

**[tinyurl.com/iitc2019](https://tinyurl.com/iitc2019)**

# Hello Angular!



# What is Angular?

Angular is a framework (JavaScript library) for building client applications in HTML, CSS, and either JavaScript or a language like TypeScript that can be compiled (more accurately, transpiled) to JavaScript.

<https://angular.io/>



# What is TypeScript?



TypeScript is a **superset** of JavaScript. That means any valid JavaScript code is valid TypeScript code.

But many prefer TypeScript because it has additional features that we don't have in the current version of JavaScript that most browsers understand.

So, when building Angular applications, we need to have our **TypeScript code converted into JavaScript code** that browsers can understand.

This process is called **transpilation** which is the combination of translate and compile. And that's the job of the **TypeScript compiler**.

# Why do I need a framework like Angular?

A common question a lot of beginners ask me is:

*“Why do we need Angular in the first place?”*

*“What’s wrong with the plain old vanilla JavaScript and jQuery?”.*

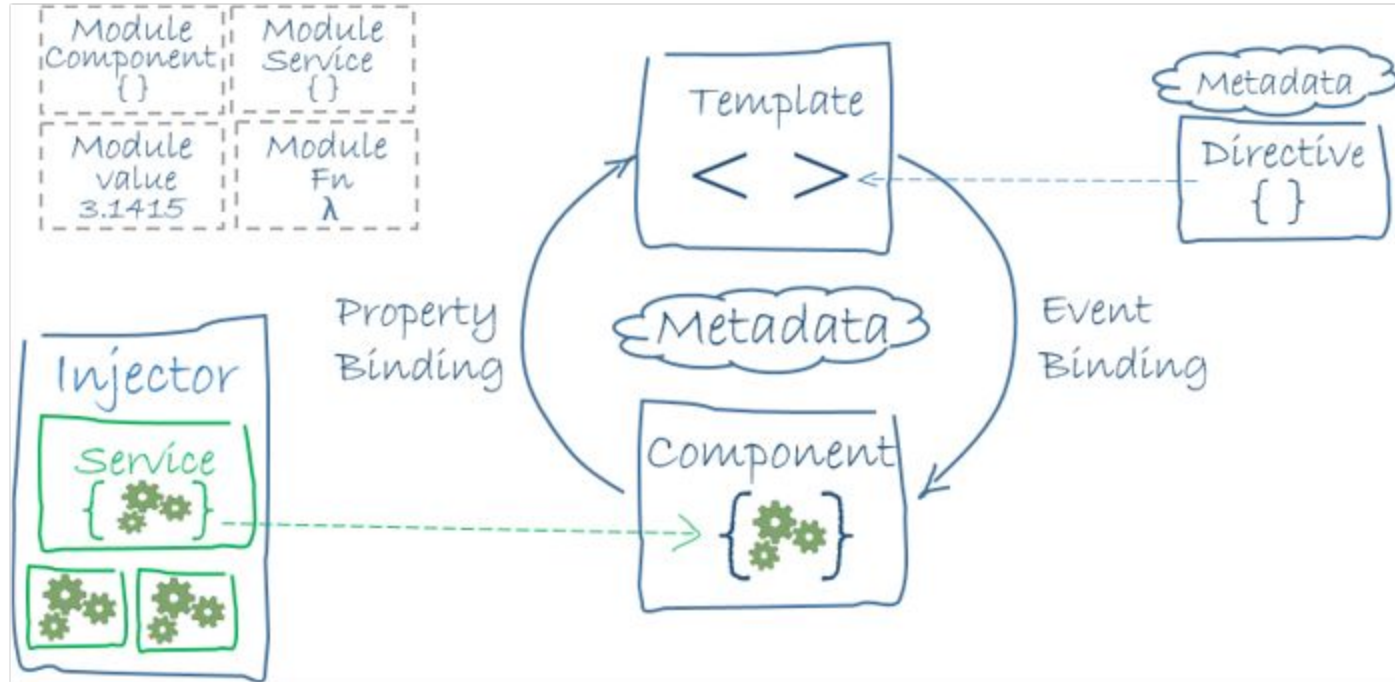
There is nothing inherently wrong with using vanilla JavaScript/jQuery.

But as your **application grows**, **structuring your code** in a clean and **maintainable** and more importantly, **testable** way, becomes harder and harder.

Using a framework like Angular, makes your life far easier.

Once you master Angular, **you’ll be able to build client applications faster and easier.**

# Architecture of Angular Apps



The **basic building blocks** of an Angular application are **NgModules**.

**NgModules** provide a **compilation context for components**.

**NgModules** collect related code into **functional sets**;

**An Angular app** is defined by a **set of NgModules**.

An app always has at least a **root module** that enables **bootstrapping**,  
and typically a **app has many more feature modules**.

**Components** define **views**, which are sets of screen elements that Angular can choose among and modify according to your program logic and data.

Every app has at least a **root component**.

**Components use services**, which provide specific **functionality not directly related to views**.

Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient.



Both **components** and **services** are **simply classes**, with **decorators** that mark their type and provide metadata that tells Angular how to use them.

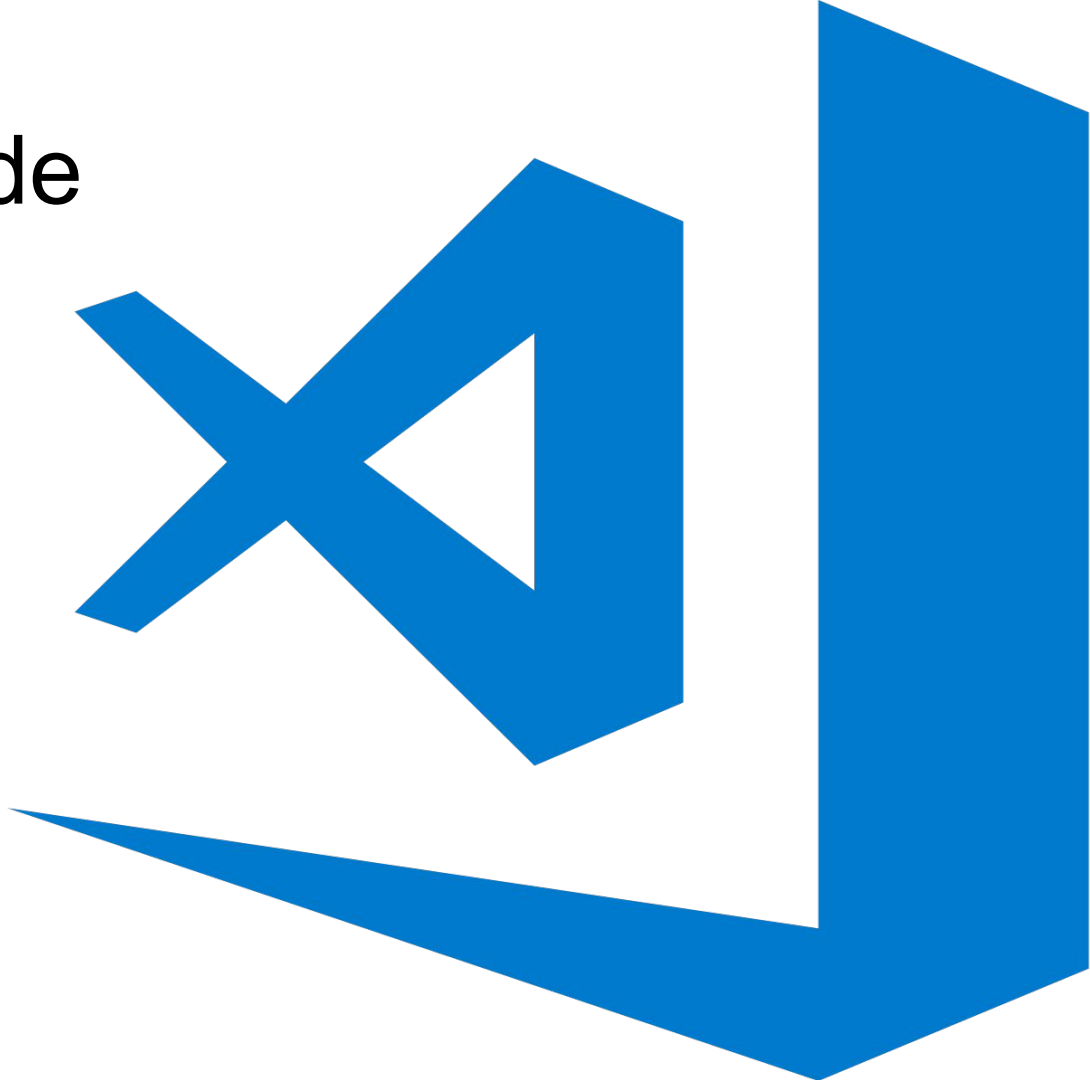
The metadata for a component class associates it with a **template that defines a view**. A template combines ordinary **HTML** with **Angular directives and binding markup** that allow Angular to modify the HTML before rendering it for display.

The metadata for a service class provides the information Angular needs to make it available to components through Dependency Injection (DI).

An **app's components** typically define **many views, arranged hierarchically**. Angular provides the **Router service** to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

# Visual Studio Code

<https://code.visualstudio.com/>



# Installation;

We need to be installed:

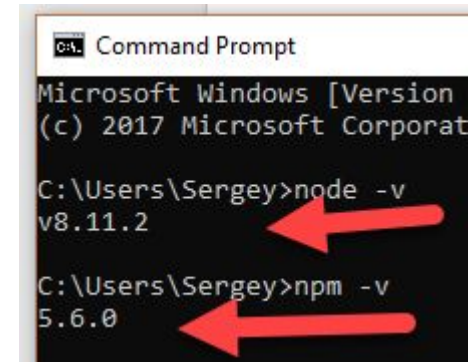
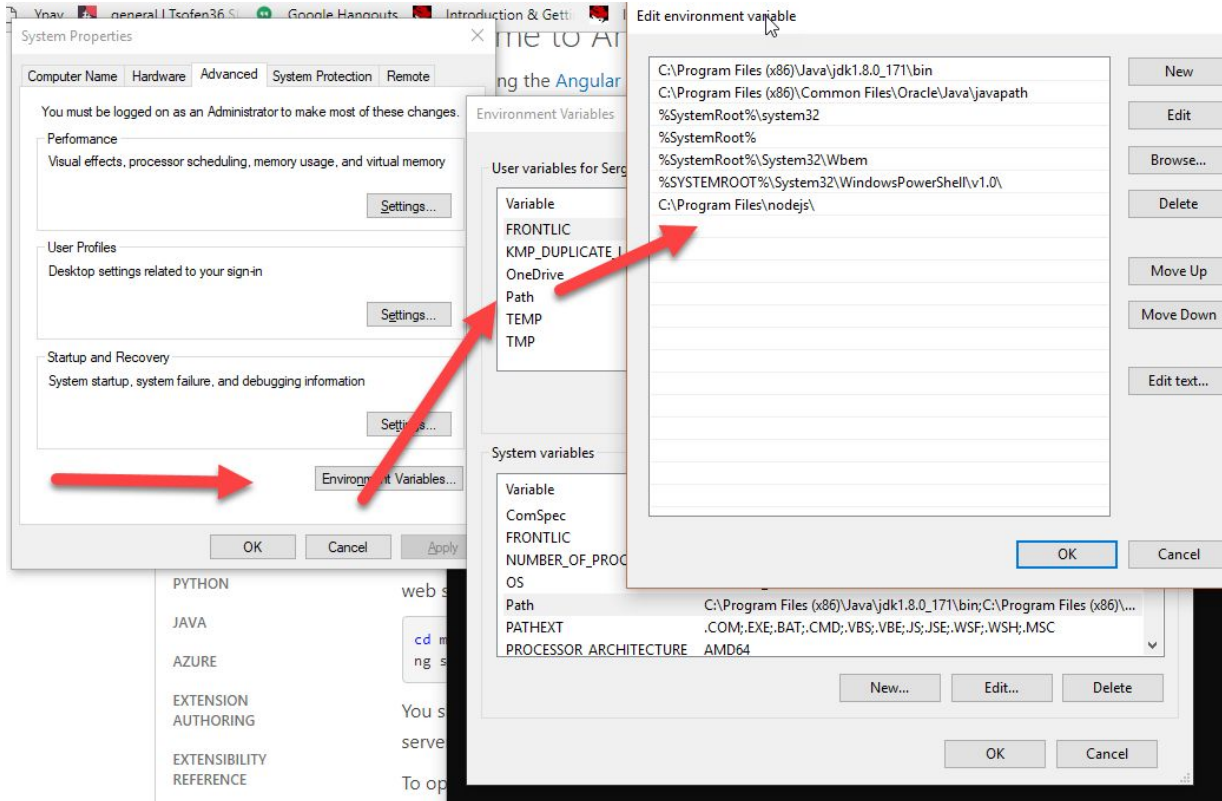
- [Node.js](#) - JavaScript runtime
- npm - (the Node.js package manager)  
npm for JavaScript, just like Maven is the most popular build and dependency resolution tool for Java
- Install Angular CLI - is a command line tool for creating angular apps (slide13)

npm is included with Node.js which you can install from [here](#).

# What is Node.js?

- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server

# After node.js (&npm) installation:



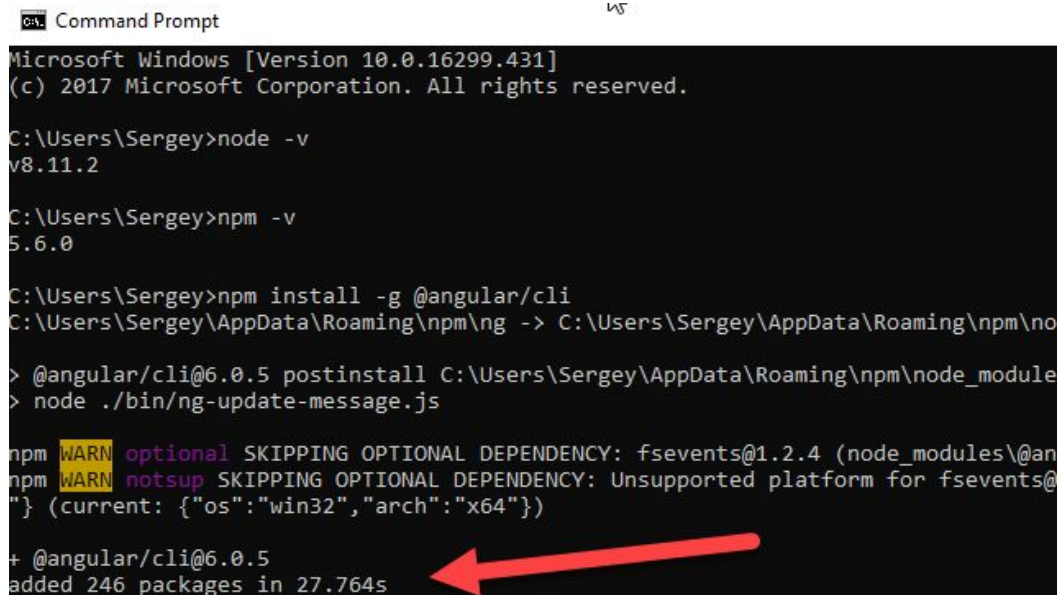
# Install Angular CLI

Run the following command:

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

More info:

