

Desarrollo Web SPA con Angular

2. Componentes



Componentes

Componentes en Angular

- Un componente es una **nueva etiqueta HTML** con una **vista** y una **lógica** definidas por el desarrollador
- La **vista** es una plantilla (*template*) en HTML con elementos especiales
- La **lógica** es una clase TypeScript vinculada a la vista

Componentes en Angular

- Copiamos el contenido de la carpeta src/app del ejem1

▲ app

app.component.html

app.component.ts

app.module.ts

Componentes en Angular

app.component.ts

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

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

app.component.html

```
<h1>My First Angular App</h1>
```

Lógica


Vista

Componentes en Angular

app.component.ts

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

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



app.component.html

```
<h1>My First Angular App</h1>
```

Lógica

Vista

Componentes en Angular

app.component.ts

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

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

Este componente no
tiene ninguna lógica

Lógica

app.component.html

```
<h1>My First Angular App</h1>
```

Vista

Componentes

ejem1

app.component.ts

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

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

app.component.html

```
<h1>My First Angular App</h1>
```

src/index.html

```
<html>
<head>...</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Para usar el componente se incluye en el **index.html** un **elemento HTML** con el nombre indicado en el selector (en este caso **app-root**)

Componentes en Angular

Toda app tiene un módulo que define los componentes de la app, el componente principal y qué otros módulos necesita

app.module.ts

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

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

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

Componentes declarados

Módulos importados

Componente principal

Componentes



Componentes en Angular

app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
    <h1>
      My First Angular App
    </h1>
  `
})
export class AppComponent {
}
```

Se puede incluir la **vista** (HTML del template) directamente en la **clase**. Si se usa la tildes invertidas (`) (grave accent), se puede escribir HTML multilínea

Visualización de una variable

La vista del componente (**HTML**) se genera en función de su estado (**atributos de la clase**)

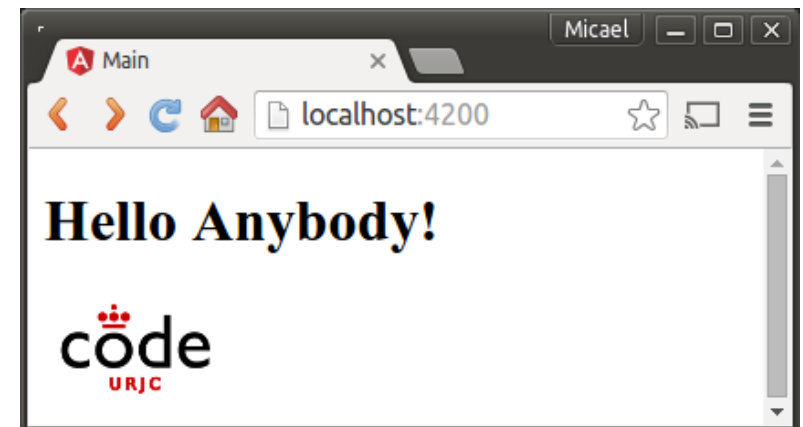
app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  name = 'Anybody';
  imgUrl = "assets/img.png";
}
```

app.component.html

```
<h1>Hello {{name}}!</h1>
<img [src]="imgUrl"/>
```



Visualización de una variable

La vista del componente (**HTML**) se genera en función de su estado (**atributos de la clase**)

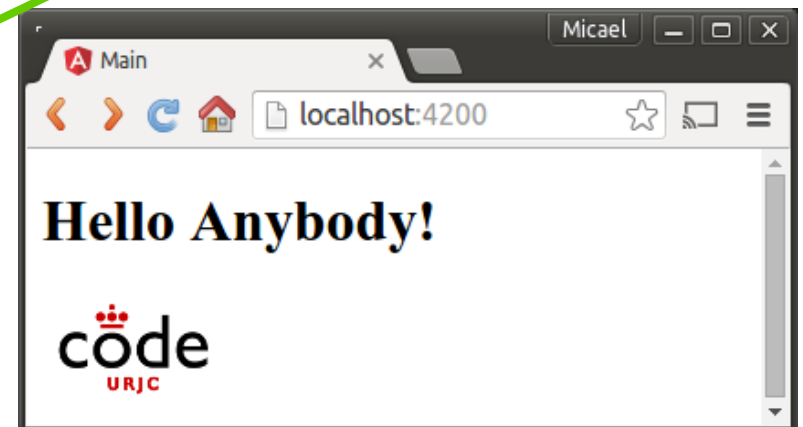
app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  name = 'Anybody';
  imgUrl = "assets/img.png";
}
```

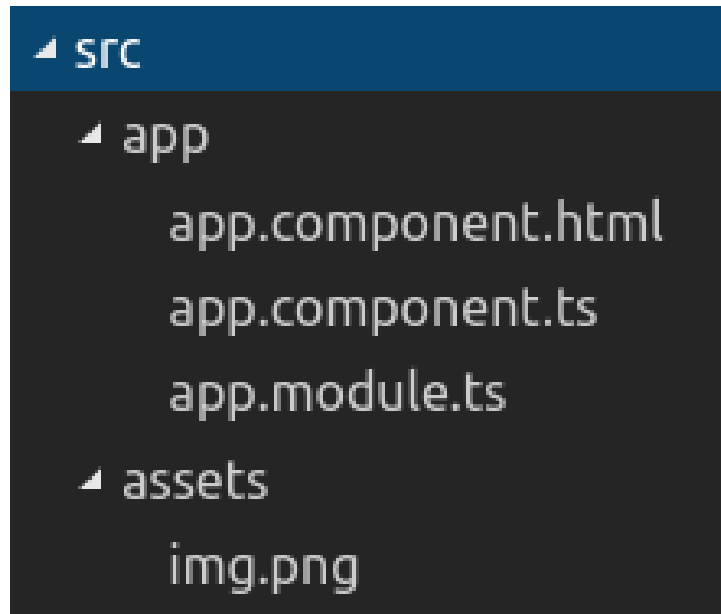
app.component.html

```
<h1>Hello {{name}}!</h1>
<img [src]="imgUrl"/>
```



Recursos de la app

Los recursos (imágenes, fonts..) deben colocarse en una carpeta **src/assets** para que estén accesibles en **desarrollo** y cuando se genera el paquete de **producción**



Ejecución de lógica

Se puede ejecutar un método ante un evento producido en la vista del componente

app.component.ts

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

@Component({
  selector: 'app',
  templateUrl: './app.component.html'
})
export class AppComponent {

  name = 'Anybody';

  setName(name:string){
    this.name = name;
  }
}
```

app.component.html

```
<h1>Hello {{name}}!</h1>

<button (click)="setName('John')">
  Hello John
</button>
```

Ejecución de lógica

Se puede ejecutar un método ante un evento producido en la vista del componente

app.component.ts

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

@Component({
  selector: 'app',
  templateUrl: './app.component.html'
})
export class AppComponent {

  name = 'Anybody';

  setName(name:string){
    this.name = name;
  }
}
```

app.component.html

```
<h1>Hello {{name}}!</h1>
<button (click)="setName('John')">
  Hello John
</button>
```



