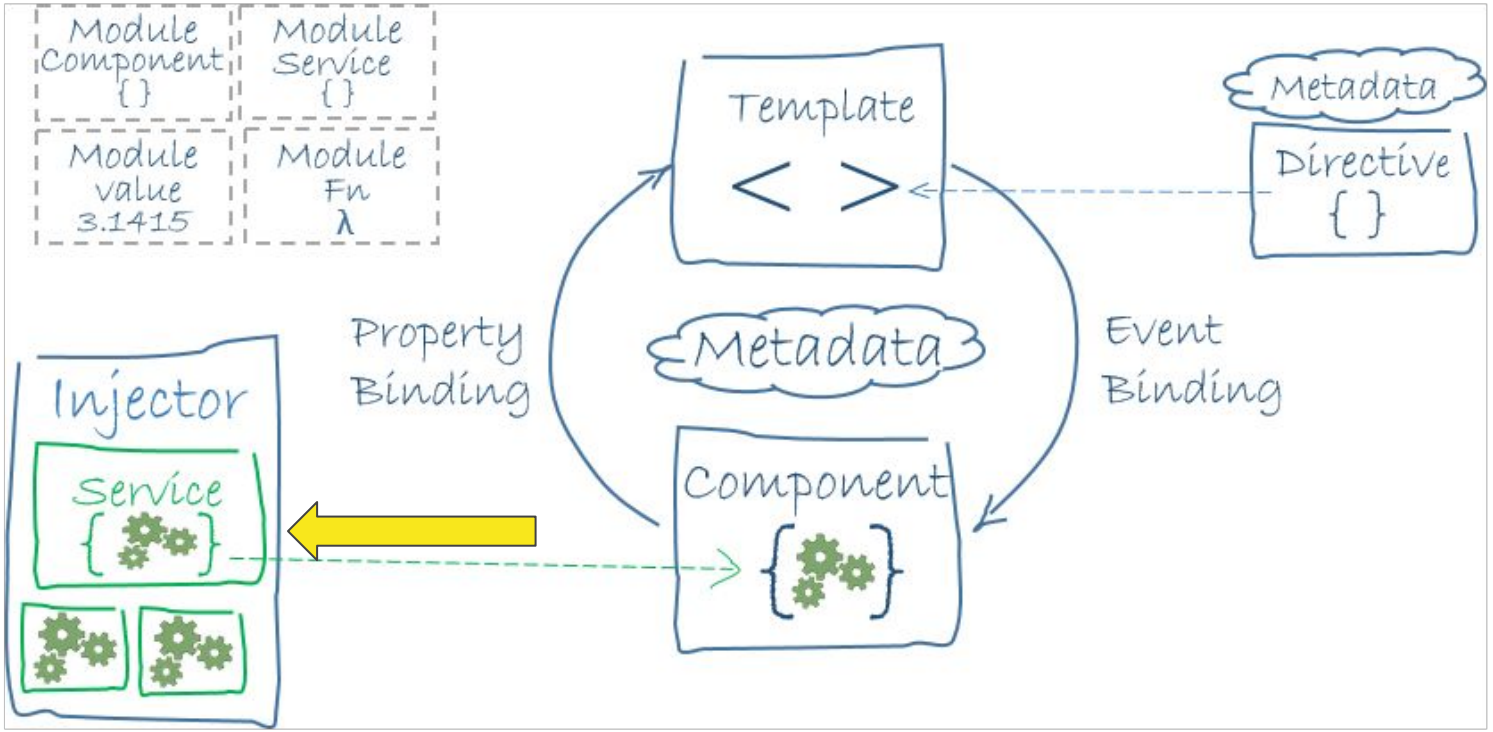
Angular

Services + Dependency Injection

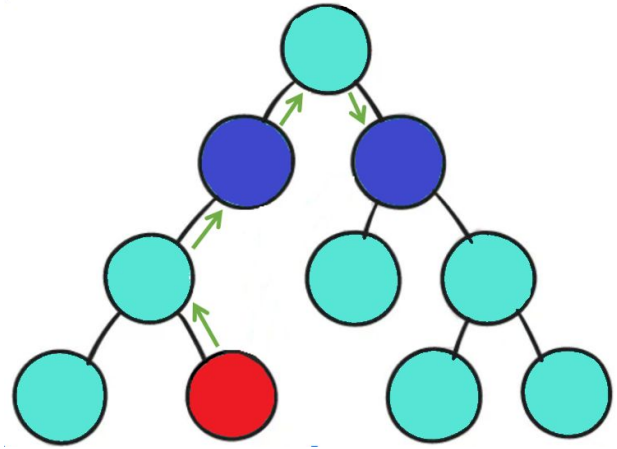
Angular - Arquitectura

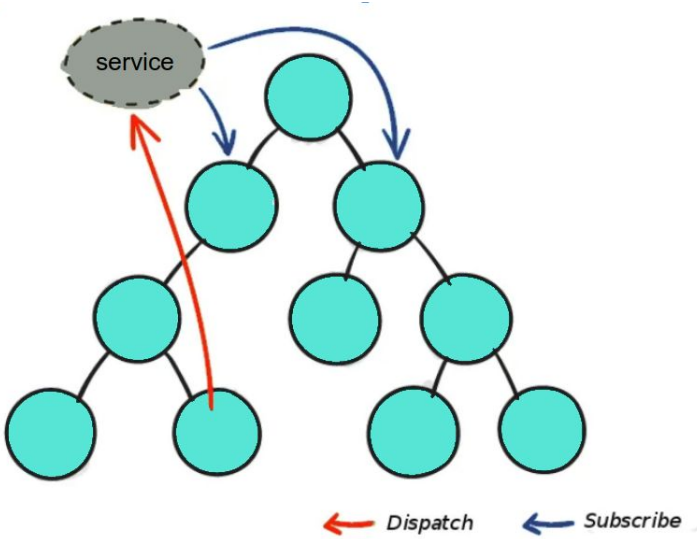


Comunicación de componentes mediante servicios?

Usar servicios para comunicar componentes

Vamos a usar un servicio para utilizarlo de “Store” (onda un almacenamiento centralizado) de los datos que queremos comunicar.





Qué son los Servicios?

Servicios

Son usados para organizar y compartir código en la aplicación.

“Un servicio es típicamente una clase con un propósito limitado y bien definido. Debería hacer algo específico y hacerlo bien.”

- En Angular usualmente se utilizan para separar la lógica del acceso a datos.
- Los servicios serán consumidos por los componentes para acceder a la funcionalidad que proveen.

https://angular.io/guide/architecture-services

Escribamos nuestro primer servicio

Servicio BeerDataService

Lo creamos con **angular-cli:**

- ng generate service **BeerCart** (ng g s **BeerCart**)

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

@Injectable({
  providedIn: 'root'
})
export class BeerCartService {
  constructor() { }
  /* TODO: acá va el código que manejará
  el carrito de cervezas */
}
```

beer-data.service.ts