<https://github.com/angular/angular-cli>



```
Command Prompt
Microsoft Windows [Version 10.0.16299.431]
(c) 2017 Microsoft Corporation. All rights reserved.

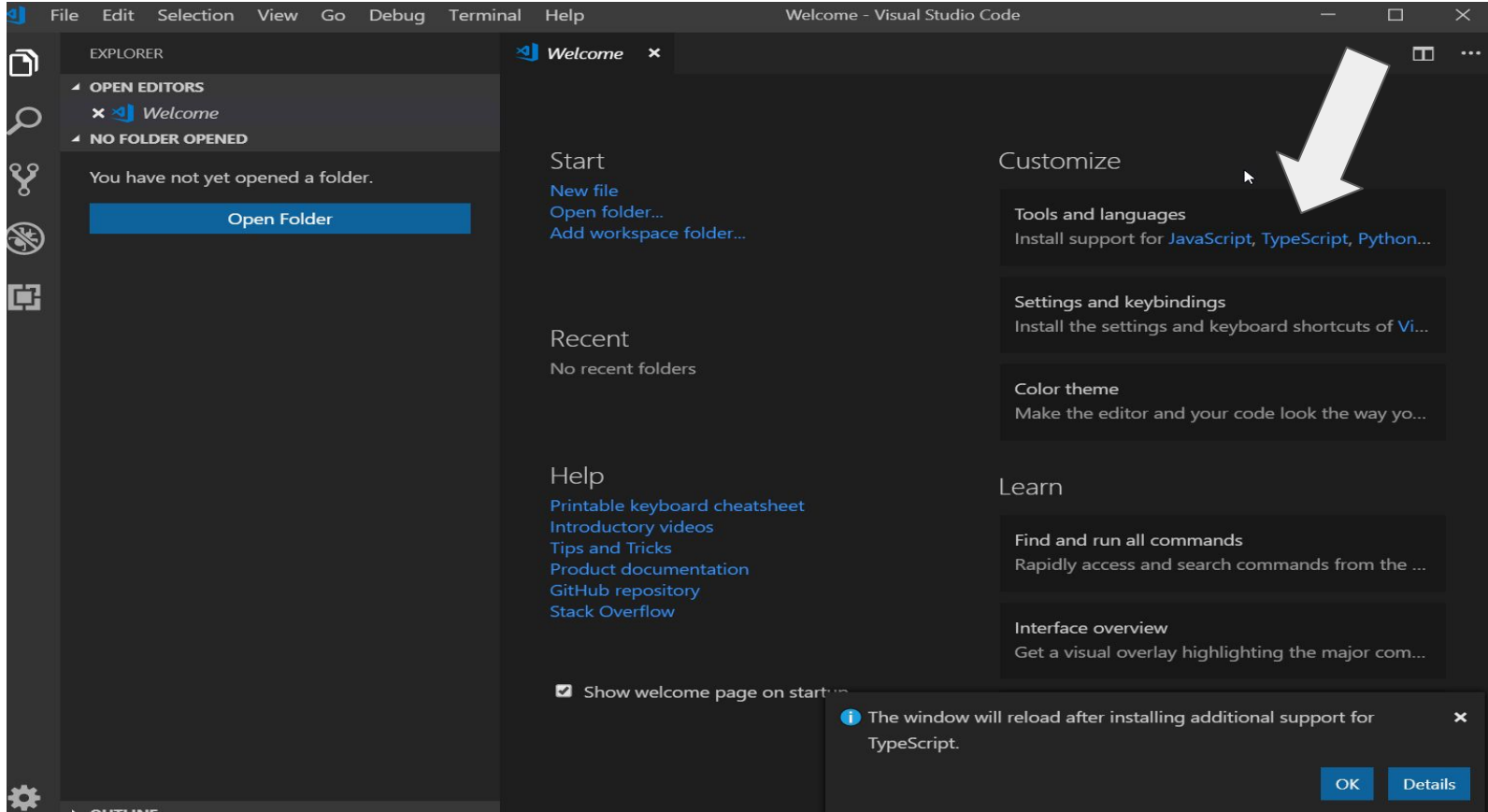
C:\Users\Sergey>node -v
v8.11.2

C:\Users\Sergey>npm -v
5.6.0

C:\Users\Sergey>npm install -g @angular/cli
C:\Users\Sergey\AppData\Roaming\npm\ng -> C:\Users\Sergey\AppData\Roaming\npm\node_modules\@angular\cli@6.0.5 postinstall C:\Users\Sergey\AppData\Roaming\npm\node_modules\@angular\cli\node_modules\npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.4 (node_modules\@angular\cli\node_modules\npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.4) (current: {"os":"win32","arch":"x64"})

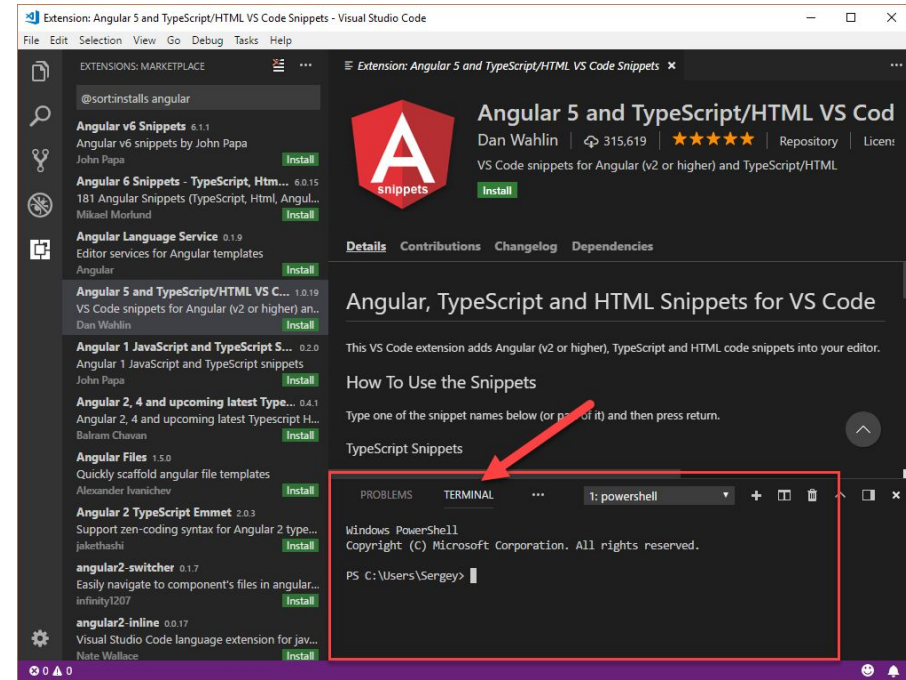
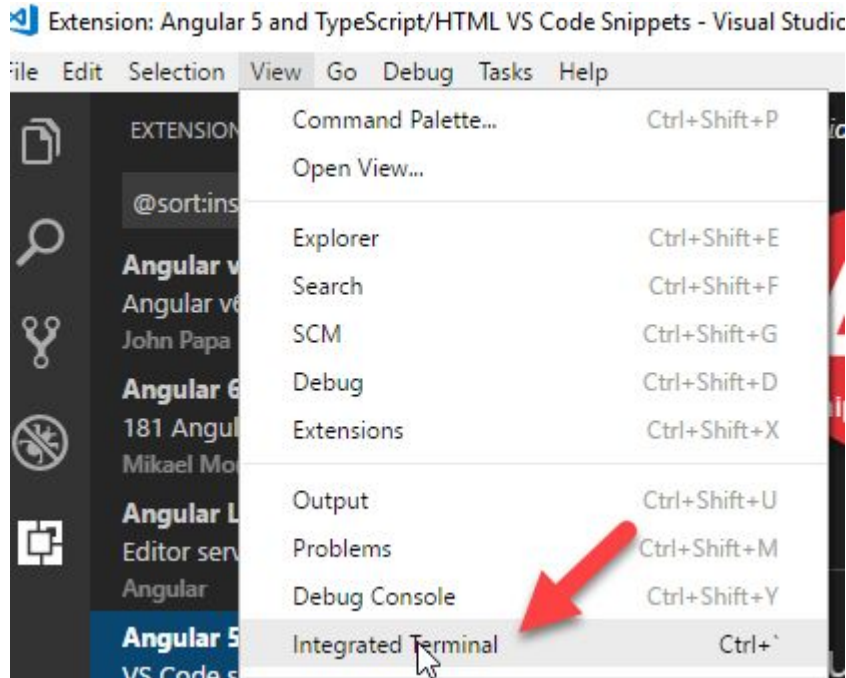
+ @angular/cli@6.0.5
added 246 packages in 27.764s
```

# open VS Code



# Create Your First Angular App

open VS Code and navigate to  
View >> Integrated Terminal





# Generating an Angular project

*'ng' is just an abbreviation of word 'angular'*

These commands will create a directory with name " *AngularDemo*" and then create an Angular application with the name "*myFirstApp*" inside that directory.

**mkdir AngularDemo**

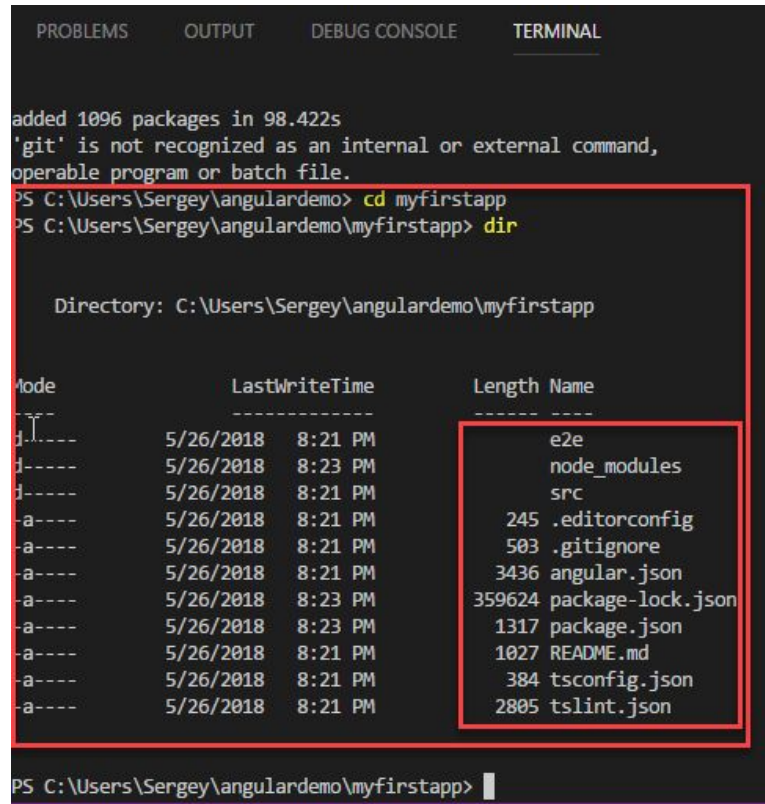
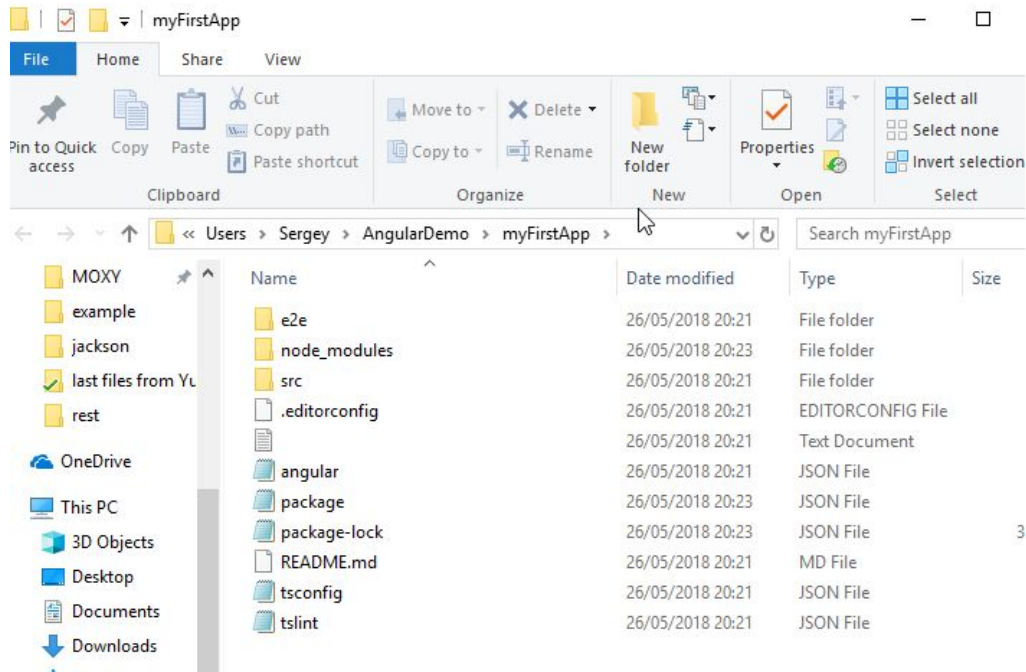
**cd AngularDemo**

**ng new myFirstApp**

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\Sergey> cd angulardemo
PS C:\Users\Sergey\angulardemo> ng new myFirstApp
CREATE myFirstApp/angular.json (3436 bytes)
CREATE myFirstApp/package.json (1316 bytes)
CREATE myFirstApp/README.md (1027 bytes)
CREATE myFirstApp/tsconfig.json (384 bytes)
CREATE myFirstApp/tslint.json (2805 bytes)
CREATE myFirstApp/.editorconfig (245 bytes)
CREATE myFirstApp/.gitignore (503 bytes)
CREATE myFirstApp/src/environments/environment.prod.ts (51 bytes)
CREATE myFirstApp/src/environments/environment.ts (631 bytes)
CREATE myFirstApp/src/favicon.ico (5430 bytes)
CREATE myFirstApp/src/index.html (297 bytes)
CREATE myFirstApp/src/main.ts (270 bytes)
```

# That's it. We have created our first Angular app using VS Code and Angular CLI



src

app

app.component.css

app.component.html

app.component.spec.ts

app.component.ts

app.module.ts

assets

.gitkeep

app/app.component.  
{ts,html,css,spec.ts}

Defines the `AppComponent` along with an HTML template, CSS stylesheet, and a unit test. It is the root component of what will become a tree of nested components as the application evolves.

app/app.module.ts

Defines `AppModule`, the [root module](#) that tells Angular how to assemble the application. Right now it declares only the `AppComponent`. Soon there will be more components to declare.

assets/\*

A folder where you can put images and anything else to be copied wholesale when you build your application.

# Serving an Angular project via a development server

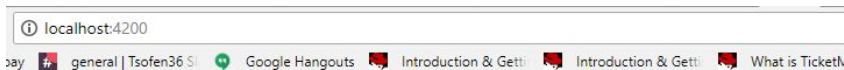
```
cd AngularDemo
```