Ejecución de lógica

Se puede ejecutar un método ante un evento producido en la vista del componente

app.component.ts

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

@Component({
  selector: 'app',
  templateUrl: './app.component.html'
})
export class AppComponent {

  name = 'Anybody';

  setName(name:string){
    this.name = name;
  }
}
```

app.component.html

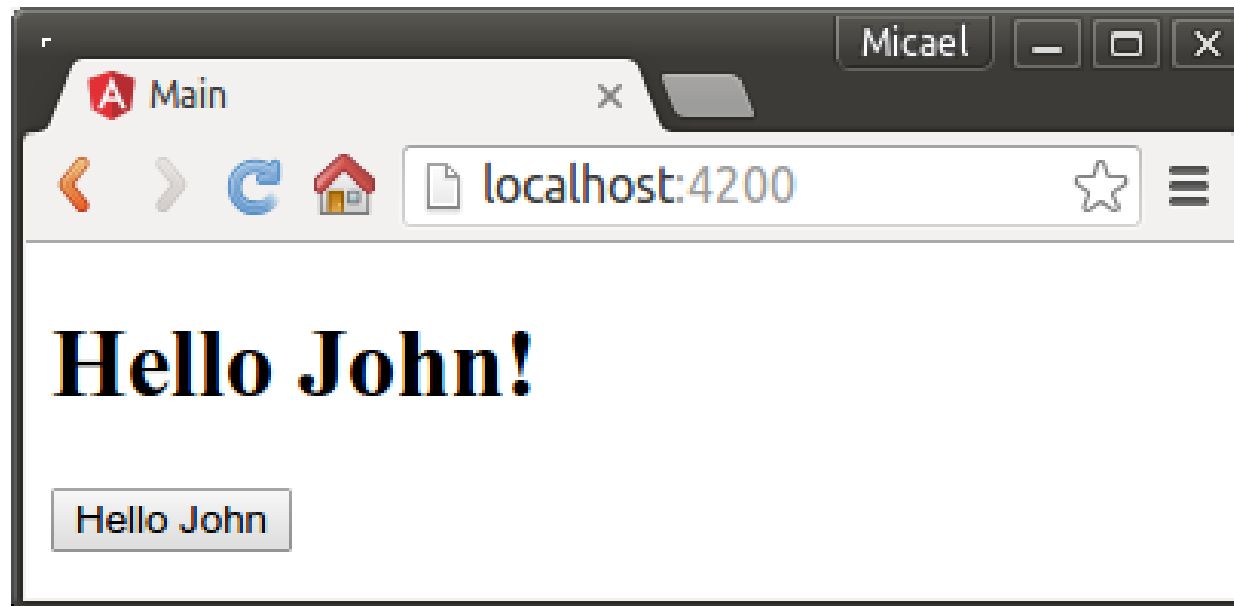
```
<h1>Hello {{name}}!</h1>

<button (click)="setName('John')">
  Hello John
</button>
```

Se puede definir cualquier **evento** disponible en el **DOM** para ese elemento

Ejecución de lógica

Se puede ejecutar un método ante un evento producido en la vista del componente



Sintaxis de los templates

```
<h1>Hello {{name}}!</h1>
```

```
<img [src]="imgUrl"/>
```

```
<button (click)="setName('John')">  
    Hello John  
</button>
```

{{ attr }}

Valor del atributo de la clase
en el texto

[prop]="attr"

Valor del atributo de la clase
en la propiedad del elemento

(event)="metodo()"

Se llama al método cuando
se produce el evento

Datos enlazados (*data binding*)

Un campo de texto se puede “enlazar” a un atributo
Atributo y campo de texto están sincronizados

app.component.ts

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

@Component({
  selector: 'app',
  templateUrl: './app.component.html'
})
export class AppComponent {
  name = 'Anybody';

  setName(name:string){
    this.name = name;
  }
}
```

app.component.html

```
<input type="text" [(ngModel)]="name">
<h1>Hello {{name}}!</h1>
<button (click)="setName('John')">
  Hello John
</button>
```

Datos enlazados (*data binding*)

Un campo de texto se puede “enlazar” a un atributo
Atributo y campo de texto están sincronizados

app.component.ts

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

@Component({
  selector: 'app',
  templateUrl: './app.component.html'
})
export class AppComponent {
  name = 'Anybody';

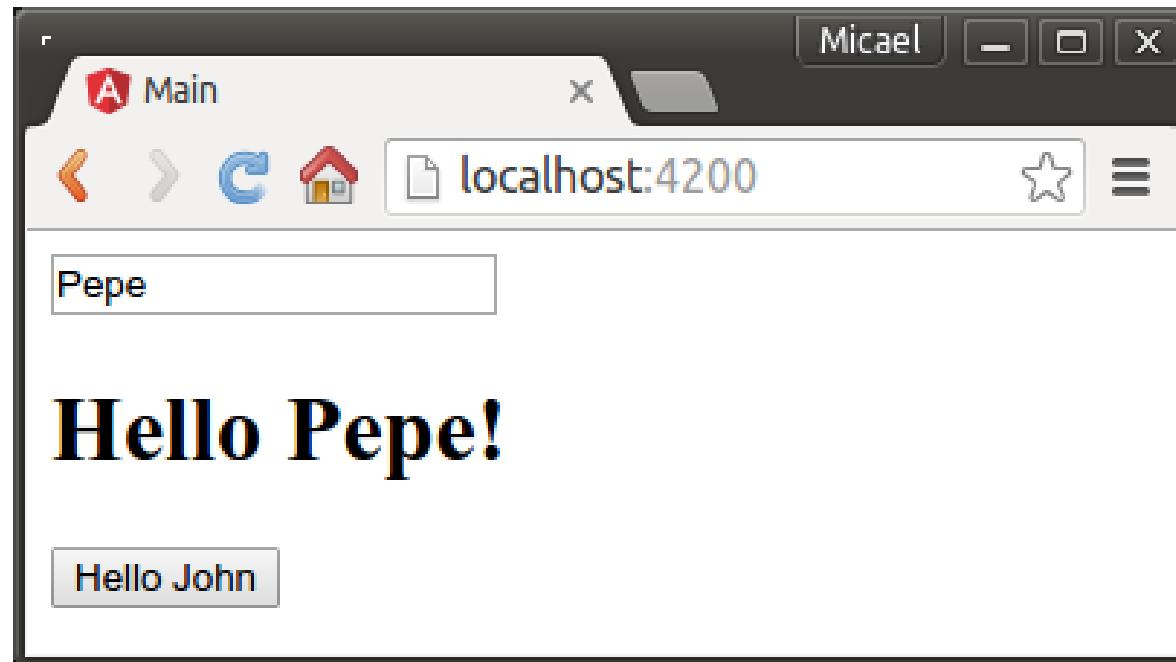
  setName(name:string){
    this.name = name;
  }
}
```

app.component.html

