called an **injector** that:

- manages a list of "injectable" objects as dependencies
- knows how to build each object
- is able to instantiate each injectable object on-demand

Services and Components

In Angular, a service (Service) is a class that can be injected into a Component to provide specific features usually **independent from the visualization logic**.

Ex:

- A component shows the list of users
- A service allows you to retrieve the list from a RESTful webservice

Services and Components

A component usually does not implement features beyond view management such as:

- Managing the retrieval and sending of data to a server
- Validating user input
- Logging on the console
- etc.

Instead, it uses services that can be reused across multiple components

Services and Components

A service is any class decorated with the **@Injectable** decorator.

- A component can use a service by defining it as a dependency within its constructor
- During the bootstrap process, Angular creates an Injector to inject services into components

Injector

The Angular injector keeps in memory all the dependencies already instantiated in order to **reuse** them if several components need the same service (the injector is in fact a dependency factory)

How does the injector know how to create dependencies?

- Using a provider

Provider

For each dependency required within an application, a provider must be registered within the injector so that it can create new instances of the dependency every time they are required by a component.

For services, the provider is simply the class itself. It is possible to create custom providers for special cases

Example

Let's create a service:

\$ ng generate service logger

```
create src/app/logger.service.spec.ts (374 bytes)  
create src/app/logger.service.ts (112 bytes)
```


Example

```
//logger.service.ts
import { Injectable } from '@angular/core';

@Injectable()
export class LoggerService {

  constructor() { }

  public log( message: string ) {
    console.log( `${new Date()}: ${message}` );
  }

}
```

The service is simply a class with the **@Injectable** annotation.

A service may in turn depend on other services

Example

```
import { Component } from '@angular/core';
import { LoggerService } from './logger.service';

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

  products: {id: number, name: string}[] = [];
  private idgen = new UniqueId();

  constructor( private log: LoggerService ) {}

  add_product( p: string ) {
    this.log.log('Adding new product');
    this.products.push( {id: this.idgen.get(), name: p } );
  }

  delete_product( id: number ) {
    this.log.log( 'Deleting product with id ' + id );
    /*....*/ }
}
```

The component that wants to use the service inserts its declaration in the signature of the constructor.

It is essential to note the parameter with the correct type which will be the key used by the injector to understand which instance to inject

Example

```
import { Component } from '@angular/core';
import { LoggerService } from './logger.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [
    {provide: LoggerService, useClass: LoggerService }
  ],
})
export class AppComponent {

  products: {id: number, name: string}[] = [];
  private idgen = new UniqueId();
  constructor( private log: LoggerService ) {}
  add_product( p: string ) {
    this.log.log('Adding new product');
    this.products.push( {id: this.idgen.get(), name: p } );
  }
  delete_product( id: number ) {
    this.log.log( 'Deleting product with id ' + id );
    /*....*/ }
}
```

To make the service available only to the component, we can register the provider in the list of providers used by the component.

These can be defined in the metadata of the component itself

Example

```
import { Component } from '@angular/core';
import { LoggerService } from './logger.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [
    { provide: LoggerService, useClass: LoggerService }
  ],
})
export class AppComponent {

  products: {id: number, name: string}[] = [];
  private idgen = new UniqueId();
  constructor( private log: LoggerService ) {}
  add_product( p: string ) {
    this.log.log('Adding new product');
    this.products.push( {id: this.idgen.get(), name: p } );
  }
  delete_product( id: number ) {
    this.log.log( 'Deleting product with id ' + id );
    /*....*/ }
}
```

Key

Class to instantiate. It can be different from the key as long as it implements the same interface

