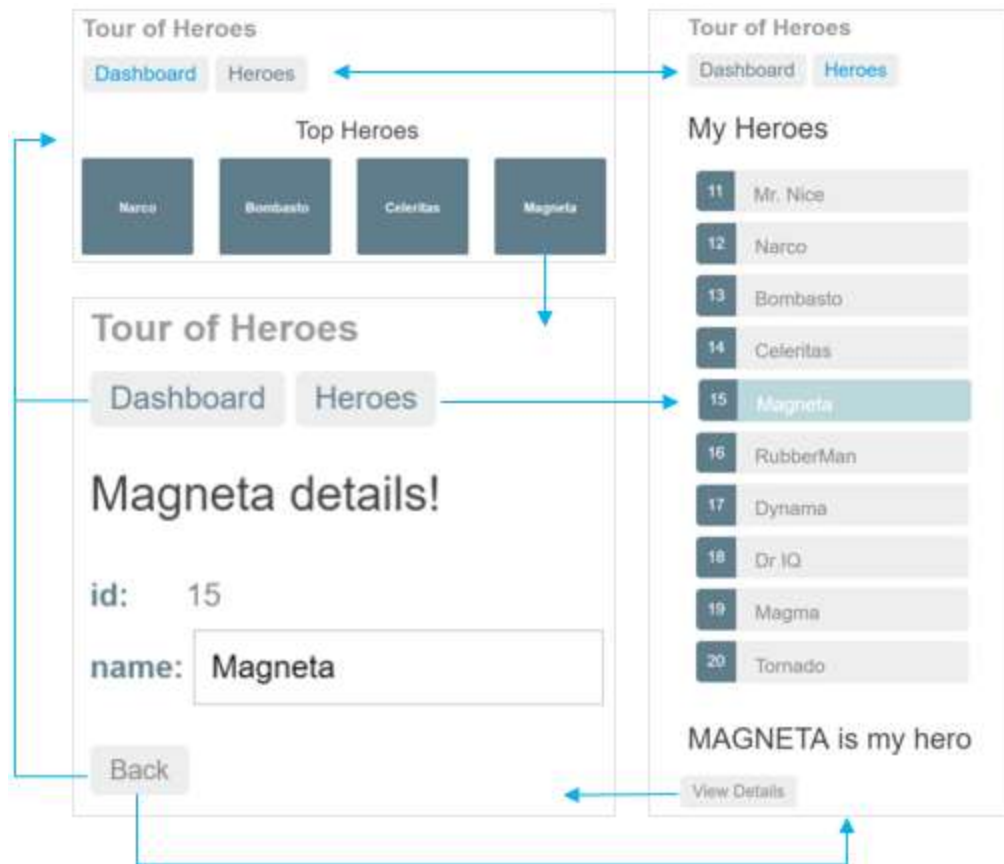


# Angular

Heroes Tutorial

# Heroes to-do

- Θα δημιουργήσουμε το παρακάτω web app:



## Tour of Heroes

[Dashboard](#)

[Heroes](#)

### Top Heroes

Narco

Bombasto

Celeritas

Magneta

# Χρήση

- Angular local setup

<https://angular.io/guide/setup-local>

[For Visual Studio Code:](#)

<https://medium.com/@Zeroesandones/how-to-create-a-shell-angular-9-project-5b5b254b3ac4>

- Online IDE <https://stackblitz.com/>  
(no local installation)

# Δημιουργία angular project

- Online
- <https://stackblitz.com/>
- src/app folder includes Angular app

# Index.html

```
<!doctype html>  
<html lang="en">  
<head>  
  <meta charset="utf-8">  
  <title>AngularTourOfHeroes</title>  
</head>  
<body>  
  <app-root></app-root>  
</body>
```

# App Component Structure

- You'll find the implementation of the shell distributed over three files:
  - `app.component.ts`
    - the component class code, written in TypeScript.
  - `app.component.html`
    - the component template, written in HTML.
  - `app.component.css`
    - the component's private CSS styles.

# Class file

- Open the component class file (app.component.ts) and change the value of the title property to 'Tour of Heroes'.

app.component.ts (class title property)

Προσέξτε ότι: selector: 'app-root'

βάλτε στη κλάση:

```
title = 'Tour of Heroes';
```



# Διπλά άγκιστρα

- app.component.html (template)  
`<h1>{{title}}</h1>`
- The double curly braces are Angular's *interpolation binding* syntax. This interpolation binding presents the component's title property value inside the HTML header tag.

# src/styles.css

```
h1 { color: #369; font-family: Arial, Helvetica,  
sans-serif; font-size: 250%; }
```

```
h2, h3 { color: #444; font-family: Arial, Helvetica,  
sans-serif; font-weight: lighter; }
```

```
body { margin: 2em; }
```

```
body, input[type="text"], button { color: #333;  
font-family: Cambria, Georgia; }
```

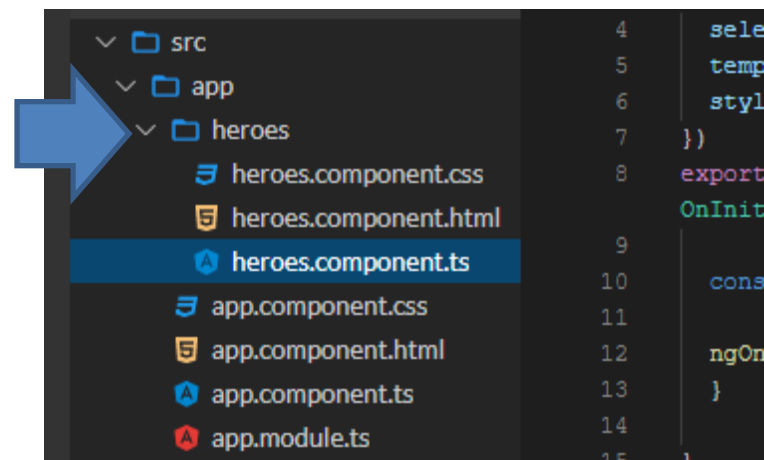
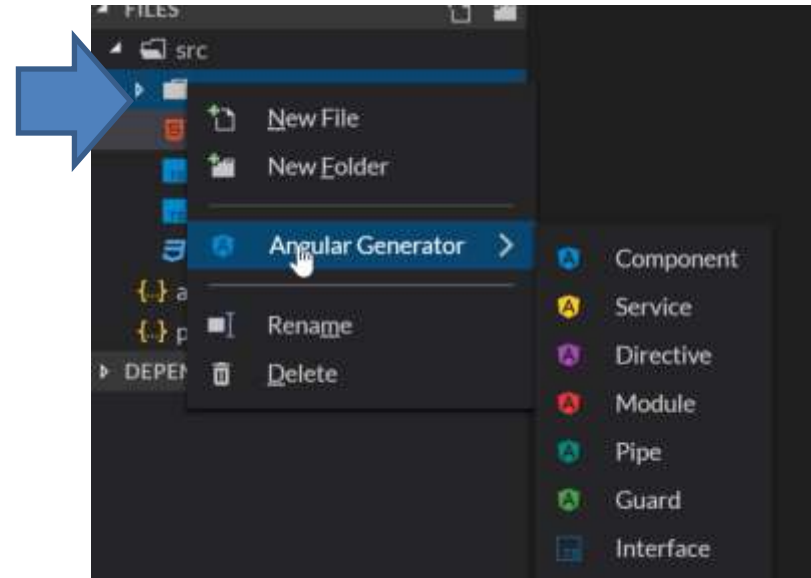
```
* { font-family: Arial, Helvetica, sans-serif; }
```

# Ως εδώ..

- You learned that Angular components display data.
- You used the double curly braces of interpolation to display the app title.

# ng generate component heroes

- Δεξί κλικ δημιουργία component
- The CLI creates a new folder, **src/app/heroes/**,
- 3 files  
HeroesComponent



# Component

Always import the [Component](#) symbol from the Angular core library and annotate the component class with `@Component`.

- `@Component` is a decorator function that specifies the Angular metadata for the component.
- The CLI generated three metadata properties:
- **selector**— the component's CSS element selector
- **templateUrl**— the location of the component's template file.
- **styleUrls**— the location of the component's private CSS styles.

```
heroes.component.ts x
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4      selector: 'app-heroes',
5      templateUrl: './heroes.component.html',
6      styleUrls: ['./heroes.component.css']
7  })
8  export class HeroesComponent implements OnInit {
9
10     constructor() { }
11
12     ngOnInit() {
13     }
14
15 }
```

# Component

- The [CSS element selector](#), 'app-heroes', matches the name of the HTML element that identifies this component within a parent component's template.

```
heroes.component.ts x
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-heroes',
5    templateUrl: './heroes.component.html',
6    styleUrls: ['./heroes.component.css']
7  })
8  export class HeroesComponent implements OnInit {
9
10     constructor() { }
11
12     ngOnInit() {
13     }
14
15 }
```

# export

- The ngOnInit() is a [lifecycle hook](#).
- Angular calls ngOnInit() shortly after creating a component.
- It's a good place to put initialization logic.

```
heroes.component.ts x
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-heroes',
5    templateUrl: './heroes.component.html',
6    styleUrls: ['./heroes.component.css']
7  })
8  export class HeroesComponent implements OnInit {
9
10     constructor() { }
11
12     ngOnInit() {
13     }
14
15 }
```

# import

- Always export the component class so you can import it elsewhere ... like in the AppModule.

```
heroes.component.ts x
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-heroes',
5    templateUrl: './heroes.component.html',
6    styleUrls: ['./heroes.component.css']
7  })
8  export class HeroesComponent implements OnInit {
9
10     constructor() { }
11
12     ngOnInit() {
13     }
14
15 }
```



# import

- heroes.component.ts (hero property)
- hero = 'Windstorm';
- heroes.component.html
- {{hero}}

```
heroes.component.ts x
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-heroes',
5    templateUrl: './heroes.component.html',
6    styleUrls: ['./heroes.component.css']
7  })
8  export class HeroesComponent implements OnInit {
9
10     constructor() { }
11
12     ngOnInit() {
13     }
14
15 }
```

# Εμφάνιση του HeroesComponent

- To display the HeroesComponent, you must add it to the template of the shell AppComponent.
- Remember that app-heroes is the [element selector](#) for the HeroesComponent.
- So add an <app-heroes> element to the AppComponent template file, just below the title.
- src/app/app.component.html
- <h1>{{title}}</h1> <app-heroes></app-heroes>