```
<input type="text" [(ngModel)]="name">
<h1>Hello {{name}}!</h1>
<button (click)="setName('John')">
  Hello John
</button>
```

Datos enlazados (*data binding*)

Un campo de texto se puede “enlazar” a un atributo
Atributo y componente están sincronizados



Templates

- Los **templates** permiten definir la vista en función de la información del componente
 - Visualización condicional
 - Repetición de elementos
 - *Safe navigation operator*
 - Pipes
 - Estilos
 - Formularios

<https://angular.io/docs/ts/latest/guide/template-syntax.html>

- **Visualización condicional**

- Se puede controlar si un elemento aparece o no en la página dependiendo del valor de un atributo de la clase usando la **directiva ngIf**
- Por ejemplo dependiendo del **valor del atributo** booleano visible

```
<p *ngIf="visible">Text</p>
```

- También se puede usar una **expresión**

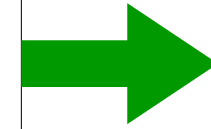
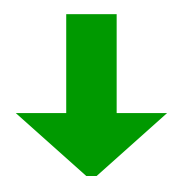
```
<p *ngIf="num == 3">Num 3</p>
```

- Repetición de elementos

- Es posible visualizar el contenido de un array con la **directiva ngFor**
- Se define cómo se visualizará cada elemento

```
<div *ngFor="let elem of elems">{{elem.desc}} </div>
```

```
elems = [  
  { desc: 'Elem1', check: true },  
  { desc: 'Elem2', check: true },  
  { desc: 'Elem3', check: false }  
]
```



```
<div>Elem1</div>  
<div>Elem2</div>  
<div>Elem3</div>
```

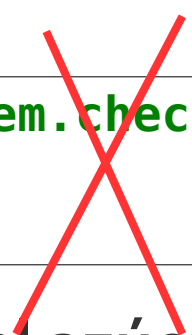
- **Directivas**

- Las **directivas** modifican a los elementos en los que se incluyen
- Existen muchas **directivas predefinidas** y podemos programar nuestras propias directivas
- Las directivas **estructurales** empiezan por * y **modifican el DOM** del documento (*ngIf, *ngFor, *ngSwitch)
- El * es **azúcar sintáctico** para la definición del template

• Directivas

- No se pueden incluir dos directivas estructurales (de tipo *) en el mismo elemento

```
<li *ngFor="let elem of elems" *ngIf="elem.check">
  {{elem.desc}}
</li>
```



- Hay que usar la versión de las **directivas sin el azúcar sintáctico (*)**, en su versión extendida con el **elemento template** (que no aparece en el DOM)

```
<ng-template ngFor let-elem [ngForOf]="elems">
  <li *ngIf="elem.check">{{elem.desc}}</li>
</ng-template>
```


- *Safe Navigation Operator*

- Si el atributo **user** tiene valor undefined, se produce un error y no se muestra el componente

```
User's name: {{user.name}}
```

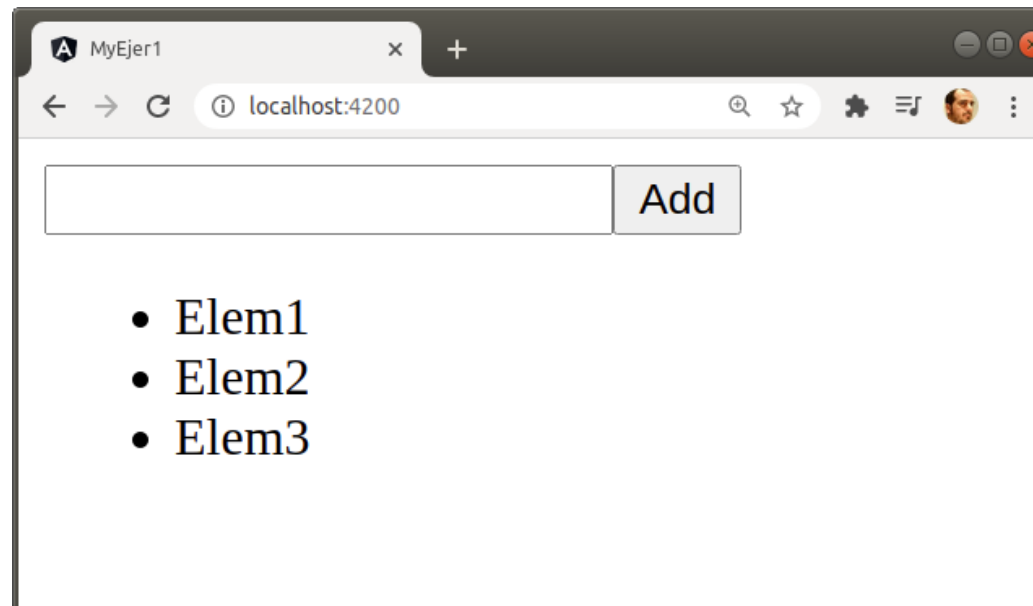
- Para evitarlo existe el *safe navigator operator*. Undefined se representa como cadena vacía

```
User's name: {{user?.name}}
```



Ejercicio 1

- Implementa una aplicación con un **campo de texto** y un **botón de Añadir**
- Cada vez que se pulse el **botón**, el **contenido** del campo de texto se **añadirá al documento**



- Visualización condicional
 - Se puede mostrar un contenido si la expresión no se cumple (**else**)

```
<div *ngIf="isValid; else otherCont">  
    valid content  
</div>  
<ng-template #otherCont>  
    <div>invalid content</div>  
</ng-template>
```

Las etiquetas <ng-template> no se visualizan por defecto en el navegador

Templates

```
<div *ngIf="isValid; else otherCont">  
    valid content  
</div>  
<ng-template #otherCont>  
    <div>invalid content</div>  
</ng-template>
```



```
<div>  
    valid content  
</div>
```

```
<div>  
    invalid content  
</div>
```


- Pipes

- Los pipes (tuberías) permiten especificar en la plantilla cómo **formatear** valores del componente (en vez de **toString**)

```
<p>Today is {{ birthday | date }}</p>
```

- Existen muchos pipes predefinidos: Fechas, Mayúsculas, etc..
- También pueden recibir parámetros

```
<p>Today is {{ birthday | date:"MM/dd/yy" }} </p>
```

- El desarrollador puede implementar sus pipes

Estilos CSS

- Existen 3 formas de definir un CSS en Angular
 - 1) Globalmente asociado al index.html
 - Local al componente:
 - 2) En la propiedad `styles` o `styleUrls` de `@Component`
 - 3) En el template

- **1) Globalmente asociado al index.html**
 - Si creamos un fichero **src/styles.css** se incluirá de forma automática en el **index.html**
 - Podemos añadir más ficheros .css a la carpeta assets o descargados de NPM.
 - Hay que añadir esos ficheros en el fichero **angular.json**, entrada "styles"

Definir CSS en Angular

- **1) Globalmente asociado al index.html**
 - Para añadir Bootstrap CSS
 - Instalamos la dependencia NPM
- Añadimos la referencia al CSS en el angular.cli