Example

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

```
import { AppComponent } from './app.component';  
import { LoggerService } from './logger.service';
```

```
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule  
  ],  
  providers: [  
    {provide: LoggerService, useClass: LoggerService }  
  ],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```



To make the service available throughout the application, the same provider can be registered directly in the definition of the root module

In this way, **only one instance** of LoggerService will be used for all components of the module

Hierarchical DI

There are actually multiple injectors, one for each component present in the application.

The injectors hierarchy is identical to the component hierarchy

The dependencies are injected respecting the hierarchical order among the components

Injector bubbling

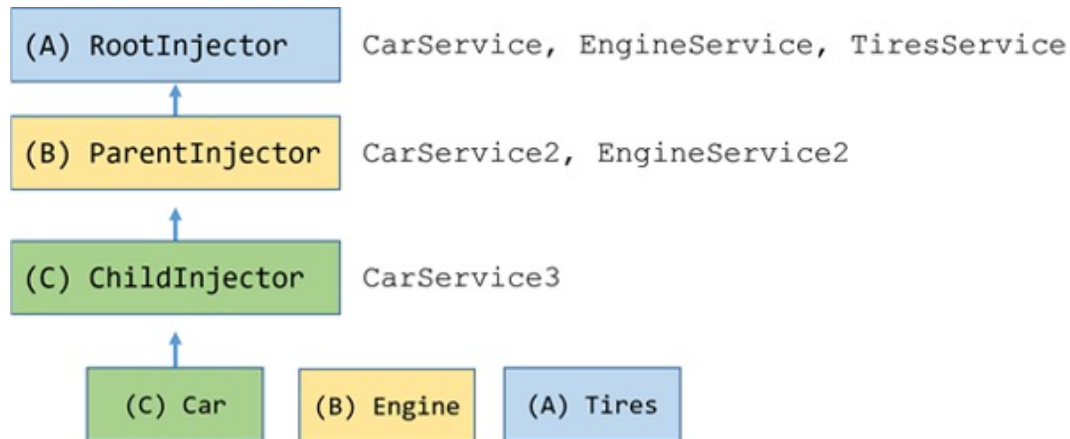
When a component requires a dependency, Angular tries to solve it using the provider registered on the component's injector.

If the component injector does not contain the provider, the request is delegated to the parent component injector, going up the hierarchy from the bottom up.

If even the module injector does not contain the right provider, an error is generated.

Specialized providers

An advantage of hierarchical DI is that we can provide services with varying degrees of specialization at different points in the hierarchy



Observables

Another very important pattern used by Angular is called "Observer" and is used to pass messages between two entities, called respectively **publisher** and **subscriber**.

They are essential to solve the problem of asynchronous data management **in a declarative way**

Observables

An Observable defines a **function** that can publish values to a certain **observer**.

Unlike the promises, the function is not executed until the entity that wants to consume those values "subscribes" to that particular Observable.

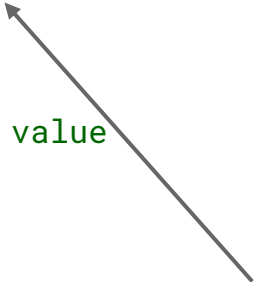
The consumer receives notifications about published values until the Observer function returns or the consumer explicitly unsubscribes

```
const sequenceObservable = new Observable( (observer) => {
  const seq = [1, 2, 3];
  let timeoutId;

  // Will run through an array of numbers, emitting one value
  // per second until it gets to the end of the array.
  function doSequence(arr, idx) {
    timeoutId = setTimeout(() => {
      observer.next(arr[idx]);
      if (idx === arr.length - 1) {
        observer.complete();
      } else {
        doSequence(arr, ++idx);
      }
    }, 1000);
  }

  doSequence(seq, 0);

  // Unsubscribe should clear the timeout to stop execution
  return {unsubscribe() {
    clearTimeout(timeoutId);
  }};
});
```



The Observable constructor takes as input the function that publishes values to a certain observer

```
const sequenceObservable = new Observable( (observer) => {  
  const seq = [1, 2, 3];  
  let timeoutId;
```

```
  // Will run through an array of numbers, emitting one value  
  // per second until it gets to the end of the array.
```

```
  function doSequence(arr, idx) {  
    timeoutId = setTimeout(() => {  
      observer.next(arr[idx]);  
      if (idx === arr.length - 1) {  
        observer.complete();  
      } else {  
        doSequence(arr, ++idx);  
      }  
    }, 1000);  
  }  
}
```

```
doSequence(seq, 0);
```

```
  // Unsubscribe should clear the timeout to stop execution  
  return {unsubscribe() {  
    clearTimeout(timeoutId);  
  }};  
});
```



Observer is an object that contains 3 functions:

- **next(v)** is the handler that manages each value v that is generated
- **error(e)** is the handler that manages any errors
- **complete()** *[optional]* is the handler that is invoked when there are no more values to publish

Subscribers

Creating an Observable does not involve the execution of the function. To start receiving the values it is necessary to invoke the subscribe method passing 3 functions as parameters: **next**, **error** e **complete**:

```
sequenceObservable.subscribe(  
  (x) => { console.log("New value generated: "+x); },  
  (err) => { console.log("Error: " + err );},  
  () => { console.log("No more values");}  
);
```

Observables vs. promises

- Observables are **declarative**, they allow you to create "recipes" that can be executed when a component invokes subscribe. Instead, promises are executed as soon as they are created.
- Observables publish a (possibly infinite) sequence of values. Promises provide only one value

Observables vs. promises

- Observers can create transformation chains on values, which are executed only when a consumer invokes subscribe:
 - `observable.map((v)=>2*v)`
- An observable can be deleted while it is running by invoking the `unsubscribe()` method

RxJS

Reactive Extensions for JavaScript is a library used by Angular that uses observables to make it easier to develop callback-based asynchronous code.

It provides the implementation of the Observable class and utilities to create Observables from other types (such as arrays or promises) and to modify the stream of produced values

RxJS

Ex.1 Create an observable from a sequence of values:

```
const nums = Observable.of(1, 2, 3);
```

Ex.2 Create a new observable that changes the values produced by the previous observable on the fly:

```
const squareValues = nums.map((val: number) => val * val);  
squareValues.subscribe(x => console.log(x));
```

```
// Logs  
// 1 4 9
```

RxJS pipe

The pipe operator allows you to combine multiple functions into a single function, that executes them in sequence for each generated value

```
import { filter } from 'rxjs/operators/filter';
import { map } from 'rxjs/operators/map';

const squareOdd = Observable.of(1, 2, 3, 4, 5)
  .pipe(
    filter(i => i % 2 !== 0),
    map(n => n * n)
  );

// Subscribe to get values
squareOdd.subscribe(x => console.log(x)); // 4 16
```

RxJS operators

AREA	OPERATORS
Creation	<code>from</code> , <code>fromPromise</code> , <code>fromEvent</code> , <code>of</code>
Combination	<code>combineLatest</code> , <code>concat</code> , <code>merge</code> , <code>startWith</code> , <code>withLatestFrom</code> , <code>zip</code>
Filtering	<code>debounceTime</code> , <code>distinctUntilChanged</code> , <code>filter</code> , <code>take</code> , <code>takeUntil</code>
Transformation	<code>bufferTime</code> , <code>concatMap</code> , <code>map</code> , <code>mergeMap</code> , <code>scan</code> , <code>switchMap</code>
Utility	<code>tap</code>
Multicasting	<code>share</code>

EventEmitter

EventEmitter is a class provided by Angular that implements an Observer by adding the `emit(v)` function. When invoked, the observer publishes the value `v` to the consumer.

```
@Component({
  selector: 'zippy',
  template: `<div class="zippy">
    <div (click)="toggle()">Toggle</div>
    <div [hidden]="!visible">
      <ng-content></ng-content>
    </div> </div>`})
export class ZippyComponent {
  visible = true;
  public visibility_change = new EventEmitter<any>();

  toggle() {
    this.visible = !this.visible; if (this.visible) { this.visibility_change.emit(true); } else {
this.visibility_change.emit(false); }
  }
}
```

HttpClient

It is one of Angular's most used classes.

It allows to make asynchronous HTTP calls and to return the values obtained in the form of observable.

```
Observable<Users[]> o = http.get<Users[]>('http://myserver.com/api/v1/users')
    .pipe(
        tap(users => this.log(`fetched users: ${JSON.stringify(users)} `)),
        catchError(this.handleError('getHeroes', []))
    );
```

```
o.subscribe( (users)=>{ this.users = users } );
```

HttpClient

The **get** and **delete** methods expect data formatted as a JSON string by default.

It is possible to specify additional headers, request parameters, etc.

Note: The HTTP request returns an observable. For this reason, the request is made only when the `subscribe()` method is invoked

HttpClient

Use:

1. Inject the HttpClient service into your component or service

```
constructor( private http: HttpClient ) {}
```


1. Make the request

```
http.get( <url>:string, <options>:{} ) : Observable< any >
```

HttpClient get options