```
cd myFirstApp
```

```
ng serve --open
```

open any browser on your machine and navigate to URL:

<http://localhost:4200>



Welcome to app!



ome links to help you start:

[f Heroes](#)

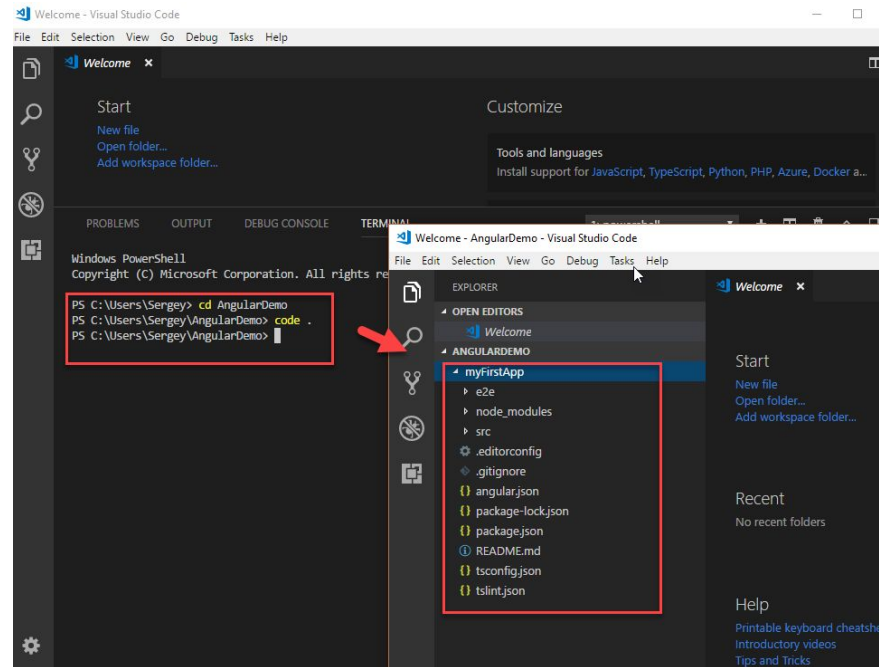
[ocumentation](#)

# Look into the code:

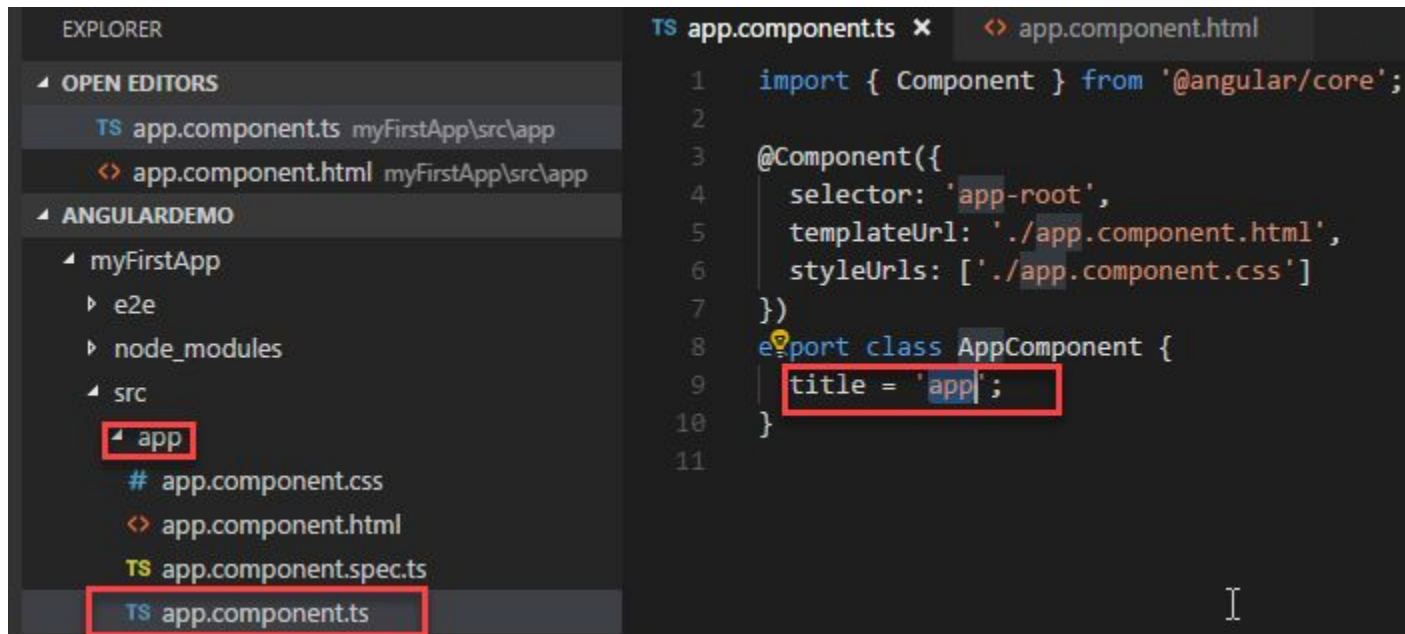
Open another VS Code window by navigating to File >> New Window and run the following set of commands in the terminal.

```
cd AngularDemo
```

```
code .
```



# Updating The App:



The image shows a screenshot of the Visual Studio Code interface. On the left is the Explorer sidebar, and on the right is the Editor view.

**EXPLORER**

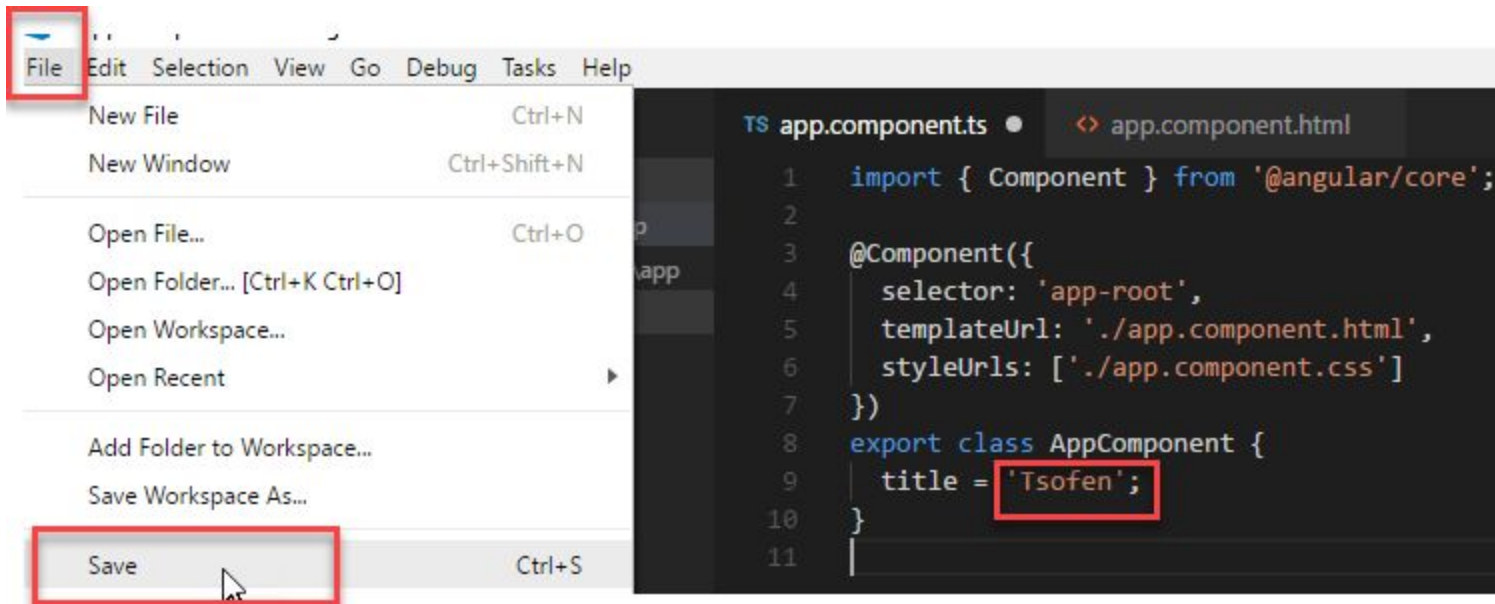
- OPEN EDITORS
  - TS app.component.ts myFirstApp\src\app
  - <> app.component.html myFirstApp\src\app
- ANGULARDEMO
  - myFirstApp
    - e2e
    - node\_modules
    - src
      - app** (highlighted with a red box)
      - # app.component.css
      - <> app.component.html
      - TS app.component.spec.ts
      - TS app.component.ts** (highlighted with a red box)

**TS app.component.ts**

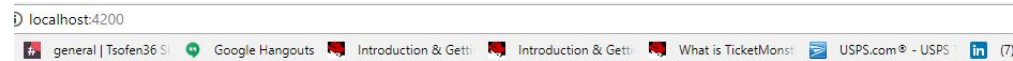
```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'app';
10 }
11
```

The Editor view shows the `app.component.ts` file. The `title = 'app';` line is highlighted with a red box. The `app.component.html` file is also open in the Editor view.





You can see the updated web page as shown below



**Welcome to Tsofen**



Open src/app/app.component.css and give the component some style

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

**Welcome to Tsofen!**

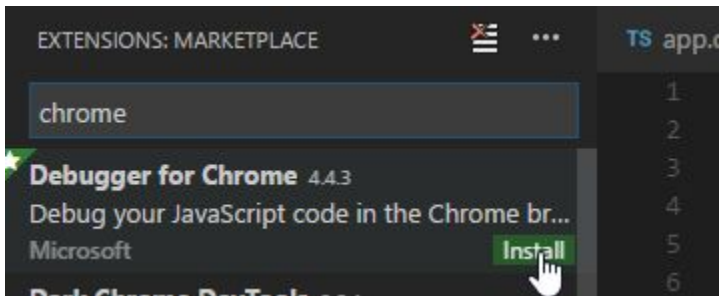




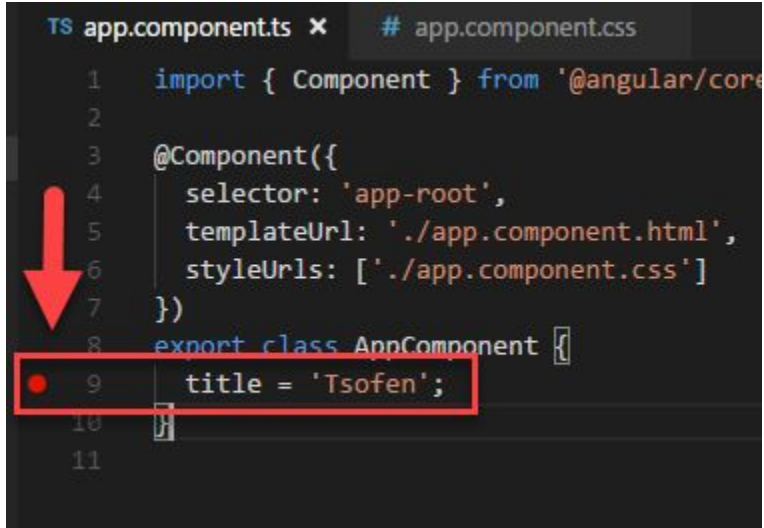
# Debugging Angular

Open the Extensions view (**Ctrl+Shift+X**) and type 'chrome' in the search box. You'll see several extensions which reference Chrome.

Press the Install button for Debugger for Chrome. The button will change to Installing then, after completing the installation, it will change to Reload. **Press Reload** to restart VS Code and activate the extension.



# Set a breakpoint



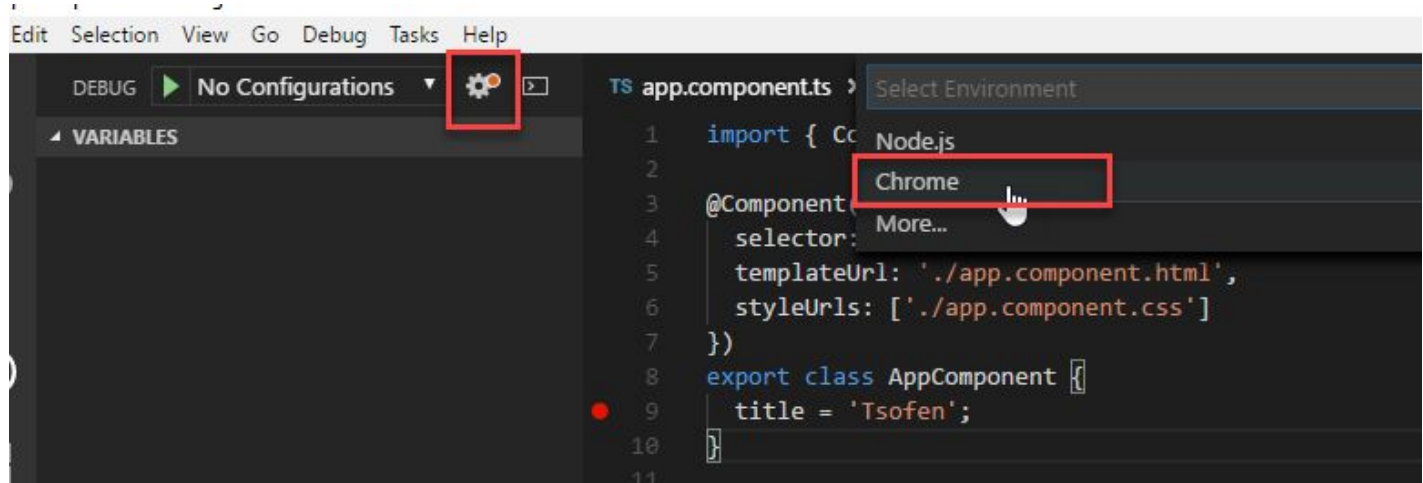
The image shows a code editor with two tabs: 'TS app.component.ts' and '# app.component.css'. The 'TS app.component.ts' tab is active, displaying the following code:

```
1 import { Component } from '@angular/core'
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'Tsofen';
10 }
11
```

A red arrow points to line 9, and a red box highlights the line `title = 'Tsofen';`, indicating that a breakpoint has been set at this location.

# Configure the Chrome debugger

1. Go to the Debug view (Ctrl+Shift+D)
2. Click on gear button to create a launch.json debugger configuration file
3. Choose Chrome from the Select Environment dropdown
4. This will create a launch.json file in a new .vscode folder in your project which includes a configuration to launch the website



We need to make one change for our example: change the port of the url from 8080 to 4200.

Press F5 or the green arrow to launch the debugger and open a new browser instance.

Refresh the page and you should hit your breakpoint.

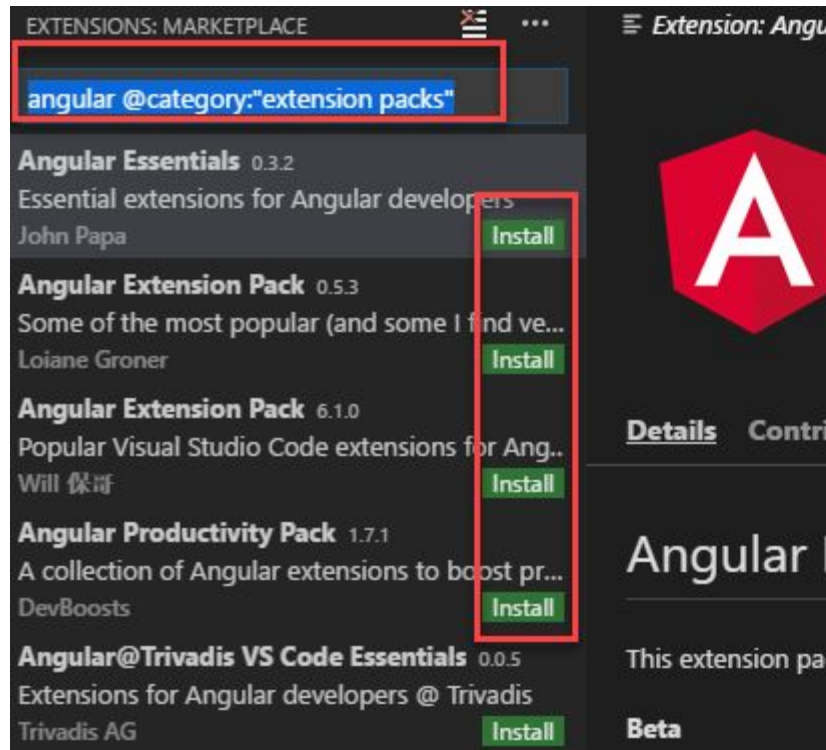
Your launch.json should look like this:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "chrome",
      "request": "launch",
      "name": "Launch Chrome against localhost",
      "url": "http://localhost:4200",
      "webRoot": "${workspaceFolder}"
    }
  ]
}
```

# Angular Extensions

In addition to the functionality VS Code provides out of the box, you can install VS Code extensions for greater functionality.

```
angular @category:"extension packs"
```



## Tour of Heroes

[Dashboard](#)[Heroes](#)