# interface

- **Create a Hero interface (δεξί κλικ στο app)**

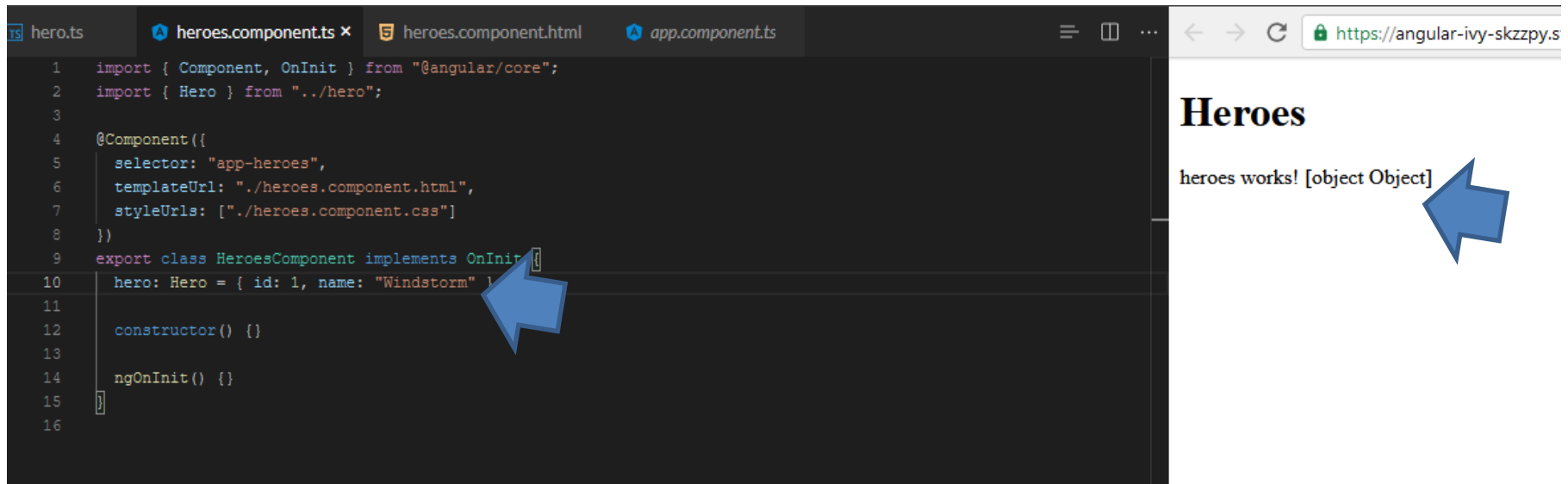
- `src/app/hero.ts`

```
export interface Hero {  
  id: number;  
  name: string;  
}
```

# Import interface

- Return to the HeroesComponent class and import the Hero interface.
- `src/app/heroes/heroes.component.ts`
- `import { Hero } from '../hero';`
- Αλλαγή του Hero ώστε να περιλαμβάνει τιμές:
- `id = 1`
- `Name = Windstorm`

# Αποτέλεσμα...: [object]



```
1 import { Component, OnInit } from "@angular/core";
2 import { Hero } from "../hero";
3
4 @Component({
5   selector: "app-heroes",
6   templateUrl: "../heroes.component.html",
7   styleUrls: ["../heroes.component.css"]
8 })
9 export class HeroesComponent implements OnInit {
10   hero: Hero = { id: 1, name: "Windstorm" };
11
12   constructor() {}
13
14   ngOnInit() {}
15 }
16
```

**Heroes**

heroes works! [object Object]

- Λάθος γιατί είναι από string → object

# HeroesComponent's template

- heroes.component.html

```
<h2>{{hero.name}} Details</h2>
```

```
<div>
```

```
<span>id: </span>{{hero.id}}
```

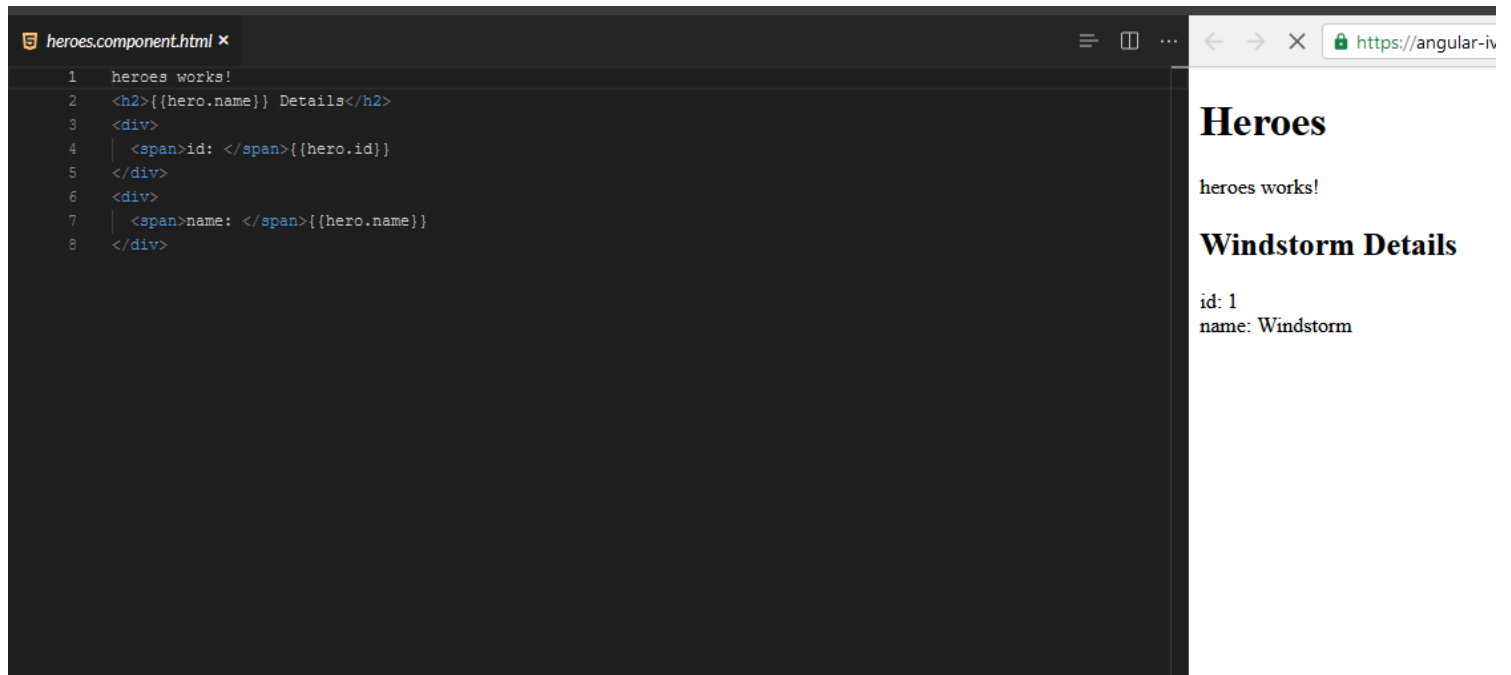
```
</div> <div>
```

```
<span>name: </span>{{hero.name}}
```

```
</div>
```

- heroes.component.html PIPES

```
<h2>{{hero.name | uppercase}} Details</h2>
```



# Edit

- Users should be able to edit the hero name in an `<input>` textbox.
- The textbox should both *display* the hero's name property and *update* that property as the user types.
- That means data flows from the component class *out to the screen* and from the screen *back to the class*.

# Two-way binding

- To automate that data flow, setup a two-way data binding between the `<input>` form element and the `hero.name` property.

*`[(ngModel)]` is Angular's two-way data binding syntax.*

- `src/app/heroes/heroes.component.html`  
(HeroesComponent's template)

```
<div>
```

```
<label>name: <input [(ngModel)]=\"hero.name\" placeholder=\"name\"/>
```

```
</label>
```

```
</div>
```



# Two-way binding

Here it binds the

hero.name property to the

HTML textbox

data can flow *in both directions*:

**from the hero.name property to the textbox, and  
from the textbox back to the hero.name.**

- ***[[ngModel]]*** is Angular's two-way data binding syntax.

```
<div>
```

```
<label>name: <input [(ngModel)]= "hero.name" placeholder="name"/>
```

```
</label>
```

```
</div>
```

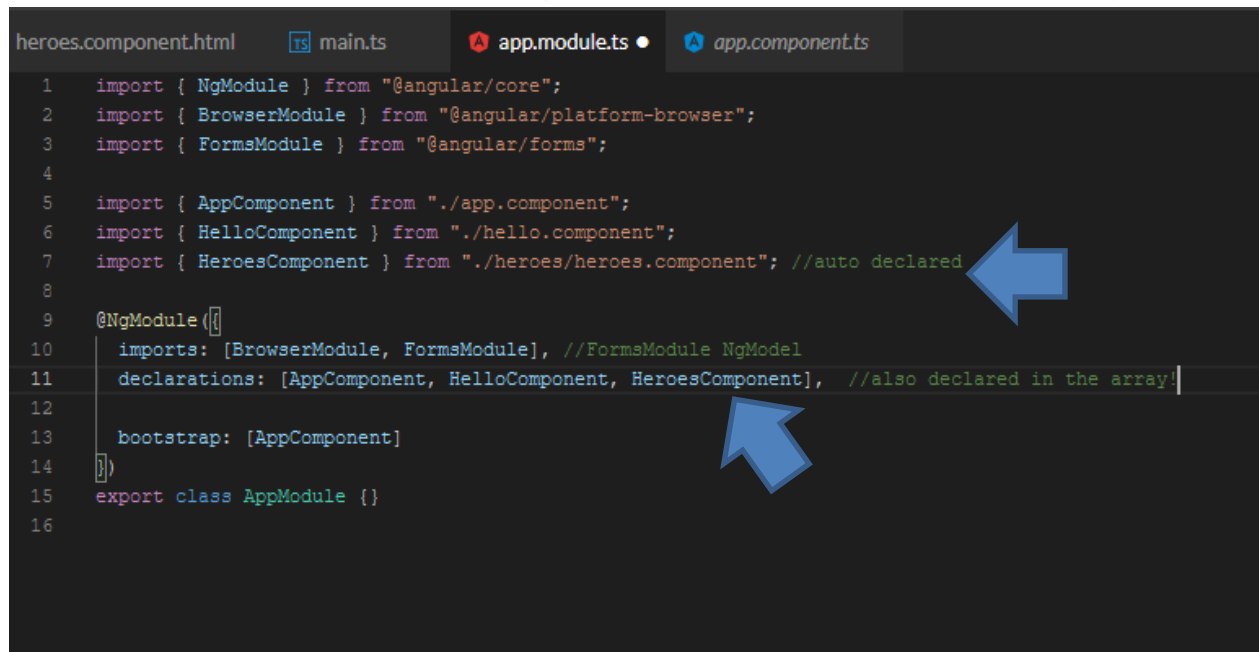
```
... FormsModule
```

# FormsModule

- The Angular CLI generated an AppModule class in **src/app/app.module.ts** when it created the project. This is where you *opt-in* to the [FormsModule](#).
- app.module.ts (FormsModule symbol import)  
import { [FormsModule](#) } from '@angular/forms'; // <-- [NgModel](#) lives here
- Then add [FormsModule](#) to the [@NgModule](#) metadata's imports array, which contains a list of external modules that the app needs.