```
npm install --save bootstrap
```

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

<https://www.tektutorialshub.com/angular/angular-global-css-styles/>

- 2) En la propiedad `styles` o `styleUrls` de `@Component`

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styles: [`  
    .red { color: red; }  
    .blue { color: blue; }  
  `]  
})  
export class AppComponent {  
  ...  
}
```


Se suelen usar los strings
multilínea con tildes
invertidas

- 2) En la propiedad `styles` o `styleUrls` de `@Component`



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

- 3) En el template (.html)



```
<style>
  .orange {
    color: orange;
  }
</style>

<h1 [class]="className">Hello {{name}}!</h1>

<button (click)="setClass('blue')">Blue</button>
...
```


- Definir el estilo de un elemento
 - Hay diversas formas de controlar los estilos de los elementos
 - 1) Asociar la **clase** de un elemento a un atributo de tipo string
 - 2) Activar una **clase concreta** con un atributo boolean
 - 3) Activar las **clases** con un mapa de **string a boolean**
 - 4) Asociar un **estilo concreto** de un elemento a un atributo

Definir el estilo de un elemento

ejem5

- 1) Asociar la clase de un elemento a un atributo string
 - Cambiando el valor del atributo se cambia la clase del elemento
 - Por ejemplo, la clase del elemento h1 se cambia modificando el atributo **className** del componente

```
<h1 [class]="className">Title!</h1>
```

Definir el estilo de un elemento

ejem5

- 2) Activar una clase concreta con un atributo boolean
- Activa o desactiva una clase red con el valor del atributo booleano **redActive**

```
<h1 [class.red]="redActive">Title!</h1>
```

- Se puede usar para varias clases

```
<h1 [class.red]="redActive"
    [class.yellow]="yellowActive">
    Title!
</h1>
```

Definir el estilo de un elemento

ejem5

- 3) Activar las clases con un mapa

- Para gestionar varias clases es mejor usar un mapa de string (nombre de la clase) a boolean (activa o no)



```
<p [ngClass]="pClasses">Text</p>
```

```
pClasses = {  
  "red": false,  
  "bold": true  
}
```

```
changeParagraph() {  
  this.pClasses.bold = true;  
}
```

Definir el estilo de un elemento

- **4) Asociar un estilo concreto a un atributo**
 - En algunos casos es mejor cambiar el estilo directamente en el elemento

```
<p [style.backgroundColor]="pColor">Text</p>
```

- Con unidades

```
<p [style.fontSize.em]="pSizeEm">Text</p>
```

```
<p [style.fontSize.%]="pSizePerc">Text</p>
```

Definir el estilo de un elemento

- 4) Asociar un estilo concreto a un atributo
 - Usando mapas de propiedad a valor

```
<p [ngStyle]="getStyles()">Text</p>
```

```
getStyles(){  
  return {  
    'font-style':this.canSave? 'italic':'normal',  
    'font-weight':!this.isUnchanged? 'bold':'normal',  
    'font-size':this.isSpecial? '24px':'8px',  
  }  
}
```

Formularios

- Existen diversas formas de controlar formularios en Angular
 - 1) ***Data binding***: Vincular un control del formulario a un atributo del componente
 - 2) Acceso a los controles desde el **código** para leer y modificar su estado
 - Aspectos avanzados que no veremos
 - **NgFrom FormGroup**
 - **Reactive Forms**

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

- 1) *Data binding* en campo de texto
 - Se vincula el control a un atributo del componente con `[(ngModel)]`
 - Cualquier cambio en el control se refleja en el valor del atributo (y viceversa)

```
<input type="text" [(ngModel)]="name">  
<p>{{name}}</p>
```

name:string



A green arrow points from the 'name' attribute in the HTML code above to the 'name' property in the TypeScript code below. Both 'name' strings are circled in green.

- **1) *Data binding* en checkbox (boolean)**
 - Cada control se asocia con [(ngModel)] a un atributo booleano y su valor depende de si está "checked"

```
<input type="checkbox" [(ngModel)]="angular"/>  
Angular  
<input type="checkbox" [(ngModel)]="javascript"/>  
JavaScript
```

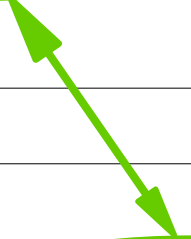
```
angular:boolean  
javascript:boolean
```



- 1) *Data binding* en checkbox (objetos)
- Cada control se asocia con [(ngModel)] a un atributo booleano de un objeto de un array

```
<span *ngFor="let item of items">  
  <input type="checkbox"  
    [(ngModel)]="item.selected"/> {{item.value}}  
</span>
```

```
items = [  
  {value: 'Item1', selected: false},  
  {value: 'Item2', selected: false}  
]
```



- **1) *Data binding* en botones de radio**

- Todos los botones del mismo grupo se asocian al mismo atributo con [(ngModel)]
- El valor del atributo es el "value" del control

```
<input type="radio" name="gender"  
  [(ngModel)]=gender value="Male"> Male  
<input type="radio" name="gender"  
  [(ngModel)]=gender value="Female"> Female
```

gender:string

• 2) Acceso a los controles desde el código

- *Template reference variables*
- Un elemento del template puede asociarse a una variable
- Podemos usar esa variable en el código del template para manejar ese elemento

```
<input #cityInput type="text">
```

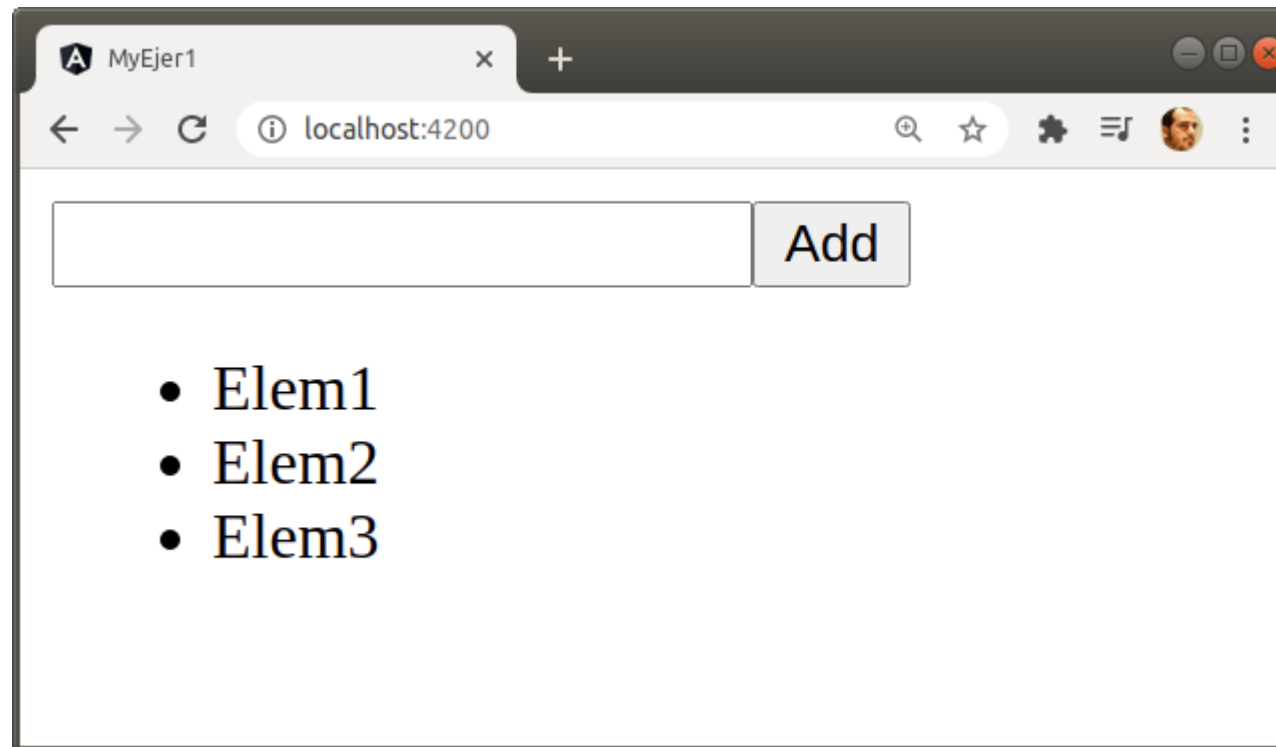
Este control input puede referenciarse con el nombre **cityInput** en el template

```
<button (click)="update(cityInput.value); cityInput.value=' '>  
  Update city  
</button>
```

