

# Angular

## Interfaces & Directivas



# Orientado a Objetos

---

Uno de los principios básicos de TypeScript es la verificación de tipos.

- Podemos usar tipos primitivos en variables:

```
let name: string;  
let price: number;
```

- Pero también podemos usar una **interface** para definir tipos de datos más complejos.

# Interfaces

— — —

1. Definimos una **interface** para nuestras cervezas.


```
export interface Beer {  
  name: string;  
  style: string;  
  price: number;  
  stock: number;  
}
```

2. Creamos una variable del tipo **Beer**

```
let beer: Beer = {  
  name: 'Bitter Call Saul',  
  style: 'Ipa',  
  price: 180,  
  stock: 300  
};
```

```
console.log(beer.name);  
console.log(beer.style);  
console.log(beer.precio);
```

any  
Property 'precio' does not exist on type 'Beer'. (2339)  
[Peek Problem](#) [Quick Fix...](#)



# Interfaces

— — —

Cuáles son las ventajas de tipar todos los datos?

[TBC]



**Trabajemos con una lista de cervezas  
(con una no hacemos nada)**

# Refactor para trabajar con una lista de cervezas

— — —

Ya tenemos definido el tipo **Beer**, pero ahora necesitamos trabajar con una colección de cervezas.

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

Importamos la  
interface **Beer**

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

```
export class BeerListComponent implements OnInit {  
  beers: Beer[] = [  
    {  
      name: 'Bitter Call Saul',  
      style: 'Ipa',  
      price: 180,  
      stock: 300  
    },  
    ...  
  ]  
}
```

Definimos un arreglo que almacena  
objetos del tipo **Beer**

# Creamos un mock

---

Se le llama Mock a un objeto que simula el comportamiento del objeto real.

- Es una buena práctica front-end usar 'Fake Data' para comenzar a desarrollar nuestra aplicación.
- Eventualmente los datos vendrán de algún web service (API).
- Se puede separar en un archivo.

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

@Component({
  selector: 'beer-list',
  templateUrl: './beer-list.component.html',
  styleUrls: ['./beer-list.component.css']
})
export class BeerListComponent implements OnInit {
  beers: Beer[] = [
    {
      name: 'Bitter Call Saul',
      style: 'Ipa',
      price: 180,
      stock: 300
    },
    {
      name: 'Red Red Wine',
      style: 'Barley Wine',
      price: 200,
      stock: 100
    },
    {
      name: 'Yellow Submarine',
      style: 'Golden Ale',
      price: 180,
      stock: 0
    }
  ],

  constructor() { }

  ngOnInit() { }
}
```





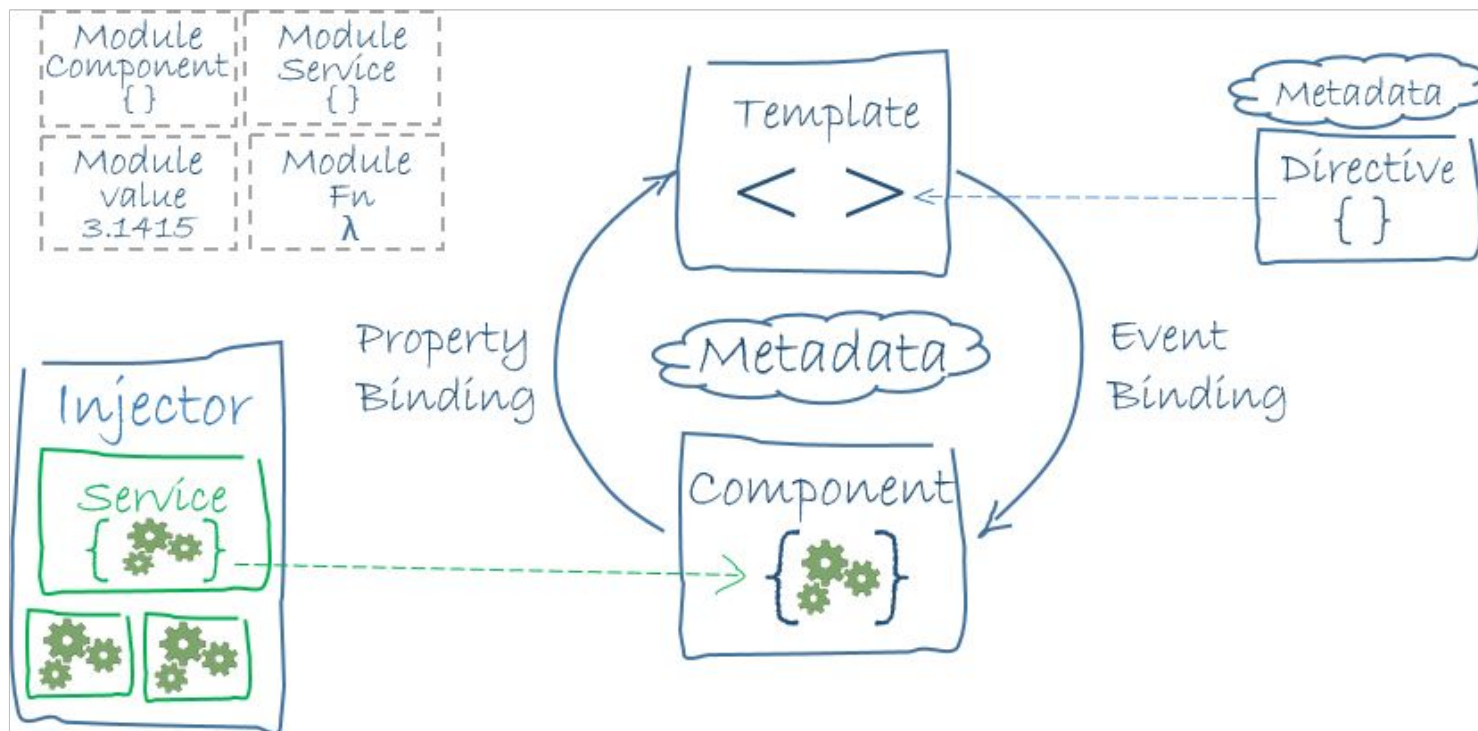
¿Cómo mostramos esa lista en el template del componente?



# Angular

## Directives

# Angular - Arquitectura





# Qué es una directiva?





# Directives en Angular

---

Es la forma de generar un template HTML dinámico.

- Existen 2 tipos de **directives**:
  - **Estructural**: Modifica el layout agregando, quitando o modificando elementos HTML. ( ejemplo \*ngIf \*ngFor)
  - **Atributos**: Modifica la apariencia y el comportamiento de elementos que ya existen. En el template parecen atributos comunes de HTML. (ejemplo [ngStyle])



# Vamos a usar directivas



# Queremos mostrar más de una cerveza:

---

## \*ngFor DIRECTIVE

La directiva \*ngFor repite el elemento por cada item en la colección.