# Declare HeroesComponent

- Every component must be declared in *exactly one* [NgModule](#).
- Angular CLI declared HeroesComponent in the AppModule when it generated that component.
- Open **src/app/app.module.ts** and find HeroesComponent imported near the top.
- `import { HeroesComponent } from './heroes/heroes.component';`
- The HeroesComponent is declared in the `@NgModule.declarations` array.



```
heroes.component.html  main.ts  app.module.ts •  app.component.ts

1  import { NgModule } from "@angular/core";
2  import { BrowserModule } from "@angular/platform-browser";
3  import { FormsModule } from "@angular/forms";
4
5  import { AppComponent } from "../app.component";
6  import { HelloComponent } from "../hello.component";
7  import { HeroesComponent } from "../heroes/heroes.component"; //auto declared ←
8
9  @NgModule({
10   imports: [BrowserModule, FormsModule], //FormsModule NgModel
11   declarations: [AppComponent, HelloComponent, HeroesComponent], //also declared in the array! ←
12
13   bootstrap: [AppComponent]
14 })
15 export class AppModule {}
16
```

# Ως εδώ...

- CLI to create a second HeroesComponent.
- Displayed the HeroesComponent by adding it to the AppComponent shell.
- UppercasePipe to format the name.
- Two-way data binding with the [ngModel](#) directive.
- AppModule.
- Imported the [FormsModule](#) in the AppModule so that Angular would recognize and apply the [ngModel](#) directive.
- Declaring components in the AppModule and *appreciated that the CLI declared it for you.*

# Display a selection list

- Create a file called `mock-heroes.ts`
  - `src/app/` folder.
- Define a `HEROES` constant as an array of ten heroes
- export it.

`src/app/mock-heroes.ts`

```
import { Hero } from './hero';
export const HEROES: Hero[] = [
  { id: 11, name: 'Dr Nice' },
  { id: 12, name: 'Narco' },
  { id: 13, name: 'Bombasto' },
  { id: 14, name: 'Celeritas' },
  { id: 15, name: 'Magneta' },
  { id: 16, name: 'RubberMan' },
  { id: 17, name: 'Dynamia' },
  { id: 18, name: 'Dr IQ' },
  { id: 19, name: 'Magma' },
  { id: 20, name: 'Tornado' }
];
```

# Displaying heroes

- `src/app/heroes/heroes.component.ts`
- `(import HEROES)`

```
import { HEROES } from '../mock-heroes';
```

# List heroes with **\*ngFor**

- heroes.component.html (heroes template)

```
<h2>My Heroes</h2>
```

```
<ul class="heroes">
```

```
  <li>
```

```
    <span class="badge">{{hero.id}}</span> {{hero.name}}
```

```
  </li>
```

```
</ul>
```

# List heroes with **\*ngFor**

```
<li *ngFor="let hero of heroes">
```

- The \*ngFor is Angular's *repeater* directive. It repeats the host element for each element in a list.

The syntax in this example is as follows:

- <li> is the host element.
- heroes holds the mock heroes list from the HeroesComponent class, the mock heroes list.
- hero holds the current hero object for each iteration through the list.

**Don't forget the asterisk (\*) in front of ngFor.**



# CSS

- src/styles.css (excerpt)
- `/* Application-wide Styles */`
- `h1 { color: #369; font-family: Arial, Helvetica, sans-serif; font-size: 250%; } h2, h3 { color: #444; font-family: Arial, Helvetica, sans-serif; font-weight: lighter; } body { margin: 2em; } body, input[type="text"], button { color: #333; font-family: Cambria, Georgia; } /* everywhere else */  
* { font-family: Arial, Helvetica, sans-serif; }`

# CSS per component

- You may prefer instead to define private styles for a specific component and keep everything a component needs—the code, the HTML, and the CSS—together in one place.
- You define private styles either inline in the [`@Component.styles`](#) array or
- as stylesheet file(s) identified in the [`@Component.styleUrls`](#) array.

# HeroesComponent's private CSS styles

```
/* HeroesComponent's private CSS styles */
```

```
.heroes {  
  margin: 0 0 2em 0;  
  list-style-type: none;  
  padding: 0;  
  width: 15em;  
}  
.heroes li {  
  cursor: pointer;  
  position: relative;  
  left: 0;  
  background-color: #EEE;  
  margin: .5em;  
  padding: .3em 0;  
  height: 1.6em;  
  border-radius: 4px;  
}  
.heroes li:hover {  
  color: #607D8B;  
  background-color: #DDD;  
  left: .1em;  
}  
.heroes li.selected {  
  background-color: #CFD8DC;  
  color: white;  
}  
.heroes li.selected:hover {  
  background-color: #BBD8DC;  
  color: white;  
}  
.heroes .badge {  
  display: inline-block;  
  font-size: small;  
  color: white;  
  padding: 0.8em 0.7em 0 0.7em;  
  background-color: #405061;  
  line-height: 1em;  
  position: relative;  
  left: -1px;  
  top: -4px;  
  height: 1.8em;  
  margin-right: .8em;  
  border-radius: 4px 0 0 4px;
```

# Click Event

- **Master/Detail**
- When the user clicks a hero in the **master** list, the component should display the selected hero's **details** at the bottom of the page.
- Listen for the hero item click event and
- update the hero detail.

# Click Event (2)

- The parentheses around click tell Angular to listen for the <li> element's click event. When the user clicks in the <li>, Angular executes the onSelect(hero) expression.
- heroes.component.html (template excerpt)  
`<li *ngFor="let hero of heroes" (click)="onSelect(hero)">`
- define an onSelect() method in HeroesComponent

# On Select

- define an onSelect() method in HeroesComponent
- src/app/heroes/heroes.component.ts (onSelect)

```
selectedHero: Hero;  
onSelect(hero: Hero): void {  
  this.selectedHero = hero; }
```

# Add a details section

- To click on a hero on the list and reveal details about that hero
- heroes.component.html (selected hero details)

```
<h2>{{selectedHero.name | uppercase}} Details</h2>
```

```
<div><span>id: </span>{{selectedHero.id}}</div>
```

```
<div> <label>name: <input  
[\(ngModel\)]="selectedHero.name"  
placeholder="name"/> </label> </div>
```

# fix - hide empty details with *\*ngIf*

- Μόλις το βάλεις δε παίζει γιατί δεν υπάρχει selectedHero

```
<div *ngIf="selectedHero">
```

```
....
```

```
</div>
```



# Style the selected hero

- .selected CSS class

14	Celeritas
15	Magneta
16	RubberMan

- You just have to apply the .selected class to the <li> when the user clicks it.
- The Angular [class binding](#) makes it easy to add and remove a CSS class conditionally.
- **[class.some-css-class]="some-condition"** to the element you want to style.

`[class.some-css-class]="some-condition"`

- Add the following `[class.selected]` binding to the `<li>` in the HeroesComponent template:
- `heroes.component.html` (toggle the 'selected' CSS class)

*`[class.selected]="hero === selectedHero"`*

**`<li *ngFor="let hero of heroes" [class.selected]="hero === selectedHero" (click)="onSelect(hero)">`**

**`<span class="badge">{{hero.id}}</span> {{hero.name}} </li>`**

# Ως εδώ...

- The Tour of Heroes app displays a list of heroes in a Master/Detail view.
- The user can select a hero and see that hero's details.
- You used \*[ngFor](#) to display a list.
- You used \*[ngIf](#) to conditionally include or exclude a block of HTML.
- You can toggle a CSS style class with a class binding.

# Feature Component

- moving the hero details into a separate, reusable HeroDetailComponent.
- ng generate component hero-detail
- Creates a directory src/app/hero-detail.

# Copy selectedHero & Rename Hero

- replace "selectedHero" with "hero" everywhere in the template
- Error in src/app/hero-detail/hero-detail.component.html (1:13)
- Property 'hero' does not exist on type 'HeroDetailComponent'.

- The HeroDetailComponent template binds to the component's hero property which is of type Hero.
- `src/app/hero-detail/hero-detail.component.ts`  
(import Hero)

```
import { Hero } from '../hero';
```

# Add the `@Input()` hero property

- hero property must be an *Input* property, annotated with the `@Input()` decorator, because the *external* HeroesComponent will bind to it like this.
- **src/app/hero-detail/hero-detail.component.ts** (import Input)  
import { Component, OnInit, Input } from '@angular/core';  
...  
OnInit {....  
    @Input() hero: Hero;  
}

# HeroDetailComponent

- That's the only change you should make to the HeroDetailComponent class.
- There are no more properties.
- There's no presentation logic.
- This component simply receives a hero object through its hero property and displays it.



# Show the HeroDetailComponent

- Don't change the HeroesComponent *class*
- change its *template*.
- **heroes.component.html (HeroDetail binding)**

```
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
```

- [hero]="selectedHero" is an Angular [property binding](#).
- **one way data binding from the selectedHero property of the HeroesComponent to the hero property of the target element, which maps to the hero property of the HeroDetailComponent.**

```

1 heroes works!
2 <div>{{hero.name}} Details</div>
3 <div>
4   <span>id: {{hero.id}}
5 </span>
6 </div>
7   <span>name: {{hero.name}}
8 </span>
9 </div>
10   <label>name: <input [(ngModel)]="hero.name" placeholder="name"/>
11 </label>
12 </div>
13
14 <div>My Heroes</div>
15 <ul class="heroes">
16   <li *ngFor="let hero of heroes" [class.selected]="hero === selectedHero" (click)="select(hero)">
17     <span class="badge">{{hero.id}}</span> {{hero.name}}
18   </li>
19 </ul>
20
21 <app-hero-detail [hero]="selectedHero"></app-hero-detail>

```

heroes works!

## Windstorm Details

id: 1  
name: Windstorm

## My Heroes

- 11 Dr Nice
- 12 Narco
- 13 Bombasto
- 14 Colossal
- 15 Magneta
- 16 RubberMan
- 17 Dynamo
- 18 Dr IQ
- 19 Magma
- 20 Tornado

## RUBBERMAN Details

id: 16  
name: RubberMan

Console

## Ως εδώ...

- separate, reusable HeroDetailComponent.
- [property binding](#) to give the parent HeroesComponent control over the child HeroDetailComponent.
- [@Input decorator](#) to make the hero property available for binding by the external HeroesComponent.