Podemos usar la propiedades y métodos del elemento definidos en su API DOM

Ejercicio 1b

- Implementa el ejercicio 1 usando *Template Reference Variables*



- **2) Acceso a los controles desde el código**
 - También podemos acceder al elemento desde el código del componente
 - Creamos un atributo en el componente de tipo **ElementRef**
 - Anotamos ese atributo con **@ViewChild('refName')**
 - El atributo apuntará al elemento con ese **refName**

- **2) Acceso a los controles desde el código**

```
<input #titleInput type="text">
<button (click)="updateTitle()">Update Title</button>
<p>Title: {{title}}</p>
```

```
export class AppComponent {

  @ViewChild('titleInput') titleInput: ElementRef;
  title: string;

  updateTitle() {
    this.title = this.titleInput.nativeElement.value;
    this.titleInput.nativeElement.value = '';
  }
}
```


- 2) Acceso a los controles desde el código

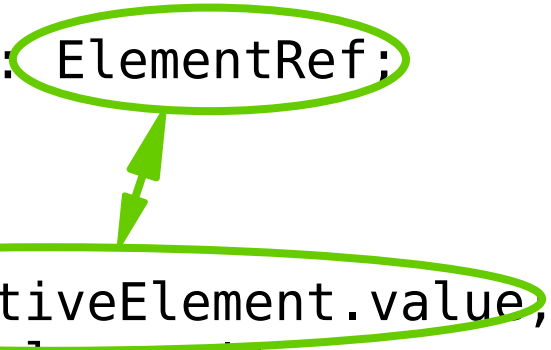
```
<input #titleInput type="text">  
<button (click)="updateTitle()">Update Title</button>  
<p>Title: {{title}}</p>
```

```
export class AppComponent {  
  @ViewChild('titleInput') titleInput: ElementRef;  
  title: string;  
  
  updateTitle() {  
    this.title = this.titleInput.nativeElement.value;  
    this.titleInput.nativeElement.value = '';  
  }  
}
```

- 2) Acceso a los controles desde el código

```
<input #titleInput type="text">  
<button (click)="updateTitle()">Update Title</button>  
<p>Title: {{title}}</p>
```

```
export class AppComponent {  
  
  @ViewChild('titleInput') titleInput: ElementRef;  
  title: string;  
  
  updateTitle() {  
    this.title = this.titleInput.nativeElement.value;  
    this.titleInput.nativeElement.value = '';  
  }  
}
```



Formularios

ejem6

Ejem1

localhost:4200

Anybody

Anybody

☐ Male ☐ Female Selected gender is

☐ Angular ☐ JavaScript

☒ Item1 ☐ Item2

- Item1

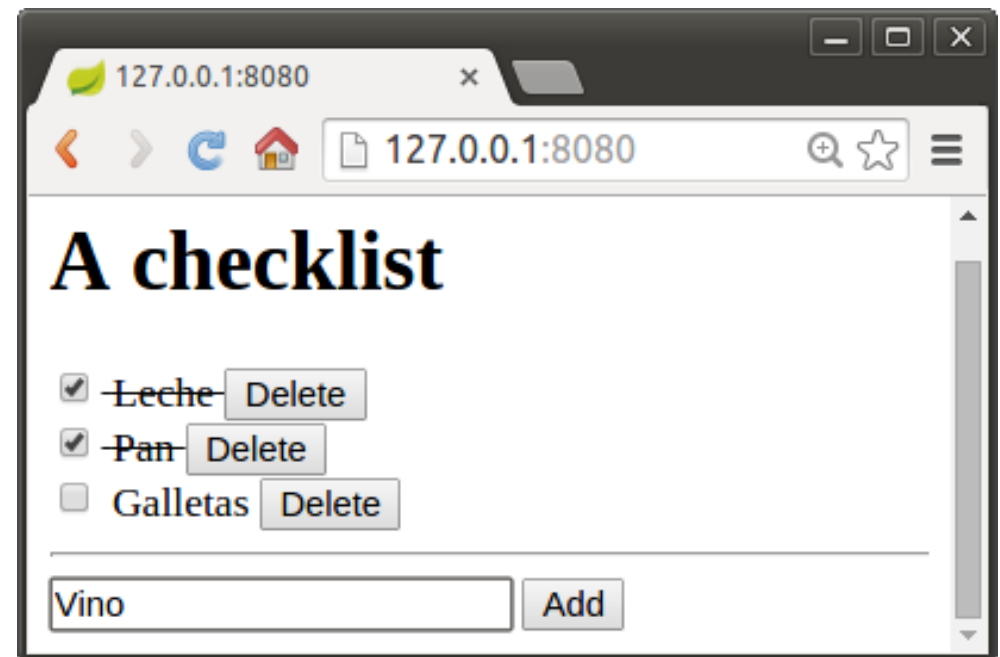
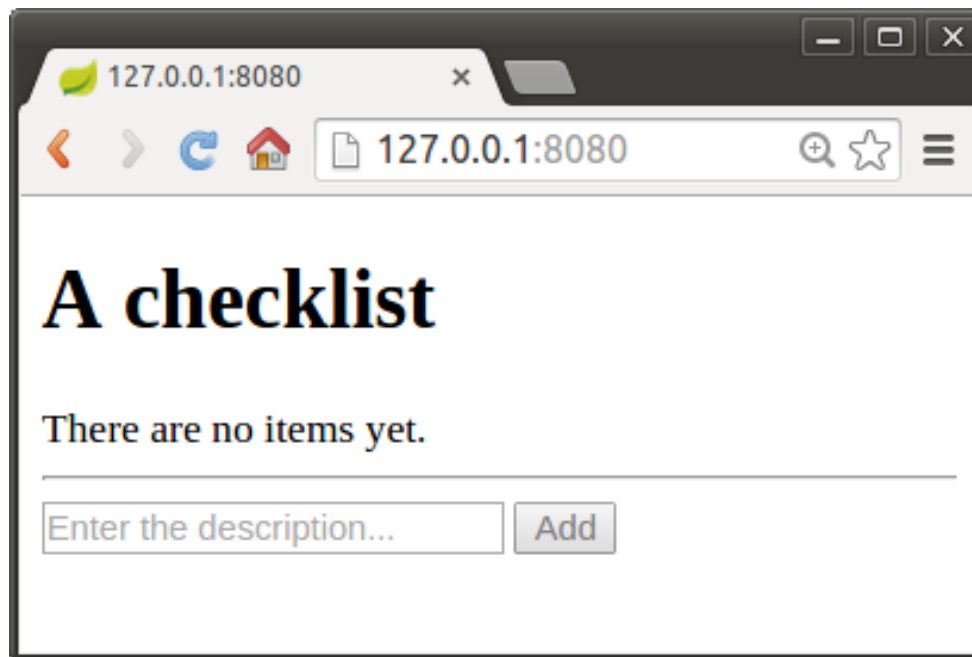
Change