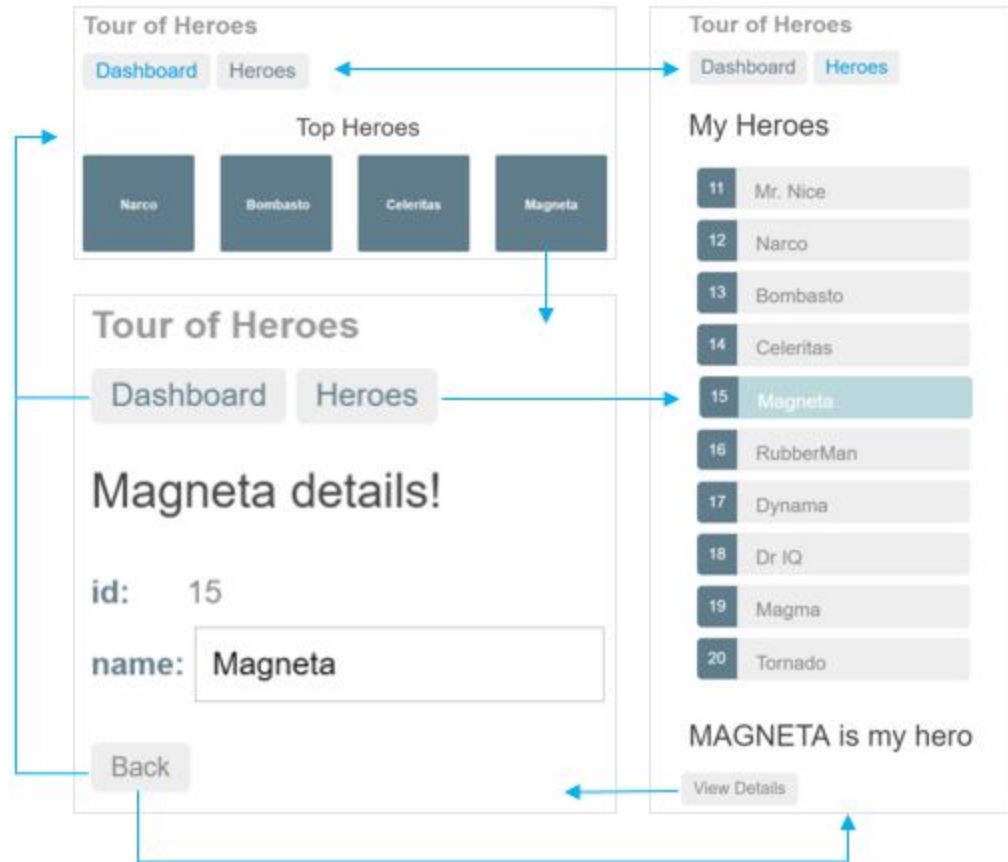
### Top Heroes

Narco

Bombasto

Celeritas

Magneta



# Create a new application

Create a new project named angular-tour-of-heroes with this CLI command.

```
ng new angular-tour-of-heroes
```



# Serve the application

Go to the project directory and launch the application.

```
cd angular-tour-of-heroes  
ng serve --open
```

You should see the app running in your browser.



ome links to help you start:

[f Heroes](#)

[ocumentation](#)

# Angular components

The page you see is the application shell. The shell is controlled by an Angular component named AppComponent.

Components are the fundamental building blocks of Angular applications. They display data on the screen, listen for user input, and take action based on that input.

# Angular components

Components are the fundamental building blocks of Angular applications. They display data on the screen, listen for user input, and take action based on that input.

Any Angular app always has a top-level component (AppComponent), which may include child components, which in turn may include their children.

The top-level component with the red border encompasses multiple child components.

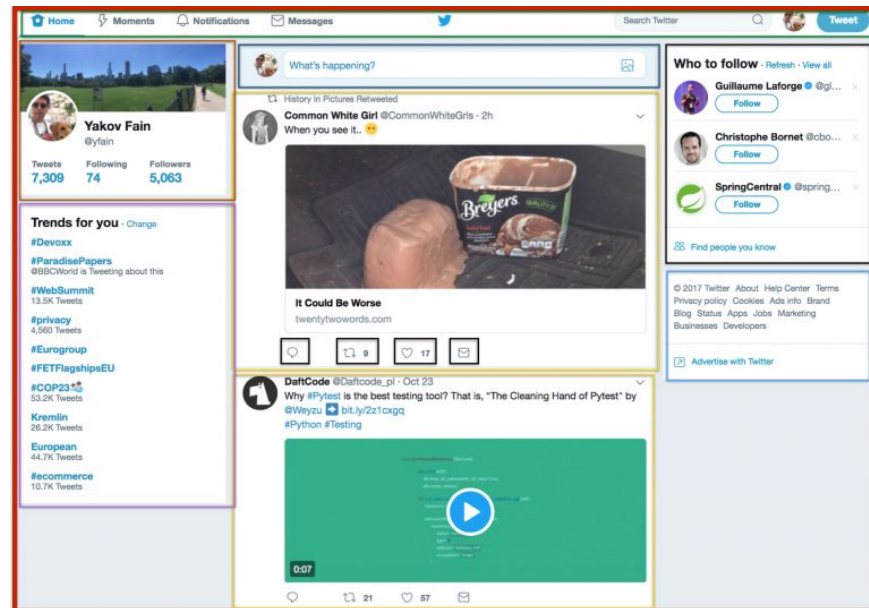
In the middle, you can see a New Tweet component on top and two instances of the Tweet component, which in turn has child components for reply, retweet, like, and direct messaging.

A parent component can pass the data to its child by binding the values to the child's component property.

A child component has no knowledge of where the data came from.

A child component can pass the data to its parent (without knowing who the parent is) by emitting events.

This architecture makes components self-contained and reusable.



# Clean separation between UI and business logic

A component is a  
TypeScript class annotated  
with a decorator  
`@Component`.

At decorator `@Component`  
you specify the component's  
HTML and CSS.

You can specify them either  
inline or in separate files.

```
1  @Component({
2    selector: 'home',
3    template: `<div class="home">Home Component
4      <input type="text"/> </div>`,
5    styles: [`.home {background: red; padding: 15px 0 0 30px;
6      height: 80px; width:70%;
7      font-size: 30px; float:left;}`}]]
8  export class HomeComponent {
9    // Implement business logic here
10 }
```

As your template or styles grow in size, you can keep them in separate files:

```
1  @Component({
2    selector: 'home',
3    templateUrl: './home.component.html',
4    styleUrls: ['./home.component.css']
5  export class HomeComponent {
6  }
```

# AppComponent files:

Navigate to the src/app folder.

You'll find the implementation of the shell AppComponent 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.

# Update AppComponent

**component class file (app.component.ts)**

Open the component class file (app.component.ts)

Change the value of (app.component.ts) class title **property** to 'Tour of Heroes'.

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

## app.component.html (template)

Open the component template file (app.component.html) and delete the default template generated by the Angular CLI.

Replace it with the following line of HTML.

```
<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.

The browser refreshes and displays the new application title.

# Add application styles

**app.component.css**— the component's private CSS styles

Most apps strive for a consistent look across the application.

The CLI generated an empty `styles.css` for this purpose. Put your application-wide styles there.

Here's an excerpt from the `styles.css` for the Tour of Heroes sample app.

`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[text], button {  
    color: #888;
```

# Summary

- You created the initial application structure using the Angular CLI.
- You learned that Angular components display data.
- You used the double curly braces of interpolation to display the app title.

# Create a new component - heroes component

The new heroes component to display hero information and place that component in the application shell.

Using the Angular CLI, generate a new component named **heroes**.

```
ng generate component heroes
```

The CLI creates a new folder, `src/app/heroes/` and generates the three files of the `HeroesComponent`.

# Add a hero property

Add a **hero property** to the HeroesComponent

Name a hero "Windstorm."

heroes.component.ts (hero property)

```
hero = 'Windstorm';
```

# Show the hero

Open the `heroes.component.html` template file.

Delete the default text generated by the Angular CLI and replace it with a data binding to the new `hero` property.

`heroes.component.html`

```
{{hero}}
```

# Show the HeroesComponent view

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>
```

.

# Create a Hero class

A real hero is more than a name.

Create a Hero class in its own file in the src/app folder. Give it id and name properties.

**src/app/hero.ts**

```
export class Hero {  
  id: number;  
  name: string;  
  constructor(id: number, name: string) { this.name = name;  
this.id = id;  
}  
}
```

## Update HeroesComponent with Hero object

Return to  
src/app/heroes/heroes.component.ts

import the Hero class.

change hero property to be of type  
Hero

Initialize it with an new hero object.

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

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

  // hero = 'windstorm';
  hero: Hero;
  constructor() { }

  ngOnInit() {
    this.hero = new Hero(1, 'my hero');
  }

}
```



# Show the hero object

Update the binding in the template to show both id and name

heroes.component.html (HeroesComponent's template)

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

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

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

The browser refreshes and displays the hero's information.

Format with the UppercasePipe

The browser refreshes and now the hero's name is displayed in capital letters.

The word uppercase in the interpolation binding, right after the pipe operator ( | ), activates the built-in UppercasePipe.

Pipes are a good way to format strings, currency amounts, dates and other display data. Angular ships with several built-in pipes and you can create your own.

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

# Edit the hero (Two-way binding)

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 flow from the component class out to the screen and from the screen back to the class.

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.

Here it binds the `hero.name` property to the HTML textbox so that data can flow in both directions: from the `hero.name` property to the textbox, and from the textbox back to the `hero.name`.

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

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

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

# The missing FormsModule

Notice that the app stopped working when you added [(ngModel)].

*Template parse errors:*

*Can't bind to 'ngModel' since it isn't a known property of 'input'.*

Although ngModel is a valid **Angular directive**, it isn't available by default.

It belongs to the optional FormsModule

## AppModule

Angular needs to know how the pieces of your application fit together and what other files and libraries the app requires. This information is called metadata

The most important @NgModule decorator annotates the top-level AppModule class.

The Angular CLI generated an AppModule class in src/app/app.module.ts when it created the project.

```
app.module.ts (FormsModule symbol import)
import { FormsModule } from
  '@angular/forms';
```

```
app.module.ts ( @NgModule imports)
```

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

# Summary

You used the CLI to create a second HeroesComponent.

You displayed the HeroesComponent by adding it to the AppComponent shell.

You applied the UppercasePipe to format the name.

You used two-way data binding with the ngModel directive.

You learned about the AppModule.

You imported the FormsModule in the AppModule so that Angular would recognize and apply the ngModel directive.

You learned the importance of declaring components in the AppModule and appreciated that the CLI declared it for you.

# Display a Heroes List

You'll expand the Tour of Heroes app to display a list of heroes, and allow users to select a hero and display the hero's details.

## Create mock heroes

You'll need some heroes to display.

Eventually you'll get them from a remote data server.

For now, you'll create some *mock heroes* and pretend they came from the server.

Create a file called `mock-heroes.ts` in the `src/app/` folder.

Define a `HEROES` constant as an array of ten heroes and export it. The file should look like this.

```
src/app/mock-heroes.ts
import { Hero } from './hero';

export const HEROES: Hero[] = [
  { id: 11, name: 'Mr. 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' }
];
```

# Update HeroesComponent with Heros

Open the `HeroesComponent` class file and import the mock `HEROES`.

`src/app/heroes/heroes.component.ts`

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

Add a `heroes` property to the class that exposes these heroes for binding.

```
heroes = HEROES;
```

# List heroes with \*ngFor

The \*ngFor is Angular's **repeater directive**.

It repeats the host element for each element in a list.

<li> is the host element

**heroes** is the list from the HeroesComponent class.

**hero** holds the current hero object for each iteration through the list.

Don't forget the asterisk (\*) in front of ngFor. It's a critical part of the syntax.

heroes.component.html (heroes template)

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

# Style the heroes

In the first tutorial, you set the basic styles for the entire application in `styles.css`.

That stylesheet didn't include styles for this list of heroes.

You could add more styles to `styles.css` and keep growing that stylesheet as you add components.

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.

This approach makes it easier to re-use the component somewhere else and deliver the component's intended appearance even if the global styles are different.

```
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.selected:hover {
  background-color: rgb(220, 187, 206) !important;
  color: white;
}
.heroes li:hover {
  color: #607D8B;
  background-color: #DDD;
  left: .1em;
}
.heroes .text {
  position: relative;
  top: -3px;
}
.heroes .badge {
  display: inline-block;
  font-size: small;
  color: white;
  padding: 0.8em 0.7em 0 0.7em;
  background-color: #607D8B;
  line-height: 1em;
```



# app\app.component.css

```
/* 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: 20px;
```

# src\styles.css

```
/* You can add global styles to this file, and also import other style files */
```

```
/* 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 {
```

# Add a click event binding

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.

In this section, you'll listen for the hero item click event and update the hero detail.

Add a click event binding to the `<li>` like this:

heroes.component.html

```
<h2>My Heroes</h2>
<ul class="heroes">
  <li *ngFor="let hero of heroes"
    (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span>
    {{hero.name}}
  </li>
</ul>
```

<https://angular.io/guide/template-syntax#event-binding>

# Add the click event handler

Rename the component's hero property to selectedHero but don't assign it.

There is no selected hero when the application starts.

Add the following onSelect() method, which assigns the clicked hero from the template to the component's selectedHero.

```
src/app/heroes/heroes.component.ts
```

```
selectedHero: Hero;
```

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

# Update the details template

The template still refers to the component's old hero property which no longer exists. Rename hero to selectedHero.

heroes.component.html

```
<h2>{{ selectedHero.name | uppercase }} Details</h2>
<div><span>id: </span>{{selectedHero.id}}</div>
<div>
  <label>name:
    <input [ (ngModel) ]="selectedHero.name"
      placeholder="name">
  </label>
</div>
```

**After the browser refreshes, the application is broken.**



**HeroesComponent.html:**

**ERROR TypeError: Cannot read property 'name' of undefined**

What happened?

When the app starts, the selectedHero is undefined by design.

Binding expressions in the template that refer to properties of selectedHero — expressions like `{{selectedHero.name}}` — must fail because there is no selected hero.

Now click one of the list items. The app seems to be working again. The heroes appear in a list and details about the clicked hero appear at the bottom of the page.

# The fix

The component should only display the selected hero details if the selectedHero exists.

Wrap the hero detail HTML in a `<div>`.

Add Angular's `*ngIf` directive to the `<div>` and set it to `selectedHero`.

Don't forget the asterisk (\*) in front of `ngIf`. It's a critical part of the syntax.

`src/app/heroes/heroes.component.html (*ngIf)`

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

    <h2>{{ selectedHero.name | uppercase }}
    Details
</h2>
    <div><span>id:
</span>{{selectedHero.id}}</div>
    <div>
        <label>name:
            <input
[ (ngModel) ]="selectedHero.name"
placeholder="name">
        </label>
    </div>

</div>
```

# Style the selected hero

Add the following `[class.selected]` binding to the `<li>` in the HeroesComponent template:

Heroes.component.html

```
<li *ngFor="let hero of heroes"
    [class.selected]="hero === selectedHero"
    (click)="onSelect(hero)">
  <span class="badge">{{hero.id}}</span> {{hero.name}}
</li>
```

14	Celeritas
15	Magneta
16	RubberMan

When the current row hero is the same as the selectedHero, Angular adds the selected CSS class. When the two heroes are different, Angular removes the class.



# Summary

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.

# Create the HeroDetailComponent

# Master/Detail Components

At the moment, the `HeroesComponent` displays both the list of heroes and the selected hero's details.

Keeping all features in one component as the application grows will not be maintainable.

You'll want to split up large components into smaller sub-components, each focused on a specific task or workflow.

In this page, you'll move the hero details into a separate, reusable `HeroDetailComponent`.

The `HeroesComponent` will only present the list of heroes.

The `HeroDetailComponent` will present details of a selected hero.

# Create the HeroDetailComponent

Use the Angular CLI to generate a new component named hero-detail.