# Services!

- The Tour of Heroes HeroesComponent is currently getting and displaying fake data.
- Components shouldn't fetch or save data directly and they certainly shouldn't knowingly present fake data.
- **They should focus on presenting data and delegate data access to a service.**

# Service

- HeroService that all application classes can use to get heroes.
- **Instead of creating that service with the new keyword**
- **Angular dependency injection to inject it into the HeroesComponent constructor.**

# Share Information

- Services are a great way to share information among classes that *don't know each other*.
- You'll create a `MessageService` and inject it in two places.
- Inject in `HeroService`, which uses the service to send a message.
- Inject in `MessagesComponent`, which displays that message, and also displays the ID when the user clicks a hero.

*ng generate service hero*

# hero.service.ts

```
import { Injectable } from '@angular/core';  
@Injectable({ providedIn: 'root', })  
export class HeroService { constructor() { } }
```

**@Injectable** → participates in the *dependency injection system*

The [@Injectable](#)() decorator accepts a metadata object for the service, the same way the [@Component](#)() decorator did for your component classes.

# HeroService

- Get hero data from anywhere—a web service, local storage, or a mock data source.
- **Removing data access from components means you can change your mind about the implementation anytime, without touching any components.**
- **They don't know how the service works.**



# src/app/hero.service.ts

- src/app/hero.service.ts

@[Injectable](#)({ providedIn: 'root', }) //single instance

```
import { Hero } from './hero';
```

```
import { HEROES } from './mock-heroes';
```

```
getHeroes(): Hero[] { return HEROES; } //return the mock heroes.
```

# Update HeroesComponent

- `src/app/heroes/heroes.component.ts` (import HeroService)

```
import { HeroService } from '../hero.service';
```

```
heroes: Hero[];
```

# Inject the HeroService

- Add a private heroService parameter of type HeroService to the constructor.
- `src/app/heroes/heroes.component.ts`

```
constructor(private heroService: HeroService) {}
```

# Call it in ngOnInit()

**Create a method to retrieve the heroes from the service.**

```
getHeroes(): void { this.heroes =  
this.heroService.getHeroes(); }
```

**While you could call getHeroes() in the constructor, that's not the best practice.**

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

# Observable data

**HeroService.getHeroes()** *synchronous signature*

This will not work in a real app.

**Προετοιμασία για πραγματικό GET Request που επιστρέφει Observable**

Observable is one of the key classes in the [RxJS library](#)

# src/app/hero.service.ts

- **src/app/hero.service.ts** (Observable imports)

```
import { Observable, of } from 'rxjs';
```

```
getHeroes(): Observable<Hero[]> {  
  return of(HEROES); }
```

returns an **Observable<Hero[]>**

*a single value, the array of mock heroes.*

# Subscribe in HeroesComponent

- The HeroService.getHeroes method used to return a Hero[]. Now it returns an Observable<Hero[]>.
- You'll have to adjust to that difference in HeroesComponent.
- **heroes.component.ts (Observable)**  
getHeroes(): void { this.heroService.getHeroes().subscribe(heroes => this.heroes = heroes); }

# Subscribe

- Observable.subscribe() is the critical difference.
- The new version waits for the Observable to emit the array of heroes—which could happen now or several minutes from now.
- The subscribe() method passes the emitted array to the callback, which sets the component's heroes property.
- This asynchronous approach *will work* when the HeroService requests heroes from the server.



# Show messages

- adding a MessagesComponent that displays app messages at the bottom of the screen
- creating an injectable, app-wide MessageService for sending messages to be displayed
- injecting MessageService into the HeroService
- displaying a message when HeroService fetches heroes successfully

# MessagesComponent

ng generate component messages

- **src/app/app.component.html**  
`<h1>{{title}}</h1>`
- `<app-heroes></app-heroes>`
- **`<app-messages></app-messages>`**

# MessageService

- ng generate service message
- src/app/message.service.ts

```
import { Injectable } from '@angular/core';  
@Injectable({ providedIn: 'root', })  
export class MessageService {  
  messages: string[] = [];  
  add(message: string) { this.messages.push(message); }  
  clear() { this.messages = []; } }
```

# MessageService

- The service exposes its cache of messages and two methods:
- `add()` a message to the cache
- `clear()` the cache.

# Inject it into the HeroService

- **src/app/hero.service.ts (import MessageService)**

```
import { MessageService } from './message.service';
```

```
constructor(private messageService: MessageService) {  
}
```

**This is a typical "*service-in-service*" scenario:**

**inject the MessageService into the HeroService which is injected into the HeroesComponent.**

# Send a message from HeroService

- Modify the getHeroes() method to send a message when the heroes are fetched.
- **src/app/hero.service.ts**

```
getHeroes(): Observable<Hero[]> {  
  // TODO: send the message _after_ fetching the heroes  
  this.messageService.add('HeroService: fetched heroes');  
  return of(HEROES); }
```

# Display the message from HeroService

- The MessagesComponent should display all messages, including the message sent by the HeroService when it fetches heroes.
- src/app/messages/messages.component.ts (import MessageService)

```
import { MessageService } from '../message.service';
```

```
constructor(public messageService: MessageService) {}
```

**Angular only binds to *public* component properties.**

# Bind to the MessageService

- src/app/messages/messages.component.html

```
<div *ngIf="messageService.messages.length">
<h2>Messages</h2>
<button class="clear"
(click)="messageService.clear()">clear</button>
<div *ngFor='let message of
messageService.messages'> {{message}} </div>
</div>
```



binds directly to the component's  
messageService.

- The `*ngIf` only displays the messages area if there are messages to show.
- An `*ngFor` presents the list of messages in repeated `<div>` elements.
- An Angular [event binding](#) binds the button's click event to `MessageService.clear()`.
- Βελτιώνεται με CSS

# Click messages

- how to send and display a message each time the user clicks on a hero, showing a history of the user's selections

# Click Messages

- src/app/heroes/heroes.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Hero } from '../hero';
import { HeroService } from '../hero.service';
import { MessageService } from '../message.service';
@Component({ selector: 'app-heroes', templateUrl: './heroes.component.html', styleUrls:
['./heroes.component.css'] })

export class HeroesComponent implements OnInit { selectedHero: Hero; heroes: Hero[];
constructor(private heroService: HeroService, private messageService: MessageService) { }
ngOnInit() { this.getHeroes(); }

onSelect(hero: Hero): void { this.selectedHero = hero; this.messageService.add(`HeroesComponent:
Selected hero id=${hero.id}`); }
getHeroes(): void { this.heroService.getHeroes().subscribe(heroes => this.heroes = heroes); } }
```

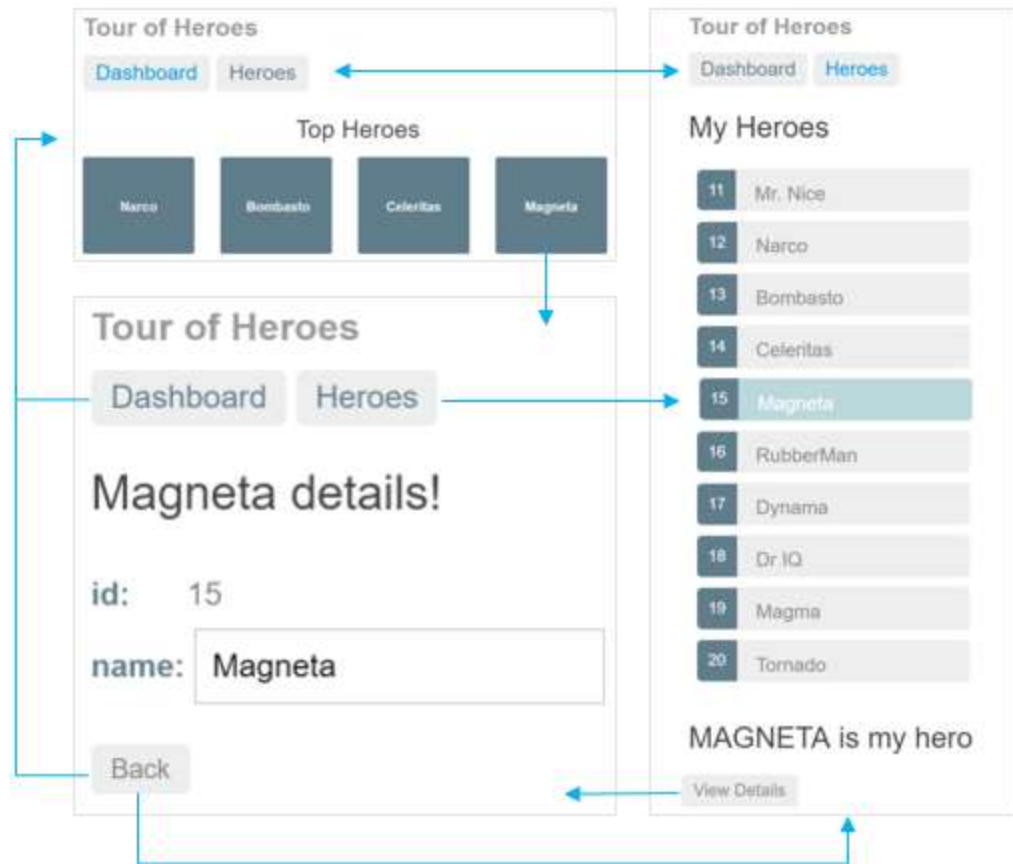
Προσθέστε MONO TA BOLD ή όλοκληρο

# Ως εδώ...

- Refactored data access to the HeroService class.
- HeroService as the *provider* of its service at the root level so that it can be injected anywhere in the app.
- [Angular Dependency Injection](#) to inject it into a component.
- HeroService *get data* method an asynchronous signature.
- the RxJS *Observable* library.
- You used RxJS of() to return an observable of mock heroes (Observable<Hero[]>).
- The component's ngOnInit lifecycle hook calls the HeroService method, not the constructor.
- MessageService for loosely-coupled communication between classes!!!
- The HeroService injected into a component is created with another injected service, MessageService.

# Routing

- in-app navigation



# AppRoutingModule

- the best practice :
  - load and configure the router in a separate, top-level module that is dedicated to routing and imported by the root AppModule
  - module class name is AppRoutingModule
  - app-routing.module.ts (src/app)

# app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HeroesComponent } from './heroes/heroes.component';

const routes: Routes = [
  { path: 'heroes', component: HeroesComponent }
];

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

# Routes

## Routes

The next part of the file is where you configure your routes. Routes tell the Router which view to display when a user clicks a link or pastes a URL into the browser address bar.

AppRoutingModule already imports  
HeroesComponent   ← localhost:303/**heroes**.



# Angular Route

- A typical Angular Route has two properties:
- **path**: a string that matches the URL in the browser address bar.
- **component**: the component that the router should create when navigating to this route.

## Component Router

Crisis Center Heroes



# RouterModule.forRoot()

- The `@NgModule` metadata initializes the router and starts it listening for browser location changes.
- The following line adds the RouterModule to the AppRoutingModuleModule imports array and configures it with the routes in one step by calling `RouterModule.forRoot()`:
- imports: `[ RouterModule.forRoot(routes) ],`
- Next, AppRoutingModuleModule exports RouterModule so it will be available throughout the app.
- exports: `[RouterModule]`

# Register in app.module.ts

- Register in app.module.ts the app-routing.module.ts