Toledo Update city

City:

Ejercicio 2

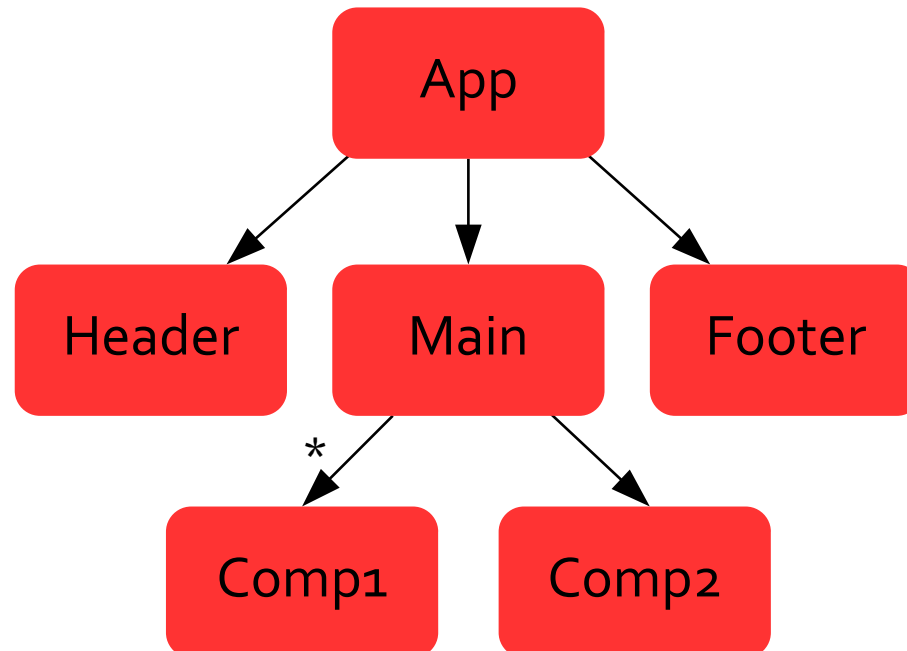
- Implementa una aplicación de **gestión de tareas**
- Las tareas se mantendrán en **memoria**



Composición de componentes

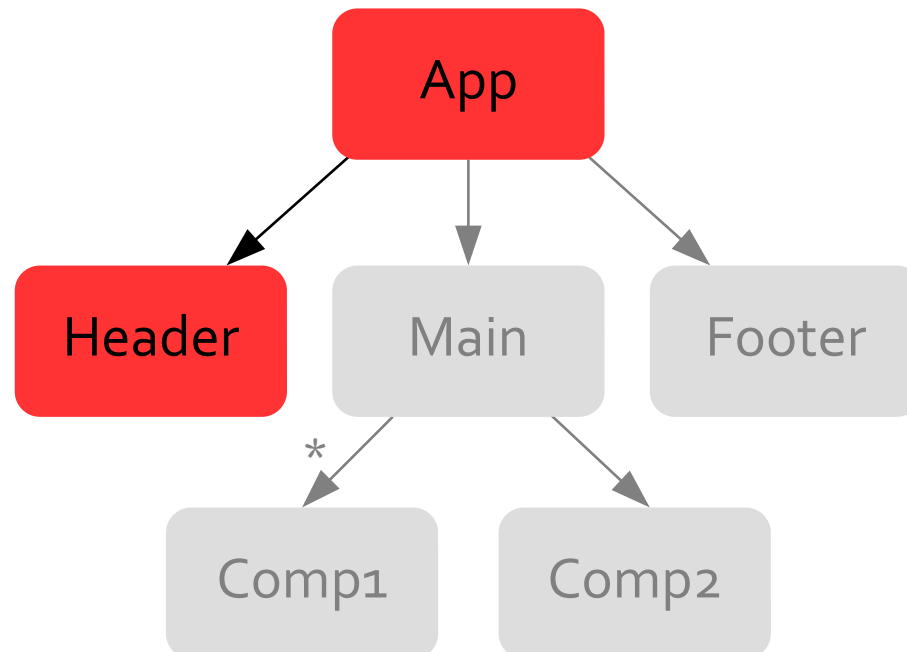
Árboles de componentes

En Angular un componente puede estar formado por más componentes formando un árbol



Árboles de componentes

En Angular un componente puede estar formado por más componentes formando un árbol



Composición de componentes

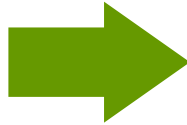
Árboles de componentes

App



Header

```
<h1>Title</h1>  
<p>Main content</p>
```



```
<header></header>  
<p>Main content</p>
```

```
<header>
```

```
<h1>Title</h1>
```


Composición de componentes

Árboles de componentes

App



Header

app.component.ts

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

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

app.component.html

```
<header></header>
<p>Main content</p>
```

header.component.ts

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

@Component({
  selector: 'header',
  templateUrl:
    './header.component.html'
})
export class HeaderComponent {}
```

header.component.html

```
<h1>Title</h1>
```

Composición de componentes

ejem7

Árboles de componentes

App

Header

app.component.ts

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

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

header.component.ts

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

@Component({
  selector: 'header',
  templateUrl:
    './header.component.html'
})
export class HeaderComponent {}
```

app.component.html

```
<header></header>
<p>Main content</p>
```

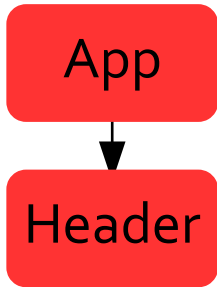
header.component.html

```
<h1>Title</h1>
```