```
ng generate component hero-detail
```

*This command scaffolds the HeroDetailComponent files and declares the component in AppModule.*

# Write the template

Cut the HTML for the hero detail from the bottom of the HeroesComponent template and paste it in the HeroDetailComponent template.

The pasted HTML refers to a selectedHero.

The new HeroDetailComponent can present any hero, not just a selected hero.

So replace "selectedHero" with "hero" everywhere in the template.

src/app/hero-detail/**hero-detail.component.html**

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

  <h2>{{ hero.name | uppercase }} Details</h2>
  <div><span>id: </span>{{hero.id}}</div>
  <div>
    <label>name:
      <input [(ngModel)]=" hero.name"
placeholder="name"/>
    </label>
  </div>

</div>
```

# Add the @Input() hero property

The HeroDetailComponent template binds to the component's hero property which is of type Hero.

Open the HeroDetailComponent class file and import the Hero symbol.

The hero property must be an Input property, annotated with the @Input() decorator, because the external HeroesComponent will bind to it like this.

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

```
</app-hero-detail>
```

This component simply receives a hero object through its hero property and displays it.

```
src/app/hero-detail/hero-detail.component.ts

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

@Input() hero: Hero;
```

## Show the HeroDetailComponent

Now HeroesComponent delegates to the HeroDetailComponent.

The two components have a parent/child relationship.

The parent HeroesComponent will control the child HeroDetailComponent by sending it a new hero to display whenever the user selects a hero from the list.

You won't change the HeroesComponent class but you will change its template.

The HeroDetailComponent selector is 'app-hero-detail'.

Add an <app-hero-detail> element near the bottom of the HeroesComponent template

heroes.component.html (HeroDetail binding)

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

```
<ul class="heroes">
  <li *ngFor="let hero of heroes"
      [class.selected]="hero === selectedHero"
      (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span>
    {{hero.name}}
  </li>
</ul>
```

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

[hero]="selectedHero" is an Angular property binding.

Now when the user clicks a hero in the list, the selectedHero changes. When the selectedHero changes, the property binding updates hero and the HeroDetailComponent displays the new hero.

# Summary

You created a separate, reusable HeroDetailComponent.

You used a property binding to give the parent HeroesComponent control over the child HeroDetailComponent.

You used the @Input decorator to make the hero property available for binding by the external HeroesComponent.



# Create the Hero Service

# Services

The Tour of Heroes HeroesComponent is currently getting and displaying fake data.

Components **shouldn't** fetch or save data directly.

They **should** focus on **presenting data** and **delegate data access to a service**.

You'll create a HeroService that all application classes can use to get heroes.

Instead of creating that service with new, you'll rely on Angular **dependency injection** to inject it into the HeroesComponent constructor.

# Dependency Injection -DI

Any application is composed of many objects that collaborate with each other to perform some useful stuff.

Traditionally each object is responsible for obtaining its own references to the dependent objects (dependencies) it collaborate with. This leads to highly coupled classes and hard-to-test code.

For example, consider a Car object.

A Car depends on wheels, engine, fuel, battery, etc. to run. Traditionally we define the brand of such dependent objects along with the definition of the Car object.

## Without Dependency Injection (DI):

```
class Car{  
    private Wheel wh= new NepaliRubberWheel();  
    private Battery bt= new ExcideBattery();  
  
    //The rest  
}
```

Here, the Car object is responsible for creating the dependent objects.

What if we want to change the type of its dependent object - say Wheel - after the initial NepaliRubberWheel() punctures? We need to recreate the Car object with its new dependency say ChineseRubberWheel(), but only the Car manufacturer can do that.

## Then what does the Dependency Injection do us for...?

When using dependency injection, **objects are given their dependencies at run time** rather than compile time (car manufacturing time).

So that we can now change the Wheel whenever we want.

# After using dependency injection:

Here, we are injecting the dependencies (Wheel and Battery) at runtime.

```
class Car{
    private Wheel wh= [Inject an Instance of Wheel (dependency of car) at runtime]
    private Battery bt= [Inject an Instance of Battery (dependency of car) at
runtime]
    Car(Wheel wh,Battery bt) {
        this.wh = wh;
        this.bt = bt;
    }
}
```

# Create the HeroService

Using the Angular CLI, create a service called hero.

```
ng generate service hero
```

@Injectable() services

Notice that the new service imports the Angular Injectable symbol and annotates the class with the @Injectable() decorator.

This marks the class as one that participates in the dependency injection system.

The HeroService class is going to provide an injectable service, and it can also have its own injected dependencies. It doesn't have any dependencies yet, but it will soon.

src/app/hero.service.ts (new service)

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

@Injectable({
  providedIn: 'root',
})
export class HeroService {

  constructor() { }

}
```

When you provide the service at the root level, Angular creates a single, shared instance of HeroService and injects into any class that asks for it.

# Get hero data

The HeroService could 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.

The implementation in this tutorial will continue to deliver mock heroes.

## Import the Hero and HEROES.

src/app/hero.service.ts (new service)

```
import { Injectable } from '@angular/core';  
import { Hero } from '../hero';  
import { HEROES } from '../mock-heroes';
```

```
@Injectable({  
  providedIn: 'root',  
})  
export class HeroService {  
  
  constructor() { }  
  
  getHeroes(): Hero[] {  
    return HEROES;  
  }  
  
}
```

# Update HeroesComponent

0. Open the HeroesComponent class file.

src/app/heroes/heroes.component.ts

1. Delete the HEROES import, because you won't need that anymore.

Import the HeroService instead.

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

2. Replace the definition of the heroes property with a simple declaration.

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

3. Inject the HeroService

Add a private heroService parameter of type HeroService to the constructor.

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

The parameter simultaneously defines a **private heroService property** and identifies it as a HeroService injection.

When Angular creates a HeroesComponent, the Dependency Injection system sets the heroService parameter to the singleton instance of HeroService.

```
import { Component, OnInit } from '@angular/core';  
import { Hero } from '../hero';  
// 1 Delete the HEROES import, because you won't need that  
anymore.  
//Import the HeroService instead.  
// import { HEROES } from '../mock-heroes';  
import { HeroService } from '../hero.service';
```

```
@Component({  
  selector: 'app-heroes',  
  templateUrl: './heroes.component.html',  
  styleUrls: ['./heroes.component.css']  
})  
export class HeroesComponent implements OnInit {  
  selectedHero: Hero; // = new Hero(1, 'Sergey');  
  // 2 Replace the definition of the heroes property with  
  a simple declaration  
  //heroes=HEROES;  
heroes: Hero[];
```

```
  // 3 Inject the HeroService  
  //Add a private heroService parameter of type  
  HeroService to the constructor.  
  constructor(private heroService: HeroService) { }
```

```
//4. Add getHeroes()  
  getHeroes(): void {  
    this.heroes = this.heroService.getHeroes();  
  }
```

```
//5. Call getHeroes() in ngOnInit  
  ngOnInit(): void {  
    this.getHeroes();  
  },
```

# Update HeroesComponent cont.

## 4. Add getHeroes()

Create a function to retrieve the heroes from the service.

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

## 5. Call getHeroes() in ngOnInit

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

Reserve the constructor for simple initialization such as wiring constructor parameters to properties. The constructor shouldn't do anything. It certainly shouldn't call a function that makes HTTP requests to a remote server as a real data service would.

Instead, call getHeroes() inside the ngOnInit lifecycle hook and let Angular call ngOnInit at an appropriate time after constructing a HeroesComponent instance.

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

## 6. See it run

After the browser refreshes, the app should run as before, showing a list of heroes and a hero detail view when you click on a hero name.

```
import { Component, OnInit } from '@angular/core';  
import { Hero } from '../hero';  
// 1 Delete the HEROES import, because you won't need that  
anymore.  
// Import the HeroService instead.  
// import { HEROES } from '../mock-heroes';  
import { HeroService } from '../hero.service';
```

```
@Component({  
    selector: 'app-heroes',  
    templateUrl: './heroes.component.html',  
    styleUrls: ['./heroes.component.css']  
})  
export class HeroesComponent implements OnInit {  
    selectedHero: Hero; // = new Hero(1, 'Sergey');  
    // 2 Replace the definition of the heroes property with  
    a simple declaration  
    //heroes=HEROES;  
    heroes: Hero[];  
  
    // 3 Inject the HeroService  
    // Add a private heroService parameter of type  
    HeroService to the constructor.  
    constructor(private heroService: HeroService) { }
```

```
//4. Add getHeroes()  
    getHeroes(): void {  
        this.heroes = this.heroService.getHeroes();  
    }  
//5. Call getHeroes() in ngOnInit  
    ngOnInit(): void {  
        this.getHeroes();  
    },
```



# Tour of Tsofen Heroes

## My Heroes

11	Mr. Nice
12	Narco
13	<b>Bombasto</b>
14	Celeritas
15	Magneta
16	RubberMan
17	Dynama
18	Dr IQ
19	Magma
20	Tornado

## BOMBASTO Details

id: 13

name:

# Reactive programming

Good tutorial: <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>

Reactive programming is programming with asynchronous data.

Typical click events are really an asynchronous event stream, on which you can observe and do some side effects. Reactive is that idea on steroids.

You are able to create data streams of anything, not just from click and hover events.

Anything can be a stream: variables, user inputs, properties, caches, data structures, etc.

For example, imagine your Twitter feed would be a data stream in the same fashion that click events are. You can listen to that stream and react accordingly.

A stream can be used as an input to another one. Even multiple streams can be used as inputs to another stream. You can merge two streams. You can filter a stream to get another one that has only those events you are interested in. You can map data values from one stream to another new one.



# Understanding Observables

An array is a collection of elements, such as [1, 2, 3, 4, 5]. You get all the elements immediately, and you can do things like map, filter and map them. This allows you to transform the collection of elements any way you'd like.

Now suppose that each element in the array occurred over time; that is, you don't get all elements immediately, but rather one at a time. You might get the first element at 1 second, the next at 3 seconds, and so on. Here's how that might be represented:

This can be described as a stream of values, or a sequence of events, or more relevantly, an observable.

An observable is a collection of values over time.

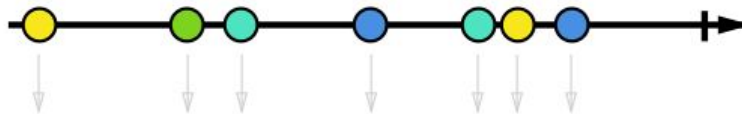
An array of values



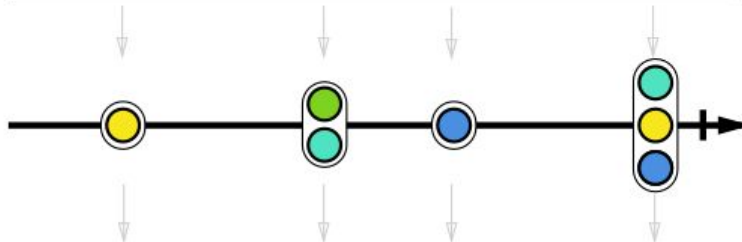
A stream of values



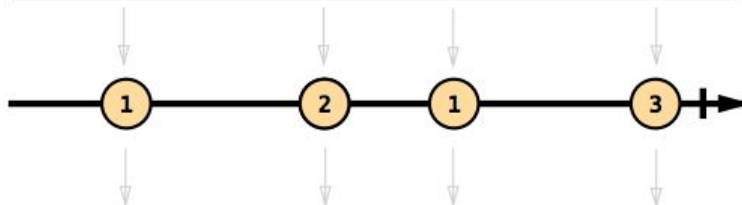
*Click stream*



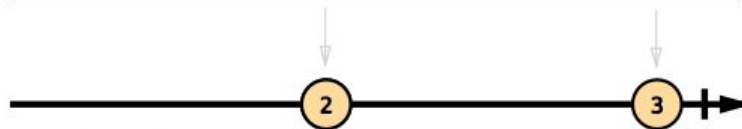
`buffer(clickStream.throttle(250ms))`



`map('get length of list')`



`filter(x >= 2)`

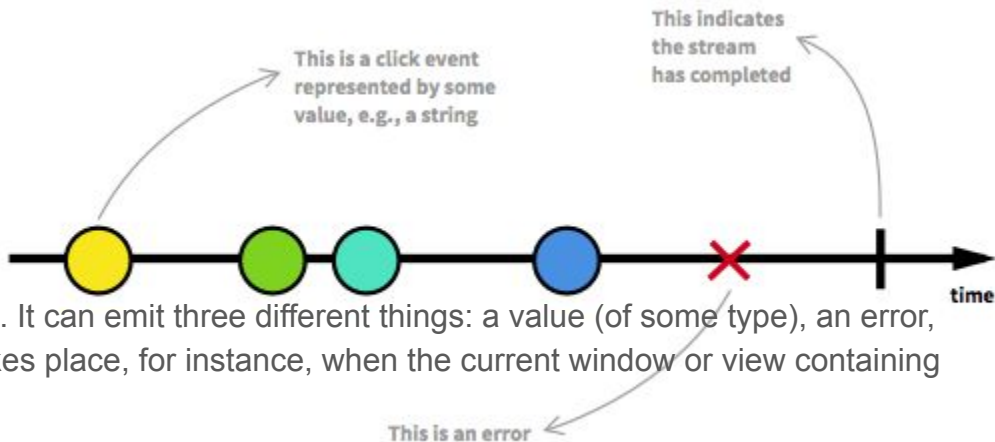


*Multiple clicks stream*

# Click event stream example

starting with our familiar "clicks on a button" event stream.

A stream is a sequence of ongoing events ordered in time. It can emit three different things: a value (of some type), an error, or a "completed" signal. Consider that the "completed" takes place, for instance, when the current window or view containing that button is closed.



We capture these emitted events only asynchronously, by defining a function that will execute when a value is emitted, another function when an error is emitted, and another function when 'completed' is emitted. Sometimes these last two can be omitted and you can just focus on defining the function for values.

The "listening" to the stream is called **subscribing**.

The **functions** we are defining are observers.

The stream is the subject (or "**observable**") being observed.

This is precisely the Observer Design Pattern.

# RxJS

Over the last years, libraries of reactive extensions (RX) became popular in many technologies.

They offer means for handling observable data pushed by some data provider (mouse events, sensors, sockets, UI components et al.)

Such libraries include a rich set of composable operators that can manipulate the data values as they move from the data provider to its subscriber(s).

**Angular comes with the JavaScript library called RxJS**

## **=> arrow function**

`(a, b, c) => { /* ... */ }`

Is (almost) equivalent to:

`function(a, b, c) { /* ... */ }`

## Basic syntax

```
(param1, param2, ..., paramN) => { statements }
```

```
(param1, param2, ..., paramN) => expression
```

```
// equivalent to: => { return expression; }
```

```
// Parentheses are optional when there's only one parameter name:
```

```
(singleParam) => { statements }
```

```
singleParam => { statements }
```

```
// The parameter list for a function with no parameters should be written with a pair of parentheses.
```

```
() => { statements }
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow\\_functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)



# RxJS Example

Imagine that the user enters the value of the stock symbol in the input control `searchInput`.

The following code fragment:

- subscribes to the observable data stream `valueChanges` produced by the form control
- and makes a request to a function to get the price of the selected stock.

To minimize the number of requests to the server, we apply the `debounceTime` operator so the function `getStockQuoteFromServer()` will be invoked only if the value in the input control doesn't change during 500 milliseconds.

In this case, I just used one operator `debounceTime` between the data provider and subscriber, but I could have used any number of operators (e.g. `map`, `filter` et al) there.

```
1 | this.searchInput.valueChanges
2 |   .debounceTime(500)
3 |   .subscribe(stock => this.getStockQuoteFromServer(stock));
4 | }
```

# of

signature:

```
of(...values, scheduler: Scheduler): Observable
```

Emit variable amount of values in a sequence.