```
import { AppRoutingModuleModule } from "../app-routing.module";
```

```
imports: [AppRoutingModule, BrowserModule,  
FormsModule],
```

# Add RouterOutlet

- Αντικατάσταση του `<app-heroes></app-heroes>`
- `src/app/app.component.html` (router-outlet)  
`<h1>{{title}}</h1>`  
`<router-outlet></router-outlet>`  
`<app-messages></app-messages>`

The AppComponent template no longer needs `<app-heroes>` because the app will only display the HeroesComponent when the user navigates to it.

# Add a navigation link (routerLink)

- **The routerLink is the selector for the RouterLink directive that turns user clicks into router navigations.**
- **It's another of the public directives in the RouterModule.**

# App.component.css

- beautify

# dashboard component

```
<h3>Top Heroes</h3>
```

```
<div class="grid grid-pad">
```

```
  <a *ngFor="let hero of heroes" class="col-1-4">
```

```
    <div class="module hero">
```

```
      <h4>{{hero.name}}</h4>
```

```
    </div>
```

```
  </a>
```

```
</div>
```

# dashboard component

```
import { Component, OnInit } from '@angular/core';
import { Hero } from '../hero';
import { HeroService } from '../hero.service';

@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrls: [ './dashboard.component.css' ]
})
export class DashboardComponent implements OnInit {
  heroes: Hero[] = [];

  constructor(private heroService: HeroService) {}

  ngOnInit() {
    this.getHeroes();
  }

  getHeroes(): void {
    this.heroService.getHeroes()
      .subscribe(heroes => this.heroes = heroes.slice(1, 5));
  }
}
```



# dashboard component

- Δημιουργία dashboard component

## CSS

```
/* DashboardComponent's private CSS styles */
[class*='col-'] {
  float: left;
  padding-right: 20px;
  padding-bottom: 20px;
}
[class*='col-']:last-of-type {
  padding-right: 0;
}
a {
  text-decoration: none;
}
*, *:after, *:before {
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
}
h3 {
  text-align: center;
  margin-bottom: 0;
}
h4 {
  position: relative;
}
.grid {
  margin: 0;
}
.col-1-4 {
  width: 25%;
}
.module {
  padding: 20px;
  text-align: center;
  color: #eee;
  max-height: 120px;
  min-width: 120px;
  background-color: #3f525c;
  border-radius: 2px;
}
```

# dashboard

- The template presents a grid of hero name links.
- The `*ngFor` repeater creates as many links as are in the component's heroes array.
- The links are styled as colored blocks by the `dashboard.component.css`.
- The links don't go anywhere yet but they will shortly.
- The class is similar to the `HeroesComponent` class.
- It defines a heroes array property.
- The constructor expects Angular to inject the `HeroService` into a private `heroService` property.
- The `ngOnInit()` lifecycle hook calls `getHeroes()`.

# dashboard

- `.subscribe(heroes => this.heroes =  
heroes.slice(1, 5));`

This `getHeroes()` returns the sliced list of heroes at positions 1 and 5, returning only four of the Top Heroes (2nd, 3rd, 4th, and 5th).

# Add the dashboard route

```
src/app/app-routing.module.ts (import  
DashboardComponent)
```

```
import { DashboardComponent } from  
'./dashboard/dashboard.component';
```

```
....
```

```
{ path: 'dashboard', component: DashboardComponent },
```

# Add a default route

- This route redirects a URL that fully matches the empty path to the route whose path is '/dashboard'.
- **src/app/app-routing.module.ts**

```
{ path: "", redirectTo: '/dashboard', pathMatch:  
'full' },
```

# Add dashboard link to the shell

- src/app/app.component.html

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

```
<nav>
```

```
  <a routerLink="/dashboard">Dashboard</a>
```

```
  <a routerLink="/heroes">Heroes</a>
```

```
</nav>
```

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

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

ΕΠΙΤΕΛΟΥΣ NAVIGATION!

# Navigating to hero details

- The user should be able to get to these details in three ways.
  - 1.By clicking a hero in the dashboard.
  - 2.By clicking a hero in the heroes list.
  - 3.*By pasting a "deep link" URL into the browser address bar that identifies the hero to display.*

Ως εδώ...



# Delete hero details from HeroesComponent

- Open the HeroesComponent template (heroes/heroes.component.html)
- delete the `<app-hero-detail>` element from the bottom.

# Add a *hero detail* route

- URL like ~/detail/11 would be a good URL
- `src/app/app-routing.module.ts`  
(import `HeroDetailComponent`)

```
import { HeroDetailComponent } from './hero-detail/hero-detail.component';
```

```
...
```

```
{ path: 'detail/:id', component: HeroDetailComponent },
```

The colon (:) in the path indicates that `:id` is a placeholder for a specific hero id.

# DashboardComponent hero links

- src/app/dashboard/dashboard.component.html  
(hero links)

```
<a *ngFor="let hero of heroes" class="col-1-4"
  routerLink="/detail/{{hero.id}}">
  <div class="module hero">
    <h4>{{hero.name}}</h4>
  </div>
</a>
```

# HeroesComponent hero links

- src/app/heroes/heroes.component.html (list with onSelect) μετατροπή σε → list with link
- (αφαίρεση του onSelect από το li)

```
<ul class="heroes">
```

```
<li *ngFor="let hero of heroes" >
```

```
<a routerLink="/detail/{{hero.id}}">
```

```
<span class="badge">{{hero.id}}</span>  
{{hero.name}} </a> </li> </ul>
```

# HeroesComponent clear template

export class HeroesComponent implements

OnInit {

heroes: Hero[];

constructor(private heroService: HeroService) { }

ngOnInit() { this.getHeroes(); }

getHeroes(): void { this.heroService.getHeroes()  
 .subscribe(heroes => this.heroes = heroes); }

}

# HeroComponent Html

```
<h2>My Heroes</h2>
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <a routerLink="/detail/{{hero.id}}">
      <span class="badge">{{hero.id}}</span> {{hero.
name}}
    </a>
  </li>
</ul>
```

# Routable HeroDetailComponent

- the parent HeroesComponent set the HeroDetailComponent.hero property and the HeroDetailComponent displayed the hero.
- HeroesComponent doesn't do that anymore.
- Now the router creates the HeroDetailComponent in response to a URL such as ~/detail/11.
- The HeroDetailComponent needs a new way to obtain the hero-to-display.

# Create Route for HeroDetail

- Get the route that created it
- Extract the id from the route
- Acquire the hero with that id from the server via the HeroService



# Προσθήκες που θα γίνουν

- The [ActivatedRoute](#) holds information about the route to this instance of the HeroDetailComponent.
  - This component is interested in the route's parameters extracted from the URL. The "id" parameter is the id of the hero to display.
- The [HeroService](#) gets hero data from the remote server
  - this component will use it to get the hero-to-display.
- The [location](#) is an Angular service for interacting with the browser.
  - You'll use it [later](#) to navigate back to the view that navigated here.

# hero-detail.component.ts

- import { [ActivatedRoute](#) } from '@angular/router'; import { [Location](#) } from '@angular/common'; import { HeroService } from '../hero.service';
- Inject the [ActivatedRoute](#), HeroService, and [Location](#) services into the constructor, saving their values in private fields:
- constructor( private route: [ActivatedRoute](#), private heroService: HeroService, private location: [Location](#) ) {}

# Extract the id route parameter

- **hero-detail.component.ts**
- In the ngOnInit() [lifecycle hook](#) call getHero() and define it as follow
- ```
ngOnInit(): void { this.getHero(); } getHero(): void  
{ const id =  
+this.route.snapshot.paramMap.get('id');  
this.heroService.getHero(id) .subscribe(hero =>  
this.hero = hero); }
```

# hook methods

- Angular calls [hook methods](#) in the following order:
  - ngOnChanges: When an [input/output](#) binding value changes.
  - ngOnInit: After the first ngOnChanges.
  - ngDoCheck: Developer's custom change detection.
  - ngAfterContentInit: After component content initialized.
  - ngAfterContentChecked: After every check of component content.
  - ngAfterViewInit: After a component's views are initialized.
  - ngAfterViewChecked: After every check of a component's views.
  - ngOnDestroy: Just before the directive is destroyed.

# heroService.getHero

```
getHero(id: number): Observable<Hero> {  
  // TODO: send the message _after_ fetching the  
  hero  
  
  this.messageService.add(`HeroService: fetched  
  hero id=${id}`);  
  
  return of(HEROES.find(hero => hero.id === id));  
}
```

# rxjs based getHero()

- getHero() has an asynchronous signature. It returns a *mock hero* as an Observable, using the RxJS of() function.
- You'll be able to re-implement getHero() as a real Http request without having to change the HeroDetailComponent that calls it.

# Hero detail deep link

- You can click a hero in the dashboard or in the heroes list and navigate to that hero's detail view.
- Try `http ... /detail/15`

# hero-detail.component (back button)

- .ts