Como lo usamos?

Creamos el Service con “new” en el componente?

- ✅ Desacoplamos los datos.
- ❌ Clases que usan el servicio tienen que saber como crearlo.
- ❌ Creamos una instancia del servicio cada vez que necesitemos datos.
- ❌ **No podemos compartir los mismos datos entre componentes.**

```
...
export class BeerListComponent implements OnInit {
  beers: Beer[] = [...];
  beerCartService: BeerCartService;
  constructor() { }
  ngOnInit(): void {
    this.beerCartService = new BeerCartService();
  }
...
```

beer-list.component.ts

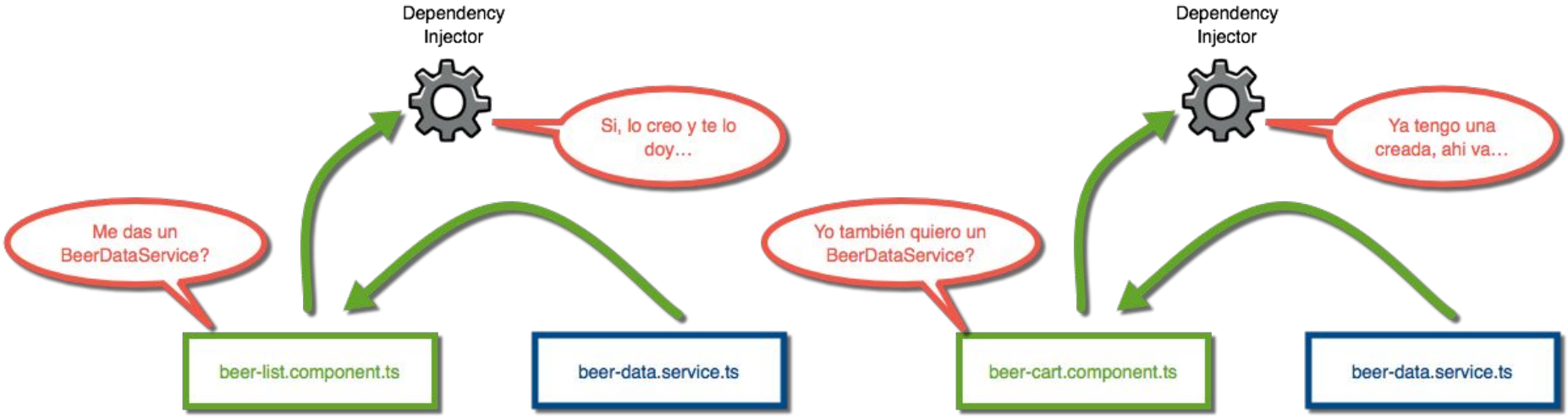
Inyección de dependencias (DI)

DI es un patrón a nivel de código mejora la modularidad del código. Es la forma por la cual podemos crear una instancia de una clase, sin saber cómo crearla.

- El dependency injector conoce como inyectar nuestras dependencias.
- **Inyectar**: Crea una clase (si es necesario) y la envía.
- **Dependencias**: Las clases de las que depende nuestro componente.

Esto se ve en más detalle en **Arquitecturas Web**

Dependency Injection



Primera vez

Segunda vez

Qué hacer para inyectar un servicio?

- Angular llama “Providers” a todo lo que se puede inyectar (son proveedores de servicios).
- El servicio tiene que tener el decorator