Example: Emitting a sequence of numbers

```
//emits any number of provided values in sequence
```

```
const source = Rx.Observable.of(1, 2, 3, 4, 5);
```

```
//output: 1,2,3,4,5
```

<http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#static-method-of>

# Observable HeroService

Observable is one of the key classes in the RxJS library.

In a later tutorial on HTTP, you'll learn that Angular's HttpClient methods return RxJS Observables.

In this tutorial, you'll simulate getting data from the server with the RxJS of() function.

7. Open the HeroService file and import the Observable and of symbols from RxJS.

src/app/hero.service.ts

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

8. Replace the getHeroes method with this one.

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

of(HEROES) returns an Observable<Hero[]> that emits a single value, the array of mock heroes.

```
import { Injectable } from '@angular/core';  
import { Hero } from '../hero';  
import { HEROES } from '../mock-heroes';
```

```
//7 import the Observable and of symbols from RxJS
```

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

```
@Injectable({  
  providedIn: 'root'  
})  
export class HeroService {
```

```
  constructor() { }
```

```
//8 Replace the getHeroes method with  
Observable<Hero[]>
```

```
getHeroes(): Observable<Hero[]> {  
  return of(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**.

Find the getHeroes method and replace it with the following code (shown side-by-side with the previous version for comparison).

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

**Observable/Subject/Stream**

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

listen

**Observer/Function**

Observer listen to Observable= **Observer(Function)** executes when **.getHeroes()** emits **heroes array**

## Observable/Subject/Stream

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

listen

## Observer/Function

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

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

The previous version assigns an array of heroes to the component's heroes property. The assignment occurs synchronously, as if the server could return heroes instantly or the browser could freeze the UI while it waited for the server's response.

That won't work when the HeroService is actually making requests of a remote server.

The new version waits for the Observable to emit the array of heroes—which could happen now or several minutes from now. Then subscribe 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.

# Create the Messages Component

# Show messages

In this section you will:

1. add a MessagesComponent that displays app messages at the bottom of the screen
2. create an injectable, app-wide MessageService for sending messages to be displayed
3. inject MessageService into the HeroService
4. display a message when HeroService fetches heroes successfully.

# Create MessagesComponent

Use the CLI to create the MessagesComponent.

```
ng generate component messages
```

The CLI creates the component files in the `src/app/messages` folder and declare MessagesComponent in AppModule.

Modify the AppComponent template to display the generated MessagesComponent

You should see the default paragraph from MessagesComponent at the bottom of the page.

```
/src/app/app.component.html
```

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

```
<app-heroes></app-heroes>
```

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

## Tour of Tsofen Heroes

My Heroes

11	Mr. Nice
12	Narco
13	Bombasto
14	Celeritas
15	Magneta
16	RubberMan
17	Dynama
18	Dr IQ
19	Magma
20	Tornado

MR. NICE Details

id: 11  
name: Mr. Nice  
messages works!





# Create the MessageService

Use the CLI to create the MessageService in src/app.

**ng generate service message**

Open MessageService and replace its contents with the following-->

The service exposes its cache of messages and two methods:

- + one to add() a message to the cache
- + another to clear() the cache

/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 = [];
  }
}
```

# Inject it into the HeroService

Re-open the HeroService and import the  
MessageService:

/src/app/hero.service.ts

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

Modify the constructor with a parameter that declares  
a private messageService property.

Angular will inject the singleton MessageService into  
that property when it creates the HeroService.

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

This is a typical "service-in-service" scenario: you  
inject the MessageService into the HeroService  
which is injected into the HeroesComponent.

```
import { Injectable } from  
'@angular/core';  
import { Hero } from './hero';  
import { HEROES } from  
'./mock-heroes';  
import { Observable, of } from 'rxjs';  
import { MessageService } from  
'./message.service';  
@Injectable({  
  providedIn: 'root'  
})  
export class HeroService {  
  
  constructor(private messageService:  
MessageService) { }  
  
  getHeroes(): Observable<Hero[]> {  
    return of(HEROES);  
  }  
}
```

# Send a message from HeroService

Modify the getHeroes method to send a message when the heroes are fetched.

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

```
import { Injectable } from '@angular/core';  
import { Hero } from '../hero';  
import { HEROES } from '../mock-heroes';  
import { Observable, of } from 'rxjs';  
import { MessageService } from  
    '../message.service';  
@Injectable({  
    providedIn: 'root'  
})  
export class HeroService {  
  
    constructor(private messageService:  
        MessageService) { }  
  
    getHeroes(): Observable<Hero[]> {  
  
        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.

Open MessagesComponent and import the MessageService.

/src/app/messages/messages.component.ts

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

Modify the constructor with a parameter that declares a public messageService property.

Angular will inject the singleton MessageService into that property when it creates the MessagesComponent.

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

The messageService property must be public because you're about to bind to it in the template.

Angular only binds to public component properties.

```
import { Component, OnInit } from
'@angular/core';
import { MessageService } from
'../message.service';
```

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

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

```
    ngOnInit() {
    }
```

```
}
```

# Bind to the MessageService

Replace the CLI-generated MessagesComponent template with the following.

This template 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().

```
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>
```

# CSS styles to messages.component. css

The messages will look better when you add the private CSS styles to messages.component.css

The browser refreshes and the page displays the list of heroes. Scroll to the bottom to see the message from the HeroService in the message area. Click the "clear" button and the message area disappears.

```
/* MessagesComponent's private CSS styles */
h2 {
  color: red;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
}
body, input[text], button {
  color: crimson;
  font-family: Cambria, Georgia;
}

button.clear {
  font-family: Arial;
  background-color: #eee;
  border: none;
  padding: 5px 10px;
  border-radius: 4px;
  cursor: pointer;
  cursor: hand;
}
button:hover {
  background-color: #cfd8dc;
}
button:disabled {
  background-color: #eee;
  color: #aaa;
  cursor: auto;
}
button.clear {
  color: #888;
  margin-bottom: 12px;
}
```

# Summary

- You refactored data access to the `HeroService` class.
- You registered the `HeroService` as the *provider* of its service at the root level so that it can be injected anywhere in the app.
- You used [Angular Dependency Injection](#) to inject it into a component.
- You gave the `HeroService` *get data* method an asynchronous signature.
- You discovered `Observable` and 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.
- You created a `MessageService` for loosely-coupled communication between classes.
- The `HeroService` injected into a component is created with another injected service, `MessageService`.

## Tour of Tsofen Heroes

### My Heroes

11	Mr. Nice
12	Narco
13	Bombasto
14	Celeritas
15	Magneta
16	RubberMan
17	Dynama
18	Dr IQ
19	Magma
20	Tornado

### Messages

clear

HeroService: fetched heroes



Here are the code files discussed on this page and your app should look like this [live example](#) / [download example](#).



# Create the BackEnd

# Heroes Backend - **pom.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
    <modelVersion>4.0.0</modelVersion>
```

```
    <parent>
```

```
        <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-starter-parent</artifactId>
```

# Heroes Backend - **application.properties**

spring.datasource.driverClassName=org.postgresql.Driver

spring.datasource.url = jdbc:postgresql://localhost:5432/**herodb**

spring.datasource.username=postgres

spring.datasource.password=sako09

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect

spring.jpa.hibernate.ddl-auto = update

spring.jpa.properties.hibernate.jdbc.lob.non\_contextual\_creation=true

# Heroes Backend - Main Class

```
package com.example.Hero;
```

```
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication
```

```
public class HeroApplication {
```

# Heroes Backend - Hero Entity

```
package com.example.Hero.domain;
```

```
import javax.persistence.Entity;
```

```
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.GenerationType;
```

```
import javax.persistence.Id;
```

```
@Entity
```

# Heroes Backend - Hero Repository

```
package com.example.Hero.domain;
```

```
import org.springframework.data.repository.CrudRepository;
```

```
public interface HeroRepository extends CrudRepository<Hero, Long> {
```

```
}
```

# Heroes Backend - **Hero Controller**

```
package com.example.Hero.web;
```

```
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
```

```
class HeroController {
```

```
}
```

```
{
  "_embedded" : {
    "heroes" : [ {
      "name" : "Hero A",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/api/heroes/1"
        },
        "hero" : {
          "href" : "http://localhost:8080/api/heroes/1"
        }
      }
    }, {
      "name" : "Hero B",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/api/heroes/2"
        },
        "hero" : {
          "href" : "http://localhost:8080/api/heroes/2"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/api/heroes"
    },
    "profile" : {
      "href" : "http://localhost:8080/api/profile/heroes"
    }
  }
}
```



No ID's....



# Configuration change **to expose ids**

We need these ids to be exposed in all of our endpoints.

This is different from the default behaviour for Spring Data Rest and thus requires we make some configuration change.

This is done by extending `RepositoryRestConfigurerAdapter` to change our `RepositoryRestConfiguration` to indicate to Spring to expose the Hero ids in its endpoints.

# RepositoryConfig

```
package com.example.Hero.domain;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.data.rest.core.config.RepositoryRestConfiguration;  
import org.springframework.data.rest.webmvc.config.RepositoryRestConfigurer;
```

```
@Configuration
```

```
public class RepositoryConfig implements RepositoryRestConfigurer {
```

```
    @Override
```

```
    public void configureRepositoryRestConfiguration(RepositoryRestConfiguration  
config) {
```

```
        // configure repositoryRestConfig (Use as class)
```

```
{
  "_embedded" : {
    "heroes" : [ {
      "id" : 1,
      "name" : "Hero A",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/api/heroes/1"
        },
        "hero" : {
          "href" : "http://localhost:8080/api/heroes/1"
        }
      }
    }, {
      "id" : 2,
      "name" : "Hero B",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/api/heroes/2"
        },
        "hero" : {
          "href" : "http://localhost:8080/api/heroes/2"
        }
      }
    }
  ]
},
"_links" : {
  "self" : {
    "href" : "http://localhost:8080/api/heroes"
  },
  "profile" : {
    "href" : "http://localhost:8080/api/profile/heroes"
  }
}
}
```



# Configure CORS (aka Cross-Origin Resource Sharing)

CORS (aka Cross-Origin Resource Sharing) configuration tells the browser to allow clients to access the server resource from a website different than that from which they were served. Create **CorsConfig.java** to implement the CORS headers that will authorize access by clients from any domain.

```
package com.example.Hero;
```

```
import java.util.Arrays;
```

```
import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.web.cors.CorsConfiguration;
```

```
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
```

```
import org.springframework.web.filter.CorsFilter;
```

```
@Configuration
```

```
public class CorsConfig {
```

```
    @Bean
```

```
    public CorsFilter corsFilter() {
```

```
        CorsConfiguration config = new CorsConfiguration();
```

Now that we have a working server,  
the next step will be to try to get Angular Tour of Heroes to reference it.  
This will require changes to Angular Tour of Heroes.

# Connect Frontend to Backend

# Front end Angular Tour of Heroes

Here are the code files discussed on slide 100 and your app should look like this [live example](#) / [download example](#).



# HTTP

You'll add the following **data persistence** features with help from 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

**HttpClient** is Angular's mechanism for communicating with a remote server over HTTP.

To make HttpClient available everywhere in the app:

- + open the root AppModule  
**src/app/app.module.ts**

- + import the HttpClientModule symbol from `@angular/common/http`

- + add it to the `@NgModule.imports` array.

```
import { AppComponent } from './app.component';
import { ComponentComponent } from
'./component/component.component';
import { HeroesComponent } from './heroes/heroes.component';
import { FormsModule } from '@angular/forms';
import { HeroDetailComponent } from
'./hero-detail/hero-detail.component';
import { MessagesComponent } from
'./messages/messages.component';
import { AppRoutingModuleModule } from './app-routing.module';
import { DashboardComponent } from
'./dashboard/dashboard.component';
```

```
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  declarations: [
    AppComponent,
    ComponentComponent,
    HeroesComponent,
    HeroDetailComponent,
    MessagesComponent,
    DashboardComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModuleModule,
    HttpClientModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

# Simulate a data server

This tutorial sample mimics communication with a remote data server by using the **In-memory Web API module**.

After installing the module, the app will make requests to and receive responses from the HttpClient without knowing that the In-memory Web API is intercepting those requests, applying them to an in-memory data store, and returning simulated responses.

It may be convenient in the early stages of your own app development when the server's web api is ill-defined or not yet implemented.

# Install the In-memory Web API package from npm

```
npm install angular-in-memory-web-api --save
```

Import the `HttpClientInMemoryWebApiModule` and the `InMemoryDataService` class, which **you will create in a moment**.

## `src/app/app.module.ts`

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

Add the `HttpClientInMemoryWebApiModule` to the `@NgModule.imports` array—after importing the `HttpClient`, —while configuring it with the `InMemoryDataService`.

```
HttpClientModule,
```

```
// The HttpClientInMemoryWebApiModule module intercepts HTTP requests  
// and returns simulated server responses.  
// Remove it when a real server is ready to receive requests.
```

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

The `forRoot()` configuration method takes an `InMemoryDataService` class that primes the in-memory database.

# src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { ComponentComponent } from './component/component.component';
import { HeroesComponent } from './heroes/heroes.component';
import { FormsModule } from '@angular/forms';
import { HeroDetailComponent } from './hero-detail/hero-detail.component';
import { MessagesComponent } from './messages/messages.component';
import { AppRoutingModule } from './app-routing.module';
import { DashboardComponent } from './dashboard/dashboard.component';

import { HttpClientModule } from '@angular/common/http';
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService } from './in-memory-data.service';

@NgModule({
  declarations: [
```

# Create a class src/app/in-memory-data.service.ts

This file replaces mock-heroes.ts, which is now safe to delete.

## Important note:

*When your server is ready,*

*detach the In-memory Web API,*

*and the app's requests will go through to the server.*

```
import { InMemoryDbService } from
'angular-in-memory-web-api';

export class InMemoryDataService implements
InMemoryDbService {
  createDb() {
    const heroes = [
      { id: 11, name: 'Mr. 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' }
    ];
    return {heroes};
  }
}
```

# Heroes and HTTP

Open `src/app/hero.service.ts`

```
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) { }
```



Define the heroesUrl with the address of the heroes resource on the server.

```
export class HeroService {
```

```
  private heroesUrl = 'api/heroes';
```

```
  // 'http://localhost:8080/api/heroes';
```

# Get heroes with HttpClient

The current HeroService.getHeroes() uses the RxJS of() function to return an array of mock heroes as an Observable<Hero[]>.

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

# Convert that method to use HttpClient

```
/** GET heroes from the server */
```

```
getHeroes (): Observable<Hero[]> {
```

```
    return this.http.get<Hero[]>(this.heroesUrl)
```

```
}
```

Refresh the browser. The hero data should successfully load from the mock server.

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

# Http methods return one value

All HttpClient methods return an RxJS Observable of something.

HTTP is a request/response protocol. You make a request, it returns a single response.

In general, 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[]>, literally "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[]>` , gives you a typed result object.

The shape of the JSON data is determined by the server's data API. The Tour of Heroes data API returns the hero data as an array.

# Add an **interface** to define the new structure coming in for our Get requests.

src\app\GetResponse.ts

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

```
export interface GetResponse
```

```
{  
  _embedded: {  
    heroes: Hero[];  
    _links: {  
      self: {href: string}  
    };  
  };  
}
```

```
{  
  "_embedded" : {  
    "heroes" : [ {  
      "id" : 1,  
      "name" : "Hero A",  
      "_links" : {  
        "self" : {  
          "href" : "http://localhost:8080/api/heroes/1"  
        },  
        "hero" : {  
          "href" : "http://localhost:8080/api/heroes/1"  
        }  
      }  
    }, {  
      "id" : 2,  
      "name" : "Hero B",  
      "_links" : {  
        "self" : {  
          "href" : "http://localhost:8080/api/heroes/2"  
        },  
        "hero" : {  
          "href" : "http://localhost:8080/api/heroes/2"  
        }  
      }  
    }  
  ],  
  "_links" : {  
    "self" : {  
      "href" : "http://localhost:8080/api/heroes"  
    },  
    "profile" : {  
      "href" : "http://localhost:8080/api/profile/heroes"  
    }  
  }  
}
```

Now update **getHeroes()** to use this new structure by adding a map inside the pipe function that extracts the heroes array. Note that the types for the get calls are also changed. They are changed to the new interface (GetResponse) from Hero[].