```
goBack(): void { this.location.back(); }
```

- .html

```
<button (click)="goBack()">go back</button>
```



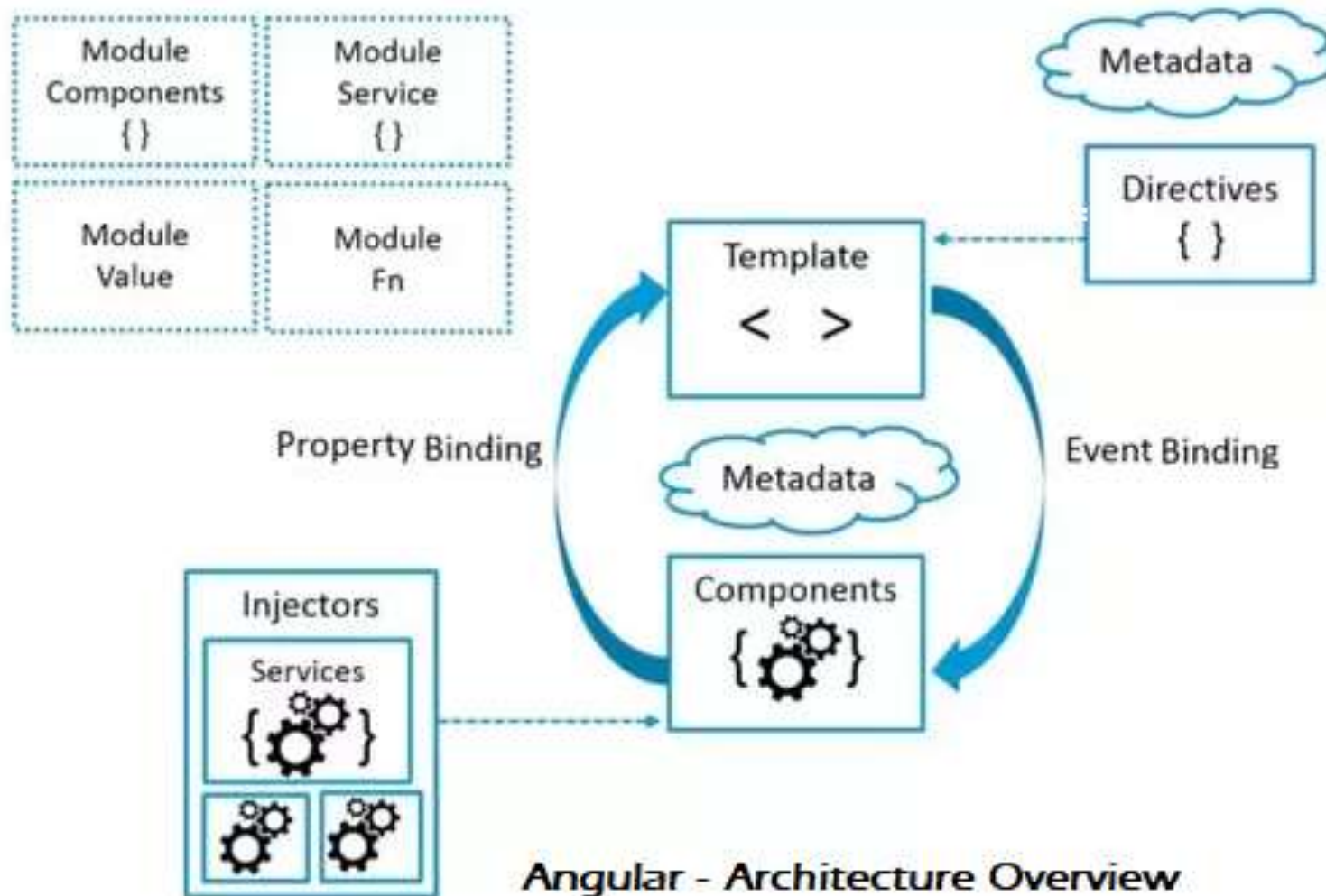
# Hero-detail CSS

```
/* HeroDetailComponent's private CSS styles */
label {
  display: inline-block;
  width: 3em;
  margin: .5em 0;
  color: #607D8B;
  font-weight: bold;
}
input {
  height: 2em;
  font-size: 1em;
  padding-left: .4em;
}
button {
  margin-top: 20px;
  font-family: Arial;
  background-color: #eee;
  border: none;
  padding: 5px 10px;
  border-radius: 4px;
  cursor: pointer;
}
button:hover {
  background-color: #cfd8dc;
}
button:disabled {
  background-color: #eee;
  color: #ccc;
  cursor: auto;
}
```

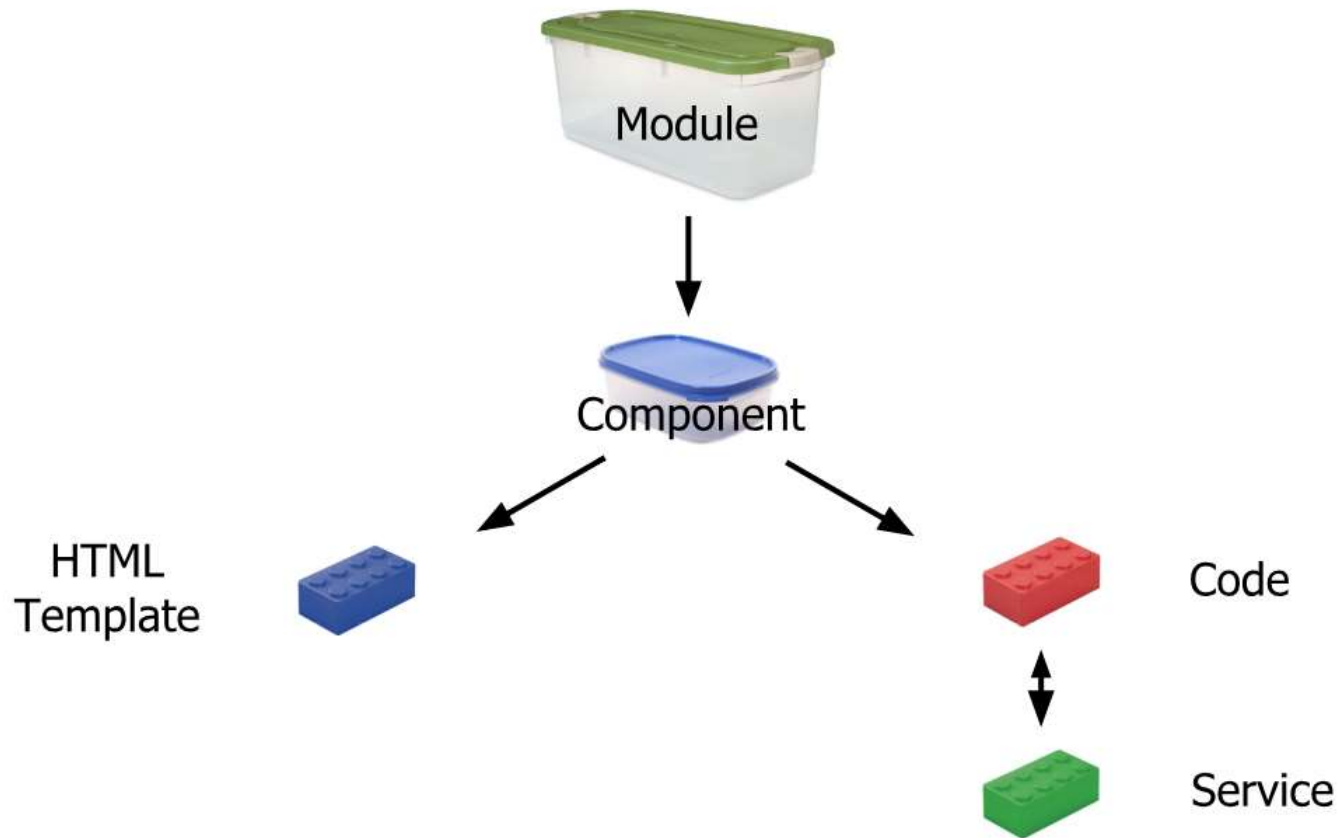
# Ως εδώ:

- You added the Angular router to navigate among different components.
- You turned the AppComponent into a navigation shell with <a> links and a <[router-outlet](#)>.
- You configured the router in an AppRoutingModuleModule
- You defined simple routes, a redirect route, and a parameterized route.
- You used the [routerLink](#) directive in anchor elements.
- You refactored a tightly-coupled master/detail view into a routed detail view.
- You used router link parameters to navigate to the detail view of a user-selected hero.
- You shared the HeroService among multiple components.

# Angular General Architecture



# Module vs Component



# Module

- **NgModule** is one of the first basic structures you meet when coding an app with Angular, but it's also the most complex, because of **different scopes**.
- Every Angular app has a *root module*, named AppModule
  - provides the mechanism that launches the application.
- NgModules can import functionality from other NgModules, and allow their own functionality to be exported and used by other NgModules.
- NgModules are shareable (independent) units in Angular apps.
- Types of modules:
  - system (Router, etc)
  - other modules
  - custom modules
  - third party modules (ngrx, rxjs, etc).

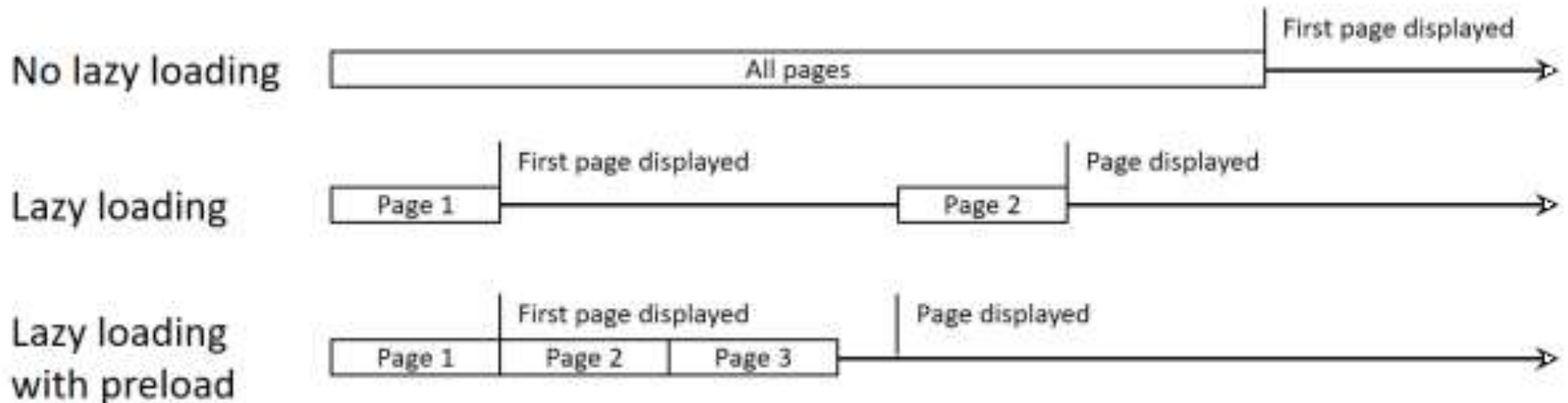
# Component

- A **component** controls a patch of the screen called a *view*.
- It is the most basic building block of a UI in an Angular application. It comes as a directive with a special decorator (`@Component`) and template.
- Component decorator allows you to mark a class as an Angular component and provide additional metadata that determines how the component should be processed, instantiated, and used at runtime.

# ***Module vs component***

- ***The module*** can be considered as a collection of *components*, directives, services, pipes, helpers, etc.
  - (One of many *modules* combines up to make an **Application**.)
- Each ***component*** can use other *components*, which are declared in the same *module*. To use components declared in other *modules*, they need to be exported from that *module* and the *module* needs to be imported into our *module* where we need that functionality.

# No lazy vs Preload vs Lazy Loading





# No lazy vs Lazy Loading

With lazy loading



Without lazy loading



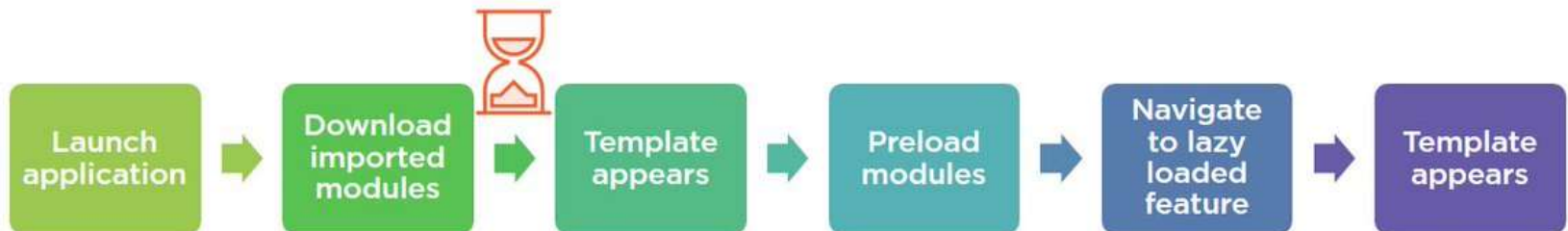
# Preload vs Lazy Loading

## Preloading Feature Modules

### Lazy Loading



### Preloading (Eager Lazy Loading)



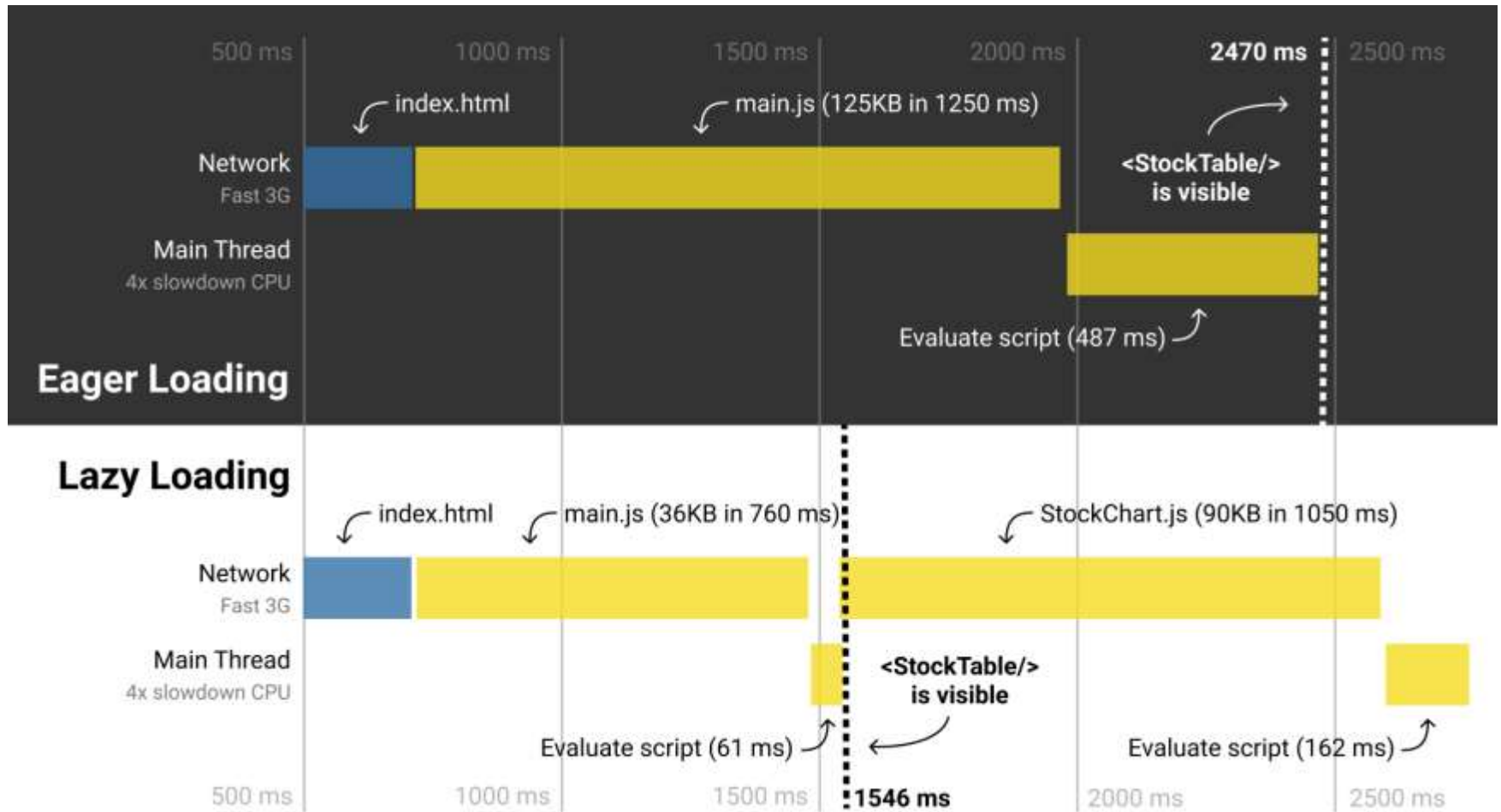
# Lazy Loading

- Modules in Angular can be [lazy-loaded](#)
  - means that they are loaded when needed, not always!
  - Lazy loading significantly improves the performance of an Angular app.

# Pre-Loading

- In addition to loading modules on-demand, you can load modules asynchronously with preloading.
- The AppModule is eagerly loaded when the application starts, meaning that it loads right away.
- Now the AdminModule loads only when the user clicks on a link, which is called lazy loading.
- Preloading allows you to load modules in the background
  - so that the data is ready to render when the user activates a particular route.
- Consider the Crisis Center. It isn't the first view that a user sees. By default, the Heroes are the first view.
  - For the smallest initial payload and fastest launch time, you should eagerly load the AppModule and the HeroesModule.
- You could lazy load the Crisis Center. But you're almost certain that the user will visit the Crisis Center within minutes of launching the app.
- Ideally, the app would launch with just the AppModule and the HeroesModule loaded and then, almost immediately, load the CrisisCenterModule in the background. By the time the user navigates to the Crisis Center, its module will have been loaded and ready.

# Lazy vs Eager Loading



# Get Data from Server

- Angular's [HttpClient](#).
  - The HeroService gets hero data with HTTP requests.
  - Users can add, edit, and delete heroes and save these changes over HTTP.
  - Users can search for heroes by name.

# Enable HTTP services

- app.module.ts (HttpClientModule import)