```
options: {  
  headers?: HttpHeaders | {  
    [header: string]: string | string[];  
  };  
  observe?: HttpObserve;  
  params?: HttpParams | {  
    [param: string]: string | string[];  
  };  
  reportProgress?: boolean;  
  responseType?: 'arraybuffer' | 'blob' | 'json' |  
  'text';  
  withCredentials?: boolean;  
}
```

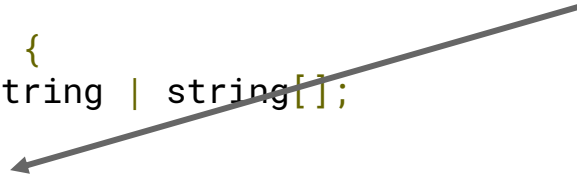
Any additional
headers such as
key:value pairs



HttpClient get options

```
options: {  
  headers?: HttpHeaders | {  
    [header: string]: string | string[];  
  };  
  observe?: HttpObserve;  
  params?: HttpParams | {  
    [param: string]: string | string[];  
  };  
  reportProgress?: boolean;  
  responseType?: 'arraybuffer' | 'blob' | 'json' |  
  'text';  
  withCredentials?: boolean;  
}
```

If set, the observable returns the complete response (with headers, return value, etc) and not just the data



HttpClient get options

```
options: {  
  headers?: HttpHeaders | {  
    [header: string]: string | string[];  
  };  
  observe?: HttpObserve;  
  params?: HttpParams | {  
    [param: string]: string | string[];  
  };  
  reportProgress?: boolean;  
  responseType?: 'arraybuffer' | 'blob' | 'json' |  
  'text';  
  withCredentials?: boolean;  
}
```

Request
parameters that are
automatically
serialized
according to the
MIME type
application/x-
www-form-
urlencoded

Sending data to the server

The `post()` and `put()` methods of `HttpClient` allow to send data to the server. The function supports the sending of generic data (objects) that are usually serialized as JSON strings

```
http.post( <url>:string, <body>: any, <options>:{} ) : Observable< any >
```

Routing

Angular provides a special component called Router that implements navigation between different views with a model similar to the one commonly used by the browser:

- The URL indicates the view displayed
- By clicking on a link, the view is replaced by the one pointed by the link
- The back and forward buttons allow you to navigate back and forth in the history of the views visited

Routing

The navigation model implemented by the router is similar to the one of the browser, but the type of web application is still **single page** type:

- Changing the current view does not cause an HTTP call
- Only a portion of the current page (a view) can be edited, not the whole page
- Routing events can be intercepted and trigger other actions

Set up routing

The first step is to specify which is the base URL from which the router can compose the URLs for all sub-views.

The base URL is set in the index.html file with the `<base>` tag:

```
<base href="/" />
```

Set up routing

A module is then generated and added to the project:

```
$ ng generate module app-routing --flat --module=app
```

Module's name

Do not generate a
dedicated directory for
the module

Automatically import the
new module into the "app"
module (usually the name
of the root module)

Set up routing

Routes are specified by instantiating a Routes object

<https://angular.io/api/router/Routes>

```
const routes: Routes = [  
  { path: '', redirectTo: '/login', pathMatch: 'full' },  
  { path: 'login', component: UserLoginComponent },  
  { path: 'messages', component: MessageListComponent }  
];
```

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


Set up routing

The `forRoot ()` method is invoked which returns a **module** with all the services and directives necessary for its use. The module is configured with the newly defined routes and exported to the other components of the application

```
const routes: Routes = [
  { path: '', redirectTo: '/login', pathMatch: 'full' }, { path: 'login', component:
  UserLoginComponent }, { path: 'messages', component: MessageListComponent }
];
@NgModule({
  imports: [ RouterModule.forRoot(routes) ],
  exports: [ RouterModule ]
})
export class AppRoutingModule { }
```

Set up routing

At this point it is possible to insert the `<router-outlet>` directive in the template of any component or directly in `index.html`. The element specifies in which place of the DOM the various views managed by the router should be inserted

```
<h1>{{title}}</h1>
```

```
<router-outlet></router-outlet>
```

```
<app-messages></app-messages>
```

Navigation

It is possible to change the view by inserting links within the HTML code, as you would do with normal pages:

```
<a routerLink="/signup">Sign-up new user</a>
```

or programmatically in the code of a component

```
constructor( private router: Router ) { }  
change_page( ) {  
    this.router.navigate(['/page']);  
}
```

To know more

Official documentation:

<https://angular.io/docs>

Framework API list

<https://angular.io/api>

Cheat-sheet

<https://angular.io/guide/cheatsheet>