```
<tr *ngFor="let beer of beers">
  <td>{{beer.name}}</td>
  <td>{{beer.style}}</td>
  <td>{{beer.price}}</td>
  <td>{{beer.stock}}</td>
</tr>
```



# Queremos mostrar si hay o no stock:



---

A nuestros clientes en realidad no les importa mucho qué cantidad de stock hay, sino simplemente saber si hay o no stock.

¿Cómo hacemos para mostrar un cartel si **no hay stock**?







# Ocultando o mostrando en el DOM

---

## \*ngIf

DIRECTIVE

La directiva \*ngIf, elimina o muestra una parte del DOM de acuerdo a la expresión que evalúa.

```
<tbody>
  <tr *ngFor="let beer of beers">
    <td><img [src]="beer.image" [alt]="beer.name"></td>
    <td>{{beer.name}}</td>
    <td>{{beer.type}}</td>
    <td>{{beer.price}}</td>
    <td *ngIf="beer.stock==0">Sin stock</td>
  </tr>
</tbody>
```





# Cervezas en Oferta

---

El cliente nos pide que marquemos con un recuadro verde las cervezas que están en oferta.

- Cosas que tenemos que hacer:
  - TBC



# Manipulando clases

---

## ngClass

DIRECTIVE

La directiva **ngClass** manipula las clases CSS de un elemento HTML

```
<tr *ngFor="let beer of beers" [ngClass]="{'clearance': beer.clearance}">
  <td>...</td>
  <td>...</td>
  <td>...</td>
</tr>
```

- Si `beer.clearance` es `true`, se le agrega la clase al elemento, si es `false` no.



# Manipulando clases

— — —

```
<some-element [ngClass]=" 'className1 'className2">  
  ...  
</some-element>
```

```
<some-element [ngClass]="['className1', 'className2']">  
  ...  
</some-element>
```

```
<some-element [ngClass]="{'className1': true, 'className2': false}">  
  ...  
</some-element>
```

# Angular

## Pipes

# Pipes en Angular



---

Los [pipes](#) son simples funciones que transforman valores que usamos en un template.

Angular tiene muchísimos **pipes** definidos, y además permite definir nuestros propios pipes.

Podemos usar **pipes** en los templates para:

- Formatear fechas
- Mostrar monedas
- Formatear números
- etc





# Queremos mostrar el precio y la fecha

---

## Ejemplos:

- Tengo un valor de fecha 1989-02-03, pero lo quiero mostrar en el template como '3 de Febrero de 1989'
  - Documentación [Date Pipe](#)
- Queremos mostrar un precio con el signo \$ adelante y con 2 decimales.
  - Documentación [Currency Pipe](#)



# Referencias

— — —

- [Angular.io - Getting Started](#) - Documentacion Oficial
- [Branch en el Repositorio](#)