```
import { HttpClientModule } from  
'@angular/common/http';
```

```
@NgModule({ imports:  
[ HttpClientModule, ],  
})
```

# In-memory Web API

- By using the In-memory Web API, you won't have to set up a server to learn about [HttpClient](#).
- app.module.ts (In-memory Web API imports)

```
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';  
import { InMemoryDataService } from './in-memory-data.service';
```

[HttpClientModule](#),

```
HttpClientInMemoryWebApiModule.forRoot( InMemoryDataService,  
{ dataEncapsulation: false } )
```



# InMemoryData

- in-memory-data.service.ts

```
import { Injectable } from '@angular/core';
import { InMemoryDbService } from 'angular-in-memory-web-api';
import { Hero } from './hero';

@Injectable({
  providedIn: 'root',
})
export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    const heroes = [
      { id: 11, name: 'Dr Nice' },
      { id: 12, name: 'Narco' },
      { id: 13, name: 'Bombasto' },
      { id: 14, name: 'Celeritas' },
      { id: 15, name: 'Magnetia' },
      { id: 16, name: 'RubberMan' },
      { id: 17, name: 'Dynamo' },
      { id: 18, name: 'Dr IQ' },
      { id: 19, name: 'Magma' },
      { id: 20, name: 'Tornado' }
    ];
    return {heroes};
  }

  // Overrides the genId method to ensure that a hero always has an id.
  // If the heroes array is empty,
  // the method below returns the initial number (11).
  // if the heroes array is not empty, the method below returns the highest
  // hero id + 1.
  genId(heroes: Hero[]): number {
    return heroes.length > 0 ? Math.max(...heroes.map(hero => hero.id)) + 1 : 11;
  }
}
```

- The in-memory-data.service.ts file will take over the function of mock-heroes.ts.

# Heroes and HTTP

- hero.service.ts (import HTTP symbols)

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
```

- inject [HttpClient](#) into the constructor in a private property called [http](#).

```
constructor( private http: HttpClient, private messageService: MessageService) { }
```

- keep injecting the MessageService

```
private log(message: string) { this.messageService.add(`HeroService: ${message}`); }
```

- Define the heroesUrl of the form :base/:collectionName with the address of the heroes resource on the server.

```
private heroesUrl = 'api/heroes'; // URL to web api
```

- **Get heroes with [HttpClient](#)**

- Αντικατάσταση:

```
getHeroes(): Observable<Hero[]> { return this.http.get<Hero[]>(this.heroesUrl) }
```

## Refresh the browser

You've swapped of() for http.get() and the app keeps working without any other changes because both functions return an Observable<Hero[]>.

# HttpClient methods return one value

- All HttpClient methods return an RxJS Observable of something.
- an observable *can* return multiple values over time.
- An observable from HttpClient always emits a single value and then completes, never to emit again.
- This particular HttpClient.get() call returns an `Observable<Hero[]>`; that is, "*an observable of hero arrays*". In practice, it will only return a single hero array.

# HttpClient.get() returns response data

- HttpClient.get() returns the body of the response as an untyped JSON object by default.
- Applying the optional type specifier, `<Hero[]>` , adds TypeScript capabilities, which reduce errors during compile time.