## Open **src/app/hero.service.ts**

Get heroes with HttpClient

Refactor getHeroes () method to use HttpClient

```
/** GET heroes from the server */
```

```
getHeroes (): Observable<Hero[]> {
```

```
    return this.http.get<GetResponse>(this.heroesUrl)
```

```
        .pipe(
```

```
            map(response => response._embedded.heroes),
```

```
        );
```

```
    }
```

**pipe() function in RxJS:** You can use pipes to link operators together. Pipes let you combine multiple functions into a single function.

The **pipe()** function takes as its arguments the functions you want to combine, and returns a new function that, when executed, runs the composed functions in sequence.

<https://angular.io/guide/rx-library>

**Map** in Rxjs used for projection, means you can transform the array in to entirely new array

# Final src/app/hero.service.ts

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

import { Observable, of } from 'rxjs';
import { GetResponse } from '../getresponse';
import { Hero } from '../hero';
import { HEROES } from '../mock-heroes';
import { MessageService } from '../message.service';
import { map, tap, catchError } from 'rxjs/operators';

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

@Injectable({
  providedIn: 'root',
})
export class HeroService {
```



Refresh the browser.

The hero data should successfully load from the rest server.

# Tour of Heroes

## My Heroes

- 1 Hero A
- 2 Hero B
- 3 Hero C

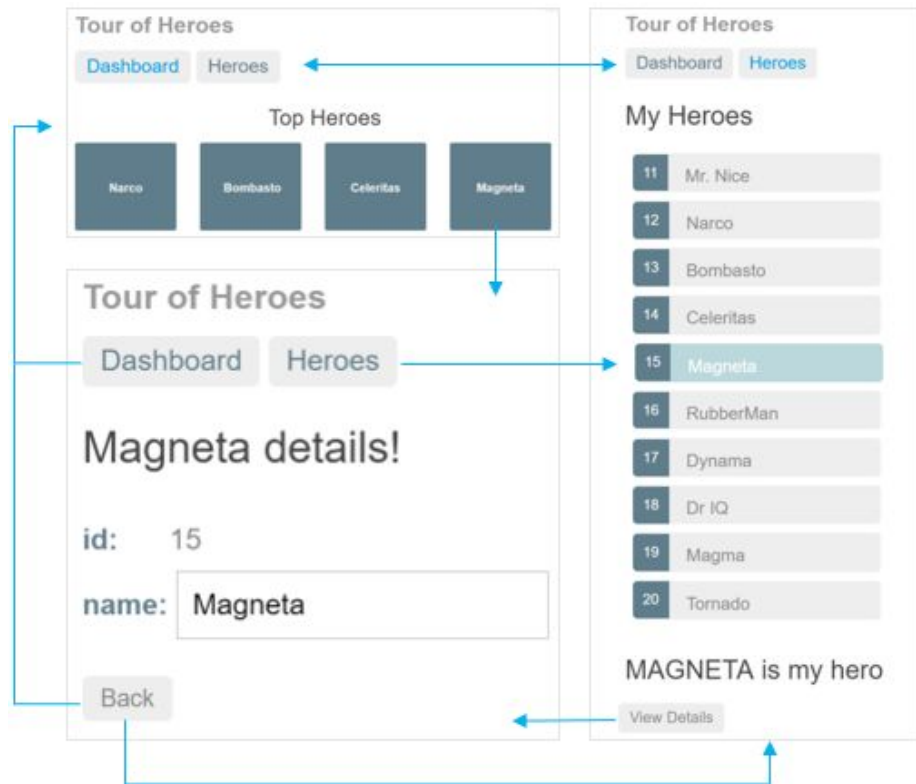
# Add Routing

Add Routing

# Routing

There are new requirements for the Tour of Heroes app:

- Add a Dashboard view
- Add the ability to navigate between the Heroes and Dashboard views
- When users click a hero name in either view, navigate to a detail view of the selected hero



# Client-side navigation with the router

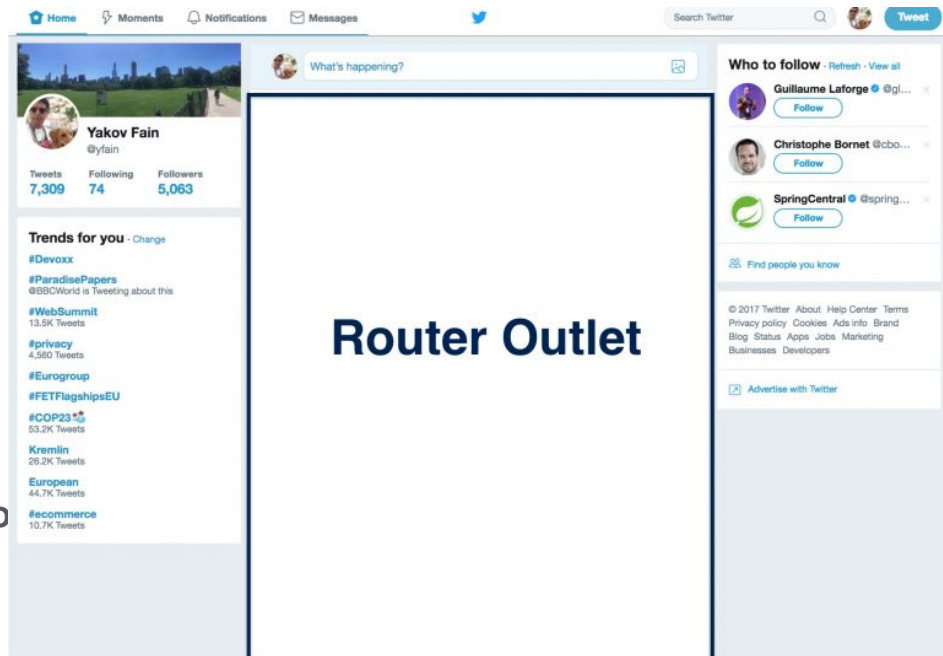
Angular is a good fit for creating **single-page apps (SPA)**.

The entire browser page is not getting refreshed – only the page fragments are getting replaced as the user interacts with your app.

The app navigation is supported by the **Router module**.

Within the component's template, you declare one (or more) **<router-outlet> area** where different views will be rendered as a response to the user interaction or programmatic events.

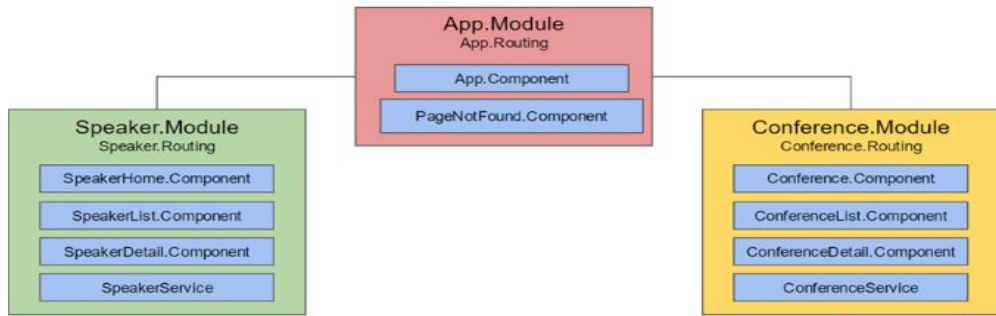
For example, in the Twitter's app you can designate the middle portion to be a **<router-outlet>**, and if the user, say, clicks on the Notifications menu, another component will be rendered in this area.



# Modularization

You can and should **modularize** your app.

Every Angular app has a **root** module, conventionally named **AppModule**, which provides the **bootstrap** mechanism that launches the application.



An app typically **contains many functional/features modules**.

Your app features should be implemented in separate modules, e.g. Billing module, Shipping module et al.

Moreover, these modules can be **loaded lazily** to minimize the initial size of the app.

For example, the landing page of your app may **load the code of the Billing module only after** the user clicks on the Billing button.

The Angular router allows you to specify a preloading strategy so the feature modules can be loaded in the background while the user starts interacting with your app.

# Add the AppRoutingModuleModule

An Angular best practice is to load and configure the router **in a separate, top-level module that is dedicated to routing** and **imported by the root AppModule**.

By convention, the module class name is AppRoutingModuleModule and it belongs in the **app-routing.module.ts** in the **src/app** folder.

Use the CLI to generate it.

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

**--flat** puts the file in **src/app** instead of its own folder.

**--module=app** tells the CLI to register it in the imports array of the **AppModule**.

```
src/app/app-routing.module.ts
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class AppRoutingModuleModule
{ }
```

#### OPEN EDITORS

app.component.html src\app  
heroes.component.html src\app\heroes  
app.module.ts src\app

#### TSOFEN-HERO

e2e  
node\_modules  
src  
  app  
    component  
    hero-detail  
    heroes  
    messages  
    app-routing.module.spec.ts  
    app-routing.module.ts  
    app.component.css  
    app.component.html  
    app.component.spec.ts  
    app.component.ts  
    app.module.ts  
    class.ts  
    hero.service.spec.ts  
    hero.service.ts  
    hero.ts

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppComponent } from './app.component';
5 import { ComponentComponent } from './component/component.component';
6 import { HeroesComponent } from './heroes/heroes.component';
7 import { FormsModule } from '@angular/forms';
8 import { HeroDetailComponent } from './hero-detail/hero-detail.component';
9 import { MessagesComponent } from './messages/messages.component';
10 import { AppRoutingModuleModule } from './app-routing.module'; // <-- NgModel lives here
11 @NgModule({
12   declarations: [
13     AppComponent,
14     ComponentComponent,
15     HeroesComponent,
16     HeroDetailComponent,
17     MessagesComponent
18   ],
19   imports: [
20     BrowserModule,
21     FormsModule,
22     AppRoutingModuleModule
23   ],
24   providers: [],
25   bootstrap: [AppComponent]
26 })
27 export class AppModule { }
28
```

app  
  component  
  hero-detail  
  heroes  
  messages  
  app-routing.module.spec.ts  
  app-routing.module.ts  
  app.component.css  
  app.component.html  
  app.component.spec.ts  
  app.component.ts  
  app.module.ts  
  class.ts  
  hero.service.spec.ts  
  hero.service.ts  
  hero.ts  
  message.service.spec.ts  
  message.service.ts  
  mock-heroes.ts

You generally don't declare components in a routing module so you can:

- delete the `@NgModule.declarations` array
- delete `CommonModule` references too

You'll configure the router with **Routes** in the **RouterModule** so **import** those two symbols from the `@angular/router` library.

**Add** an `@NgModule.exports` array with **RouterModule** in it.

Exporting `RouterModule` makes router directives available for use in the `AppModule` components that will need them.

src/app/app-routing.module.ts

```
import { NgModule } from '@angular/core';  
//import { CommonModule } from '@angular/common';  
import { RouterModule, Routes } from  
'@angular/router';  
  
@NgModule({  
  // declarations: [],  
  // imports: [CommonModule]  
  exports: [ RouterModule ]  
})  
export class AppRoutingModule {}
```



# Add routes

Routes tell the router which view to display when a user clicks a link or pastes a URL into the browser address bar.

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

You intend to navigate to the HeroesComponent when the URL is

<http://localhost:4200/heroes>

1. **Import the HeroesComponent** so you can reference it in a Route.

2. **Define an array of routes** with a single route to that component.

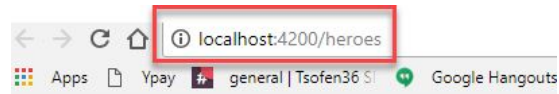
src/app/app-routing.module.ts

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

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

@NgModule({
  // imports: [CommonModule], declarations: []
  exports: [ RouterModule ]

})
export class AppRoutingModule {}
```



## Tour of Tsofen Heroes

### My Heroes

11	Mr. Nice
12	Narco
13	Bombasto
14	Celeritas
15	Magneta
16	RubberMan
17	Dynama
18	Dr IQ
19	Magma
20	Tornado

### Messages

clear

HeroService: fetched heroes

<http://localhost:4200/heroes>

## RouterModule.forRoot()

You first must **initialize the router and start it listening for browser location changes**.

Add RouterModule to the @NgModule.imports array and configure it with the routes by calling RouterModule.forRoot()

```
imports: [  
  RouterModule.forRoot(routes) ],
```

The method is called forRoot() because **you configure the router at the application's root level (app)**.

**The forRoot() method** supplies the service providers and directives needed for routing, and **performs the initial navigation based on the current browser URL**.

src/app/app-routing.module.ts

```
import { NgModule } from '@angular/core';  
// import { CommonModule } from '@angular/common';  
import { RouterModule, Routes } from '@angular/router';  
import { HeroesComponent } from '../heroes/heroes.component';  
  
const routes: Routes = [  
  { path: 'heroes', component: HeroesComponent }  
];  
  
@NgModule({  
  
  // imports: [CommonModule], declarations: []  
  imports: [ RouterModule.forRoot(routes) ],  
  exports: [ RouterModule ]  
  
})  
export class AppRoutingModule {}
```

## Add RouterOutlet

Open the AppComponent template **replace** the `<app-heroes>` element with a `<router-outlet>` element.

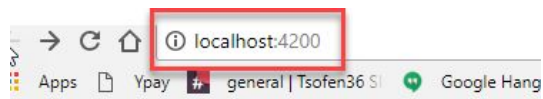
**src/app/app.component.html**

```
<h1>{{title}}</h1>
<!-- <app-heroes></app-heroes> -->
<router-outlet></router-outlet>
<app-messages></app-messages>
```

You removed `<app-heroes>` because you will only display the HeroesComponent when the user navigates to it.

**The `<router-outlet>` tells the router where to display routed views.**

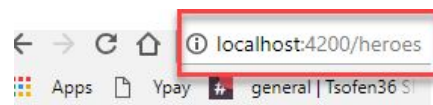
*The RouterOutlet is one of the router directives that became available to the AppComponent because AppModule imports AppRoutingModule which exported RouterModule.*



## Tour of Tsofen Heroes



<http://localhost:4200/heroes>



## Tour of Tsofen He

### My Heroes

11 Mr. Nice

12 Narco

13 Bombasto

14 Celeritas

15 Magneta

16 RubberMan

17 Dynama

18 Dr IQ

19 Magma

20 Tornado

### Messages

clear

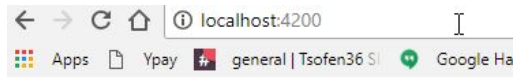
HeroService: fetched heroes

## Add a navigation link (routerLink)

Users should be able to click a link to navigate.

Add a `<nav>` element and, within that, an anchor element that, when clicked, triggers navigation to the HeroesComponent. **src/app/app.component.html**

```
<h1>{{title}}</h1>
<nav>
  <a routerLink="/heroes">Heroes</a>
</nav>
<!-- <app-heroes></app-heroes> -->
<router-outlet></router-outlet>
<app-messages></app-messages>
```



### Tour of Tsofen Heroes

Heroes



### Tour of Tsofen Heroes

Heroes

My Heroes

11	Mr. Nice
12	Narco
13	Bombasto
14	Celeritas
15	Magneta
16	RubberMan
17	Dynama
18	Dr IQ
19	Magma
20	Tornado

Messages

clear

HeroService: fetched heroes

A `routerLink` attribute is set to `"/heroes"`, the string that the router matches to the route to HeroesComponent. 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.

# Add some CSS : `src/app/app.component.css`

Tour of Tsofen Heroes

Heroes

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