Para incluir un
componente se usa
su **selector**

Composición de componentes

Árboles de componentes



app.module.ts

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

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

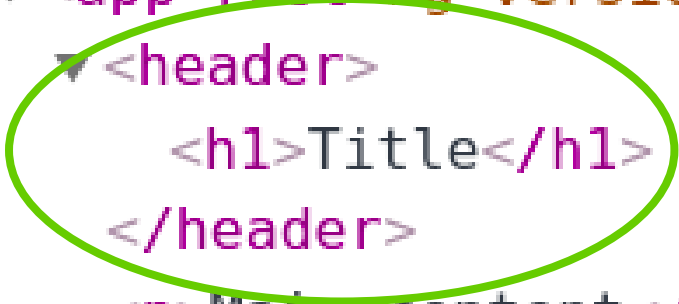
@NgModule({
  declarations: [AppComponent, HeaderComponent],
  imports: [BrowserModule, FormsModule],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Todos los
componentes de
la app deben
declararse en el
@NgModule

Árboles de componentes

- Al cargar la app en el navegador, en el árbol **DOM** cada componente incluye en su **elemento** el contenido de la **vista** (HTML)

```
▼ <app-root ng-version="9.0.7">  
  ▼ <header>  
    <h1>Title</h1>  
  </header>  
  <p>Main content</p>  
</app-root>
```



Composición de componentes

- Comunicación entre un **componente padre** y un **componente hijo**
 - Configuración de propiedades (Padre → Hijo)
 - Envío de eventos (Hijo → Padre)
 - Invocación de métodos (Padre → Hijo)
 - Con variable template
 - Inyectando hijo con @ViewChild
 - Compartiendo el mismo servicio (Padre Hijo)

Configuración de propiedades (Padre → Hijo)

- El componente padre puede especificar **propiedades** en el componente hijo como si fuera un elemento **nativo HTML**

El título de <header> será el valor del atributo appTitle

```
<header [title]='appTitle'></header>  
<p>Main content</p>
```

Configuración de propiedades (Padre → Hijo)

app.component.ts

```
...  
  
export class AppComponent {  
  appTitle = 'Main Title';  
}
```

header.component.ts

```
import {Component, Input} from  
  '@angular/core';  
...  
export class HeaderComponent {  
  
  @Input()  
  title: string;  
}
```

app.component.html

```
<header [title]='appTitle'></header>  
<p>Main content</p>
```

header.component.html

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

Composición de componentes

Configuración de propiedades (Padre → Hijo)

app.component.ts

```
...  
  
export class AppComponent {  
  appTitle = 'Main Title';  
}
```

app.component.html

```
<header [title]='appTitle'></header>  
<p>Main content</p>
```

header.component.ts

```
import {Component, Input} from  
  '@angular/core';  
...  
export class HeaderComponent {  
  @Input()  
  title: string;  
}
```

header.component.html

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


Composición de componentes

Configuración de propiedades (Padre → Hijo)

app.component.ts

```
...  
  
export class AppComponent {  
  appTitle = 'Main Title';  
}
```

header.component.ts

```
import {Component, Input} from  
  '@angular/core';  
...  
export class HeaderComponent {  
  
  @Input()  
  title: string;  
}
```

app.component.html

```
<header [title]='appTitle'></header>  
<p>Main content</p>
```

header.component.html

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

Envío de eventos (Hijo → Padre)

- El componente hijo puede generar eventos que son atendidos por el padre como si fuera un elemento **nativo HTML**

La variable \$event apunta al evento generado

```
<header (hidden)='hiddenTitle($event)'></header>  
<p>Main content</p>
```

Envío de eventos (Hijo → Padre)

app.component.ts

```
...  
export class AppComponent {  
  hiddenTitle(hidden: boolean){  
    console.log("Hidden:"+hidden)  
  }  
}
```

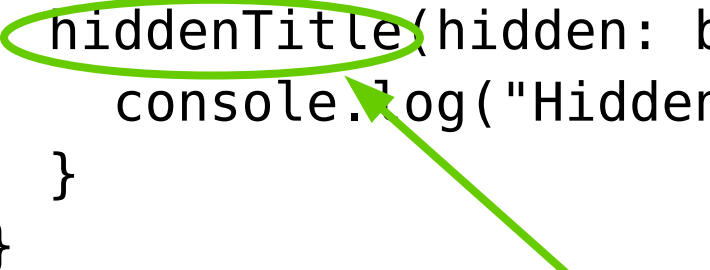
app.component.html

```
<header (hidden)='hiddenTitle($event)'></header>  
<p>Main content</p>
```

Envío de eventos (Hijo → Padre)

app.component.ts

```
...  
export class AppComponent {  
  hiddenTitle(hidden: boolean){  
    console.log("Hidden:"+hidden)  
  }  
}
```



app.component.html

```
<header (hidden)=hiddenTitle($event)></header>  
<p>Main content</p>
```

Envío de eventos (Hijo → Padre)

app.component.ts

```
...  
export class AppComponent {  
  hiddenTitle(hidden: boolean) {  
    console.log("Hidden:" + hidden)  
  }  
}
```

Los eventos pueden tener valores que se capturan con **\$event**

app.component.html

```
<header (hidden)='hiddenTitle($event)'></header>  
<p>Main content</p>
```

Envío de eventos (Hijo → Padre)

header.component.ts

```
import {Component, Output, EventEmitter} from '@angular/core';
...
export class HeaderComponent {

    @Output()
    hidden = new EventEmitter<boolean>();

    visible = true;

    click(){
        this.visible = !this.visible;
        this.hidden.next(this.visible);
    }
}
```

header.component.html

```
<h1 *ngIf="visible">Title</h1>
<button (click)='click()'>Hide/Show</button>
```

Envío de eventos (Hijo → Padre)

header.component.ts

```
import {Component, Output, EventEmitter} from '@angular/core';
...
export class HeaderComponent {
  @Output()
  hidden = new EventEmitter<boolean>();

  visible = true;

  click(){
    this.visible = !this.visible;
    this.hidden.emit(this.visible);
  }
}
```