# Error handling

- To catch errors, you **"pipe" the observable** result from `http.get()` through an RxJS `catchError()` operator.
- `hero.service.ts`
- `import { catchError, map, tap } from 'rxjs/operators';`

# HandleError

- hero.service.ts

```
getHeroes(): Observable<Hero[]> {  
  return this.http.get<Hero[]>(this.heroesUrl)  
    .pipe(  
      catchError(this.handleError<Hero[]>('getHeroes'  
        , [])) );  
}
```

# HandleError(2)

- Hero.service.ts

```
/** * Handle Http operation that failed. * Let the app continue. * @param  
operation - name of the operation that failed * @param result - optional  
value to return as the observable result */  
private handleError<T>(operation = 'operation', result?: T) {  
  return (error: any): Observable<T> => {  
    // TODO: send the error to remote logging infrastructure  
    console.error(error);  
    // log to console instead // TODO: better job of transforming error for user  
    consumption  
    this.log(`${operation} failed: ${error.message}`);  
    // Let the app keep running by returning an empty result.  
    return of(result as T);  
  }  
}
```



# HandleError type parameter

- each service method returns a different kind of Observable result, handleError() takes a type parameter so it can return the safe value as the type that the app expects.
- **hero.service.ts**

```
getHeroes(): Observable<Hero[]> { return  
  this.http.get<Hero[]>(this.heroesUrl)  
    .pipe( tap(_ => this.log('fetched heroes')),  
      catchError(this.handleError<Hero[]>('getHeroes',  
        [])) ); }
```

# GET by ID

- Most web APIs support a *get by id* request in the form `:baseUrl/:id`.
- `Hero.service.ts`
- ```
getHero(id: number): Observable<Hero> { const url = `${this.heroesUrl}/${id}`; return this.http.get<Hero>(url).pipe( tap(_ => this.log(`fetched hero id=${id}`)), catchError(this.handleError<Hero>(`getHero id=${id}`)) ); }
```

# three significant differences from getHeroes()

- getHero() constructs a request URL with the desired hero's id.
- The server should respond with a single hero rather than an array of heroes.
- getHero() returns an Observable<Hero> ("*an observable of Hero objects*") rather than an observable of hero *arrays* .

# CRUD

- Use frontend Angular to CRUD
- Create (insert)
- Update
- Delete

# Update heroes

- Edit a hero's name in the hero detail view. As you type, the hero name updates the heading at the top of the page. But when you click the "go back button", the changes are lost.
- If you want changes to persist, you must write them back to the server.
- At the end of the hero detail template, add a save button

# Save button

- hero-detail.component.html (save)  
`<button (click)="save()">save</button>`
- hero-detail.component.ts (save)  
`save(): void {  
 this.heroService.updateHero(this.hero)  
 .subscribe(() => this.goBack()); }  
}`

# Add HeroService.updateHero()

- hero.service.ts (update)

```
updateHero(hero: Hero): Observable<any> {  
  return this.http.put(this.heroesUrl, hero,  
    this.httpOptions).pipe(  
    tap(_ => this.log(`updated hero id=${hero.id}`)),  
    catchError(this.handleError<any>('updateHero')) ); }
```

```
httpOptions = { headers: new HttpHeaders(  
  'Content-Type': 'application/json' ) };
```

# Update

The [HttpClient.put\(\)](#) method takes three parameters:

- the URL
- the data to update (the modified hero in this case)
- options

The URL is unchanged. The heroes web API knows which hero to update by looking at the hero's id.

- The heroes web API expects a special header in HTTP save requests.



# Create (Insert)

- heroes.component.html (add)

(κάτω από το update/save!)

```
<div> <label>Hero name:
```

```
<input #heroName /> </label>
```

```
<!-- (click) passes input value to add() and then  
clears the input -->
```

```
<button (click)="add(heroName.value);  
heroName.value=''"> add </button>
```

```
</div>
```

# Add

- When the given name is non-blank, the handler creates a Hero-like object from the name (it's only missing the id) and passes it to the services addHero() method.
- When addHero() saves successfully, the subscribe() callback receives the new hero and pushes it into to the heroes list for display.
- addHero() differs from updateHero() in two ways:
  - It calls [HttpClient.post\(\)](#) instead of put().
  - It expects the server to generate an id for the new hero, which it returns in the Observable<Hero> to the caller.

# Function add

- heroes.component.ts (add)

```
add(name: string): void {  
  name = name.trim();  
  if (!name) { return; }  
  this.heroService.addHero({ name } as Hero)  
    .subscribe(hero => { this.heroes.push(hero); }); }
```

# Service Add

- Post Hero

```
/** POST: add a new hero to the server */  
addHero(hero: Hero): Observable<Hero> {  
    return this.http.post<Hero>(this.heroesUrl, hero,  
    this.httpOptions).pipe(  
        tap((newHero: Hero) => this.log(`added hero w/  
id=${newHero.id}`)),  
        catchError(this.handleError<Hero>('addHero')) ); }  
}
```

# Delete a hero

- heroes.component.html

```
<button class="delete" title="delete hero"  
(click)="delete(hero)">x</button>
```

- heroes.component.ts (delete)

```
delete(hero: Hero): void { this.heroes =  
this.heroes.filter(h => h !== hero);  
this.heroService.deleteHero(hero).subscribe(); }
```

# Σβήνουμε από τη λίστα

- Φορτώνονται όλα ξανά αν πάω στο dashboard και ξανάεπιστρέψω...
- Although the component delegates hero deletion to the HeroService, it remains responsible for updating its own list of heroes. The component's delete() method immediately removes the *hero-to-delete* from that list, anticipating that the HeroService will succeed on the server.
- There's really nothing for the component to do with the Observable returned by heroService.delete() **but it must subscribe anyway**.
- If you neglect to subscribe(), the service will not send the delete request to the server. As a rule, an Observable *does nothing* until something subscribes.

# Delete in Service

- add a deleteHero() method to HeroService like this.

- hero.service.ts (delete)

```
/** DELETE: delete the hero from the server */
deleteHero(hero: Hero | number): Observable<Hero> {
  const id = typeof hero === 'number' ? hero : hero.id;
  const url = `${this.heroesUrl}/${id}`;
  return this.http.delete<Hero>(url, this.httpOptions).pipe(
    tap(_ => this.log(`deleted hero id=${id}`)),
    catchError(this.handleError<Hero>('deleteHero')) ); }
```

# Delete sends..

- deleteHero() calls [HttpClient.delete\(\)](#).
- The URL is the heroes resource URL plus the id of the hero to delete.
- You don't send data as you did with put() and post().
- You still send the httpOptions.



# Search by name

## Hero Search

ma
Magneta
RubberMan
Dynama
Magma

- chain Observable operators together
- min the number of similar HTTP req
- consume network bandwidth economically
- heroes search feature to the Dashboard.
- As the user types a name into a search box,
- you'll make repeated HTTP requests for heroes filtered by that name.
- **goal is to issue only as many requests as necessary.**

# HeroService.searchHeroes()

- **hero.service.ts**

```
/* GET heroes whose name contains search term */
searchHeroes(term: string): Observable<Hero[]> {
  if (!term.trim()) {
    // if not search term, return empty hero array.
    return of([]);
  }
  return this.http.get<Hero[]>(`${this.heroesUrl}/?name=${term}`).pipe(
    tap(x => x.length ?
      this.log(`found heroes matching "${term}"`) :
      this.log(`no heroes matching "${term}"`)),
    catchError(this.handleError<Hero[]>('searchHeroes', []))
  );
}
```

# SearchHeroes()

- it closely resembles getHeroes(),
- the only significant difference being the URL
- includes a query string with the search term.

# Add search to the Dashboard

- dashboard.component.html

```
<h3>Top Heroes</h3> <div class="grid grid-pad"> <a *ngFor="let  
hero of heroes" class="col-1-4" routerLink="/detail/{{hero.id}}">  
<div class="module hero"> <h4>{{hero.name}}</h4> </div> </a>  
</div>
```

```
<app-hero-search></app-hero-search>
```

# Hero-Search Component

- add a component with a selector that matches <app-hero-search>
- hero-search.component.html

```
<div id="search-component">
<h4><label for="search-box">Hero Search</label></h4> <input
#searchBox id="search-box" (input)="search(searchBox.value)" />
<ul class="search-result">
  <li *ngFor="let hero of heroes$ | async" >
    <a routerLink="/detail/{{hero.id}}"> {{hero.name}} </a>
  </li>
</ul> </div>
```

# AsyncPipe

- Notice that the `*ngFor` iterates over a list called **heroes\$**, not heroes.
- The **\$** is a convention that indicates heroes\$ is an Observable, not an array.

`<li *ngFor="let hero of heroes$ | async" >`

- Since `*ngFor` can't do anything with an Observable,
- use the pipe character (`|`) followed by `async`.
- This identifies Angular's [AsyncPipe](#) and subscribes to an Observable automatically so you won't have to do so in the component class.

# HeroSearchComponent class

- hero-search.component.ts

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

import { Observable, Subject } from 'rxjs';

import {
  debounceTime, distinctUntilChanged, switchMap
} from 'rxjs/operators';

import { Hero } from '../hero';
import { HeroService } from '../hero.service';

@Component({
  selector: 'app-hero-search',
  templateUrl: './hero-search.component.html',
  styleUrls: [ './hero-search.component.css' ]
})
export class HeroSearchComponent implements OnInit {
  heroes$: Observable<Hero[]>;
  private searchTerms = new Subject<string>();

  constructor(private heroService: HeroService) {}

  // Push a search term into the observable stream.
  search(term: string): void {
    this.searchTerms.next(term);
  }

  ngOnInit(): void {
    this.heroes$ = this.searchTerms.pipe(
      // wait 300ms after each keystroke before considering the term
      debounceTime(300),

      // ignore new term if same as previous term
      distinctUntilChanged(),

      // switch to new search observable each time the term changes
      switchMap((term: string) => this.heroService.searchHeroes(term)),
    );
  }
}
```

# Heroes\$

- Notice the declaration of heroes\$ as an Observable:
- src/app/hero-search/hero-search.component.ts

heroes\$: Observable<Hero[]>;

set it in [ngOnInit\(\)](#).



# The searchTerms property RxJS Subject.

- hero-search.component.ts

```
private searchTerms = new Subject<string>();  
// Push a search term into the observable  
stream. search(term: string): void {  
  this.searchTerms.next(term);  
}
```

- A Subject is both a source of observable values and an Observable itself.
- You can subscribe to a Subject as you would any Observable.
- You can also push values into that Observable by calling its next(value) method as the search() method does.

# Event binding

- The event binding to the textbox's input event calls the search() method.

```
<input #searchBox id="search-box"  
(input)="search(searchBox.value)" />
```

- Every time the user types in the textbox, the binding calls search() with the textbox value, a "search term".
- The searchTerms becomes an Observable emitting a steady stream of search terms.

# Chaining RxJS operators

- Passing a new search term directly to the `searchHeroes()` after every user keystroke would create an excessive amount of HTTP requests (taxing server resources and burning through data plans)
- Instead, the `ngOnInit()` method pipes the `searchTerms` observable through a sequence of RxJS operators that reduce the number of calls to the `searchHeroes()`
- Returning an observable of timely hero search results (each a `Hero[]`).

# delays

- hero-search.component.ts

```
this.heroes$ = this.searchTerms.pipe(  
  // wait 300ms after each keystroke before  
  considering the term debounceTime(300),  
  // ignore new term if same as previous term  
  distinctUntilChanged(),  
  // switch to new search observable each time the  
  term changes switchMap((term: string) =>  
    this.heroService.searchHeroes(term)), );
```