```
h1 {  
  font-size: 1.2em;  
  color: #999;  
  margin-bottom: 0;  
}
```

```
h2 {  
  font-size: 2em;  
  margin-top: 0;  
  padding-top: 0;  
}
```

```
nav a {  
  padding: 5px 10px;
```



localhost:4211

Apps



Adding a sprite | DGi



S

# Tour of Heroes

Heroes



localhost:4211/heroes

Apps



Adding a sprite | DGi



Stylish : rb-ko.com

## Tour of Heroes

Heroes

## My Heroes

1

Hero A

2

Hero B

3

Hero C

## Messages

clear

HeroService: fetched heroes



**Add a dashboard view**

**Add a dashboard view**

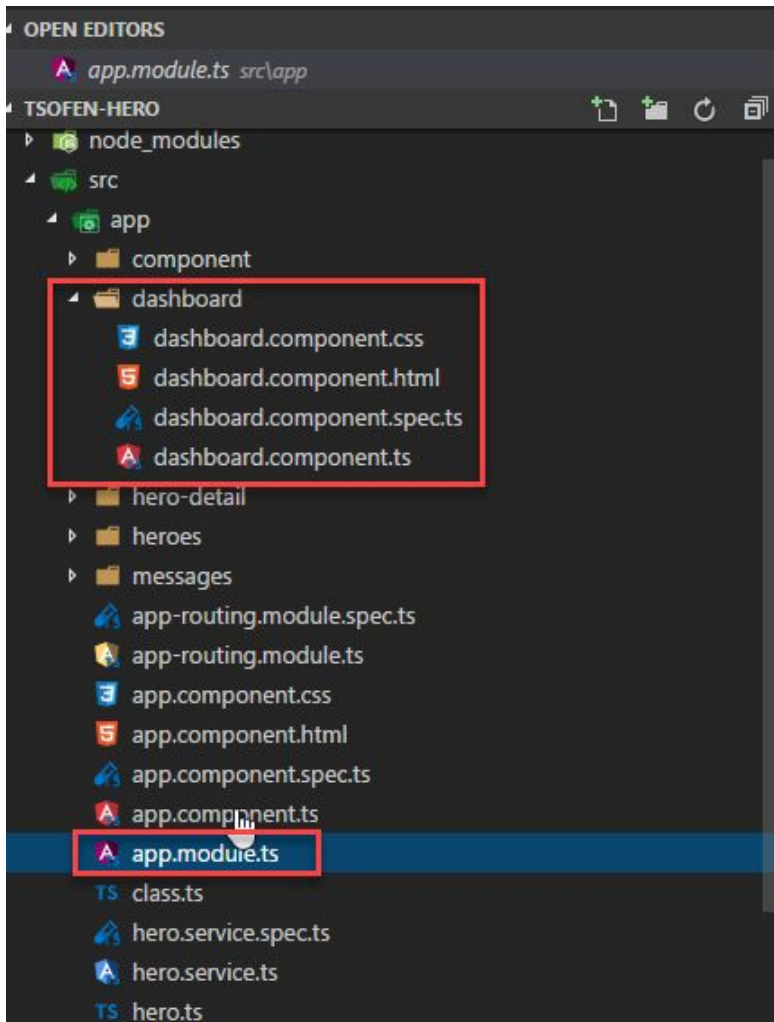
# Add a dashboard view

Routing makes more sense when there are multiple views.  
So far there's only the heroes view.

**Add a DashboardComponent using the CLI:**

```
ng generate component dashboard
```

The CLI generates the files for the DashboardComponent and declares it in AppModule.



```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppComponent } from './app.component';
5 import { ComponentComponent } from './component/component.component';
6 import { HeroesComponent } from './heroes/heroes.component';
7 import { FormsModule } from '@angular/forms';
8 import { HeroDetailComponent } from './hero-detail/hero-detail.component';
9 import { MessagesComponent } from './messages/messages.component';
10 import { AppRoutingModuleModule } from './app-routing.module';
11 import { DashboardComponent } from './dashboard/dashboard.component';
12 @NgModule({
13   declarations: [
14     AppComponent,
15     ComponentComponent,
16     HeroesComponent,
17     HeroDetailComponent,
18     MessagesComponent,
19     DashboardComponent
20   ],
21   imports: [
22     BrowserModule,
23     FormsModule,
24     AppRoutingModuleModule
25   ],
26   providers: [],
27   bootstrap: [AppComponent]
28 })
29 export class AppModule { }
```

# src/app/dashboard/dashboard.component.ts

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()`.

This `getHeroes()` **reduces the number of heroes displayed to four (2nd, 3rd, 4th, and 5th).**

```
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));
  }
}
```

# src/app/dashboard/dashboard.component.html

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!!!.*

```
<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>
```

# src/app/dashboard/**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;
```

# Add the dashboard route

To navigate to the dashboard, the router needs an appropriate route.

Import the DashboardComponent in the AppRoutingModuleModule.

src/app/app-routing.module.ts

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

Add a route to the AppRoutingModuleModule.routes array that matches a path to the DashboardComponent.

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

src/app/app-routing.module.ts

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

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

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

```
@NgModule({
```

```
  // imports: [CommonModule], declarations: []  
  imports: [ RouterModule.forRoot(routes) ],  
  exports: [ RouterModule ]
```

```
})
```

```
export class AppRoutingModule {}
```

# Add a default route

When the app starts, the browsers address bar points to the web site's root.

That doesn't match any existing route so the router doesn't navigate anywhere. The space below the <router-outlet> is blank.

To make the app navigate to the dashboard automatically, add the following route to the AppRoutingModule.Routes array.

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

This route redirects a URL that fully matches the empty path to the route whose path is '/dashboard'.

After the browser refreshes, the router loads the DashboardComponent

The browser address bar shows the /dashboard URL.

src/app/app-routing.module.ts

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

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

const routes: Routes = [
  { path: 'heroes', component: HeroesComponent }
,
  { path: 'dashboard', component: DashboardComponent }
,
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' }
,
];

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



Tour of Tsofen Heroes

Heroes

Top Heroes

Narco

Bombasto

Celeritas

Magneta

Messages

clear

HeroService: fetched heroes

# Add dashboard link to the shell

The user should be able to navigate back and forth between the DashboardComponent and the HeroesComponent by clicking links in the navigation area near the top of the page.

Add a dashboard navigation link to the AppComponent shell template, just above the Heroes link.

After the browser refreshes you can navigate freely between the two views by clicking the links.

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>
```

Tour of Tsofen Heroes

Dashboard Heroes

Narco

Messages

clear

HeroService: fetched heroes



# Navigating to hero details

The HeroDetailsComponent displays details of a selected hero.

At the moment the HeroDetailsComponent is only visible at the bottom of the HeroesComponent

The user should be able to get to these details in three ways.

- +By clicking a hero in the dashboard.

- +By clicking a hero in the heroes list.

- +By pasting a "deep link" URL into the browser address bar that identifies the hero to display.

You'll enable navigation to the  
HeroDetailsComponent

and

liberate it from the HeroesComponent.

# Delete hero details from HeroesComponent

When the user clicks a hero item in the HeroesComponent, the app should navigate to the HeroDetailComponent, replacing the heroes list view with the hero detail view.

The heroes list view should no longer show hero details as it does now.

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

Clicking a hero item now does nothing. You'll fix that shortly after you enable routing to the HeroDetailComponent.

## heroes/heroes.component.html

```
<h2>My Heroes</h2>
<ul class="heroes">
  <li *ngFor="let hero of heroes"
      [class.selected]="hero ===
selectedHero"
      (click)="onSelect(hero)">
    <span
class="badge">{{hero.id}}</span>
    {{hero.name}}
  </li>
</ul>

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

# Add a hero detail route

A URL like ~/detail/11 would be a good URL for navigating to the Hero Detail view of the hero whose id is 11.

Open AppRoutingModuleModule and import HeroDetailComponent.

src/app/app-routing.module.ts

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

Then add a parameterized route to the AppRoutingModuleModule.routes array that matches the path pattern to the hero detail view.

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

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

src/app/app-routing.module.ts

```
const routes: Routes = [  
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },  
  { path: 'dashboard', component: DashboardComponent },  
  { path: 'detail/:id', component: HeroDetailComponent },  
  { path: 'heroes', component: HeroesComponent }  
];
```

# DashboardComponent hero links

The DashboardComponent hero links do nothing at the moment.

Now that the router has a route to HeroDetailComponent, fix the dashboard hero links to navigate via the parameterized dashboard route.

src/app/dashboard/dashboard.component.htm

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

You're using Angular interpolation binding within the \*ngFor repeater to insert the current iteration's hero.id into each routerLink.

src/app/dashboard/dashboard.component.htm

```
<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>
```

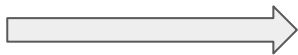


# HeroesComponent hero links

The **hero** items in the **HeroesComponent** are `<li>` elements whose click events are bound to the component's `onSelect()` method.

**src/app/heroes/heroes.component.html**

```
<ul class="heroes">
  <li *ngFor="let hero of heroes"
      [class.selected]="hero === selectedHero"
      (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span>
    {{hero.name}}
  </li>
</ul>
```



Strip the `<li>` back to just its `*ngFor`, wrap the badge and name in an anchor element (`<a>`), and add a `routerLink` attribute to the anchor that is the same as in the dashboard template

**src/app/heroes/heroes.component.html**

```
<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>
```

# Remove dead code (optional)

While the HeroesComponent class still works, the onSelect() method and selectedHero property are no longer used.

Here's the class after pruning away the dead code.

```
src/app/heroes/heroes.component.ts
export class HeroesComponent implements
OnInit {
    heroes: Hero[];

    constructor(private heroService:
HeroService) { }

    ngOnInit() {
        this.getHeroes();
    }

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

# Routable HeroDetailComponent

Previously, 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.

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

# src/app/hero-detail/hero-detail.component.ts

The **ActivatedRoute** holds information about the route to this instance of the HeroDetailComponent.

This component is interested in the route's bag of 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 and 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.

Add the following imports:

**src/app/hero-detail/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

**src/app/hero-detail/hero-detail.component.ts**

```
ngOnInit(): void {  
    this.getHero();  
}  
  
getHero(): void {  
    const id =  
+this.route.snapshot.paramMap.get('id');  
  
    this.heroService.getHero(id)  
        .subscribe(hero => this.hero = hero);  
}  
  
goBack(): void {  
    this.location.back();  
}
```

The `route.snapshot` is a static image of the route information shortly after the component was created.

The `paramMap` is a dictionary of route parameter values extracted from the URL.

The "id" key returns the id of the hero to fetch.

Route parameters are always strings.

The JavaScript **(+)** operator converts the string to a number, which is what a hero id should be.

Add a `goBack()` method to the component class that navigates backward one step in the browser's history stack using the Location service that you injected previously.

The browser refreshes and the app crashes with a compiler error. `HeroService` doesn't have a `getHero()` method. Add it now.

# Final

## src/app/hero-detail/hero-detail.component.ts

```
import { Component, OnInit, Input } from '@angular/core';  
import { ActivatedRoute } from '@angular/router';  
import { Location } from '@angular/common';
```

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

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

```
export class HeroDetailComponent implements OnInit {
```

# Add HeroService.getHero()

Note the backticks ( ` ) that define a JavaScript template literal for embedding the id.

Try it

The browser refreshes and the app is working again.

You can click a hero in the dashboard or in the heroes list and navigate to that hero's detail view.

If you paste localhost:4200/detail/1 in the browser address bar, the router navigates to the detail view for the hero with id: 1, "Hero A".

src/app/hero.service.ts

```
getHero(id: number): Observable<Hero> {  
  const url = `${this.heroesUrl}/${id}`;  
  return this.http.get<Hero>(url);  
}
```

## src/app/hero.service.ts

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

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

```
import { GetResponse } from './getresponse';
```

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

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

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

```
import { map, tap, catchError } from 'rxjs/operators';
```

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

```
@Injectable({
```



## Find the way back

By clicking the browser's back button, you can go back to the hero list or dashboard view, depending upon which sent you to the detail view.

It would be nice to have a button on the HeroDetail view that can do that.

Add a go back button to the bottom of the component template and bind it to the component's goBack() method.

**src/app/hero-detail/hero-detail.component.html**

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

# Summary

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.

# Update heroes

# Update heroes

Editing 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 with a click event binding that invokes a new component method named `save()`.

src/app/hero-detail/hero-detail.component.html (save)

```
<button (click)="save()">save</button>
```

src/app/hero-detail/hero-detail.component.ts (save)

```
save(): void {  
    this.heroService.updateHero(this.hero)  
        .subscribe(() => this.goBack());  
}
```

The overall structure of the `updateHero()` method is similar to that of `getHeroes()`, but it uses `http.put()` to persist the changed hero on the server.

`src/app/hero.service.ts (update)`

```
/** PUT: update the hero on the server
 */
updateHero (hero: Hero):
Observable<any> {
    const url =
`${this.heroesUrl}/${hero.id}`;
    return this.http.put(url, hero,
httpOptions);
}
```

Add `const` at the head of the class

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

## Add `HeroService.updateHero()`

The `HttpClient.put()` method takes three parameters

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

+The heroes web API expects a special header in HTTP save requests. That header is in the `httpOptions` constant defined in the `HeroService`.

Refresh the browser, change a hero name, save your change, and click the "go back" button. The hero now appears in the list with the changed name.

**Add a new hero**

# Add a new hero

To add a hero, this app only needs the hero's name.  
You can use an input element paired with an add button.

src/app/heroes/**heroes.component.html** (add)

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

```
<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>
```

In response to a click event, call the component's click handler and then clear the input field so that it's ready for another name.

src/app/heroes/**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);  
    });  
}
```

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.

You'll write **HeroService.addHero** in the next section.



# Add HeroService.addHero()

src/app/hero.service.ts (addHero)

```
/** POST: add a new hero to the server */  
addHero (hero: Hero): Observable<Hero> {  
    return this.http.post<Hero>(this.heroesUrl,  
httpOptions);  
}
```

Tour of Tsofen Heroes <https://tsofen36.slack.com/messages/>

Dashboard Heroes

11 Mr. Sergey

12 Narco

13 Bombasto

14 Celeritas

15 Magneta

16 RubberMan

17 Dynama

18 Dr IQ

19 Magma

20 Tornado

21 Pini

Hero name:

# Delete a hero

```
DELETE FROM heroes WHERE id = 1;
```

# Delete a hero

Each hero in the heroes list should have a delete button.

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

src/app/heroes/heroes.component.html

```
<ul class="heroes">  
  <li *ngFor="let hero of heroes">  
    <a routerLink="/detail/{{hero.id}}">  
      <span  
class="badge">{{hero.id}}</span>  
      {{hero.name}}  
    </a>  
    <button class="delete" title="delete  
hero"  
      (click)="delete(hero)">x</button>  
  </li>  
</ul>
```

## Add the delete() handler

### src/app/heroes/heroes.component.ts (delete)

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

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(). It must subscribe anyway.

# Add HeroService.deleteHero()

src/app/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, httpOptions);
}
```

## Position the delete button at the far right of the hero entry

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

```
.heroes {  
  margin: 0 0 2em 0;  
  list-style-type: none;  
  padding: 0;  
  width: 15em;  
}
```

```
.heroes li {  
  position: relative;  
  cursor: pointer;  
  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 a {  
  color: #888;
```

<https://github.com/finstertthecat/heroes-frontend>





# Angular Tour of Heroes With JSON

db

<https://docs.google.com/presentation/d/1kIGJFGHioLloWMwcrNhEltow7TLigInZQH4duJlQvwY/edit#slide=id.p10>