Se declara un atributo de tipo **EventEmitter** con la anotación **@Output**

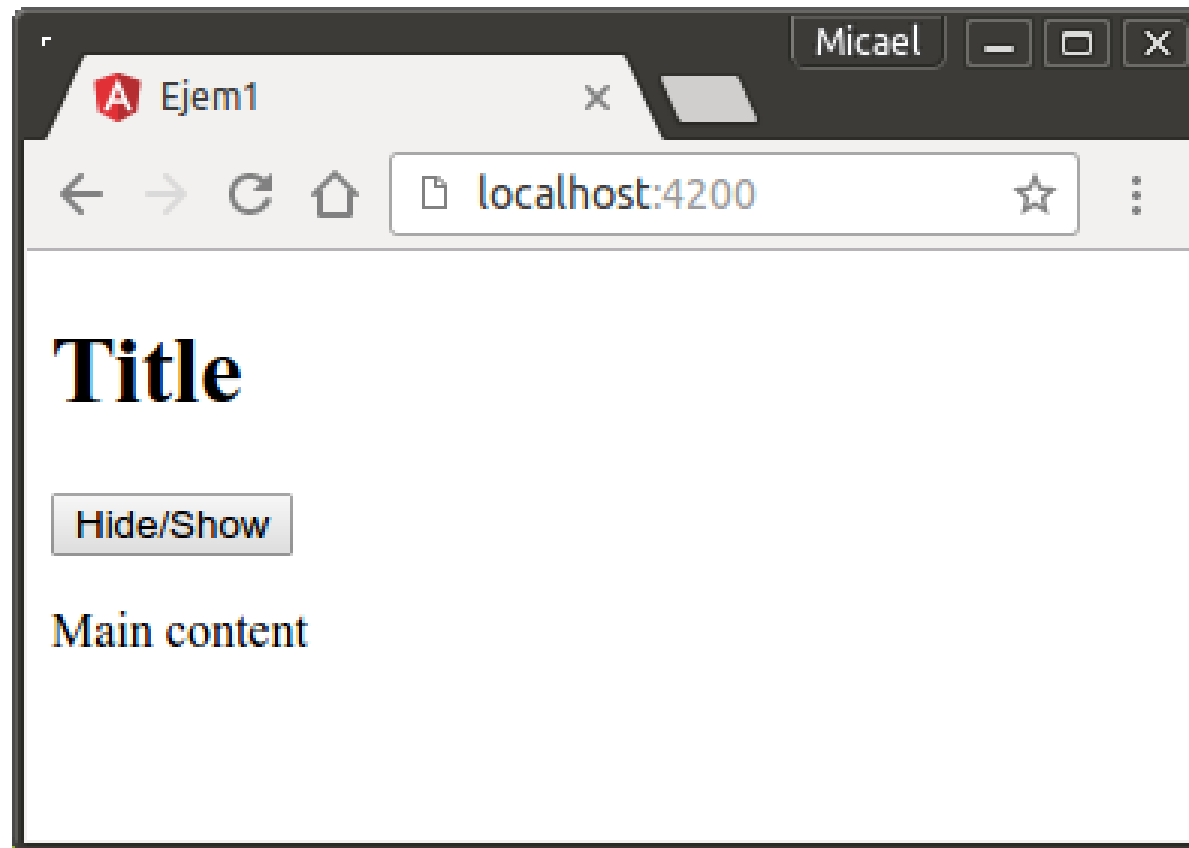
Para **lanzar un evento** se invoca el método **emit(valor)**

header.component.html

```
<h1 *ngIf="visible">Title</h1>
<button (click)='click()'>Hide/Show</button>
```

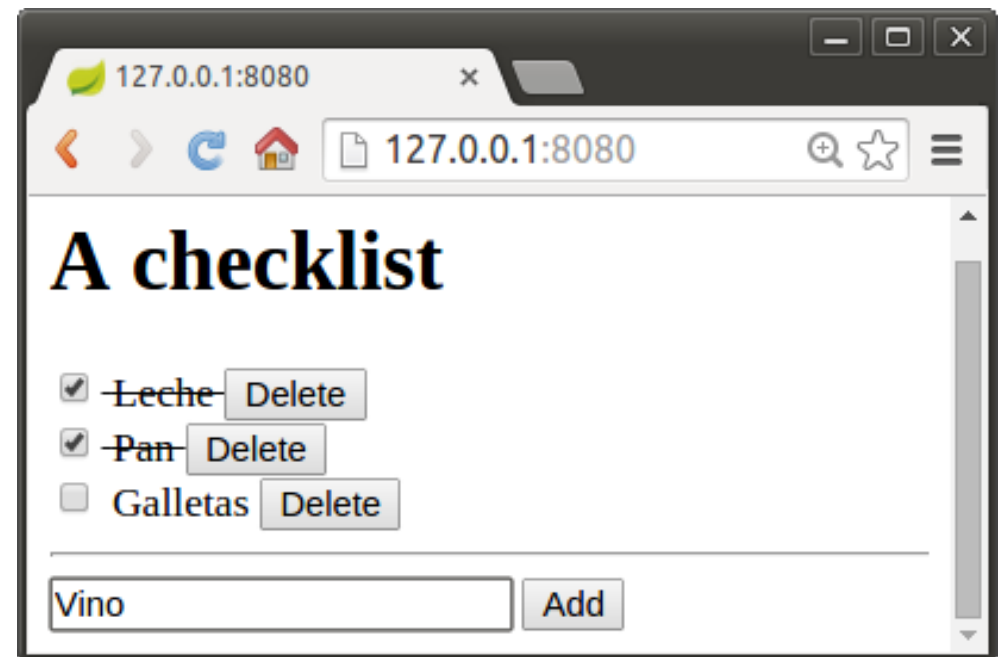
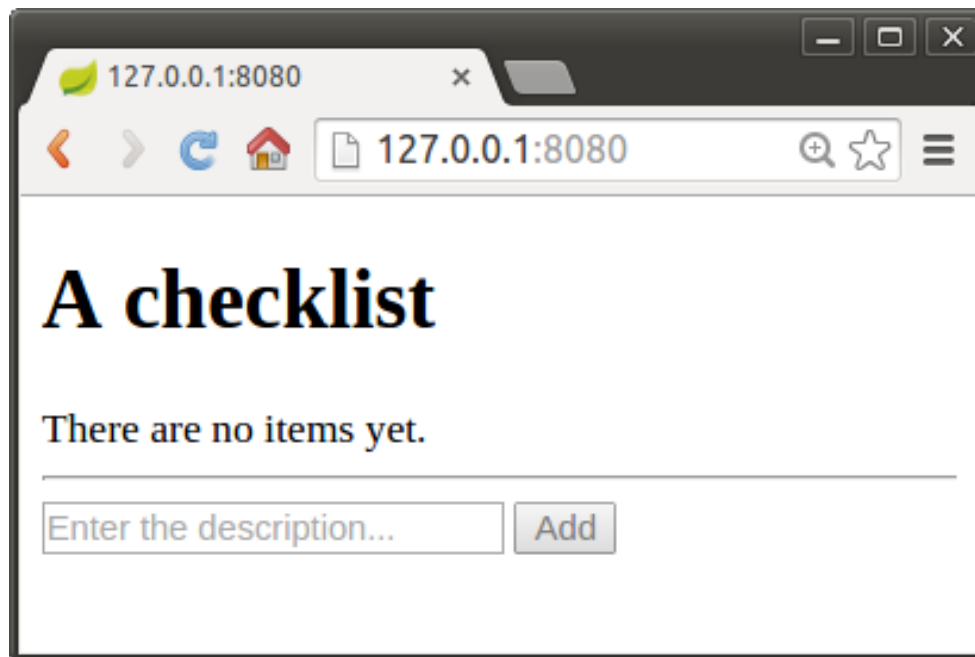
Composición de componentes

Envío de eventos (Hijo → Padre)



Ejercicio 3

- Refactoriza la aplicación de **gestión de tareas** para que cada tarea se visualice en su propio componente



Composición de componentes

- ¿Cuándo crear un nuevo componente?
 - El ejercicio y los ejemplos son **excesivamente sencillos** para que compense la creación de un nuevo componente **hijo**
 - En casos reales se crearían nuevos componentes:
 - Cuando la lógica y/o el *template* sean suficientemente **complejos**
 - Cuando los componentes hijos puedan **reutilizarse** en varios contextos