```
@Injectable({ providedIn: "root"} )
```

- Eso lo agrega en los providers del módulo (app.module.ts)
- Los sub componentes, pueden inyectar un BeerDataService.

Qué hacer para inyectar un servicio?

Refactoreamos **beer-list.component.ts**

```
...
import { BeerCartService } from '../beer-cart.service';

@Component({...})
export class BeerListComponent implements OnInit {

 beers: Beer[] = [...];

 constructor(private beerCartService: BeerCartService) { }

 addToCart(beer: Beer): void {
   ...
   this.beerCartService.llamarAlgo();
 }
```

Importamos el servicio

Inyectamos la dependencia del servicio.

Se crea una variable privada con una instancia del servicio.

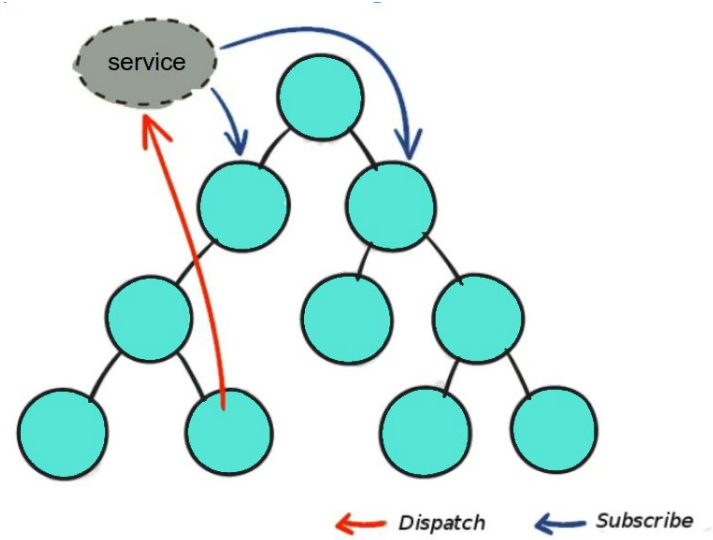
La usamos :)

Ventajas

- Ahora nuestra aplicación es más escalable y testeable
  - **Escalable**: Porque nuestra dependencia no está fuertemente atada a nuestras clases.
  - **Testeable**: Es mas facil de mockear servicios para testear nuestros componentes.
  - Flexible: En diferentes configuraciones podriamos usar diferentes subclases

Y con esto cómo comunicamos componentes?

Ahora vamos a usar la **comunicación por medio de un servicio.**



Agreguemos funcionalidad

- Qué tenemos que agregarle al servicio?
  - Una *lista de cervezas*
  - Función *agregar* al carrito

Actualizar **beer-list**

- Creamos una función addToCart(beer)
- La llamamos desde el click en el botón comprar.
- Desactivamos el botón de comprar si no hay stock.

```
import { BeerDataService } from '../beer-data.service';
import { CartService } from '../cart.service';

@Component({
  selector: 'beer-list',
  templateUrl: './beer-list.component.html',
  styleUrls: ['./beer-list.component.css']
})
export class BeerListComponent implements OnInit {
beers : Beer[];

  constructor(private beerDataService : BeerDataService, private cartService: CartService) { }

  addCart(beer: Beer) {
    this.cartService.addToCart(beer);
    beer.stock -= beer.quantity;
  }

}
```

Solución

Buscador

```
addToCart(beer: Beer) {
  let item: Beer = this.cartList.find((v1) => v1.name == beer.name);
  if(!item) {
    this.cartList.push({ ... beer});
  } else {
    item.quantity += beer.quantity;
  }
  console.log(this.cartList);
}
```

Clonar el objeto (notación funcional)

Cómo actualizamos el carrito de compras?

Actualizar **cart**

- En el carrito tenemos una lista de cervezas.
- Tenemos que hacer que esta lista, se sincronice con la lista del servicio.

Vamos a introducir algunos conceptos

Observable (Prog 2)

- Un objeto que es observable tiene la particularidad de que cuando cambia, puede emitir un evento que otros puedan escuchar.
- Si tengo una lista de cervezas y agrego una, me gustaría que el carrito escuche que cambio y actualice.
- El servicio tiene una lista de items observable y desde el carrito me suscribo para tener la info actualizada.

Más info:
http://reactivex.io/rxjs/manual/overview.html#observable

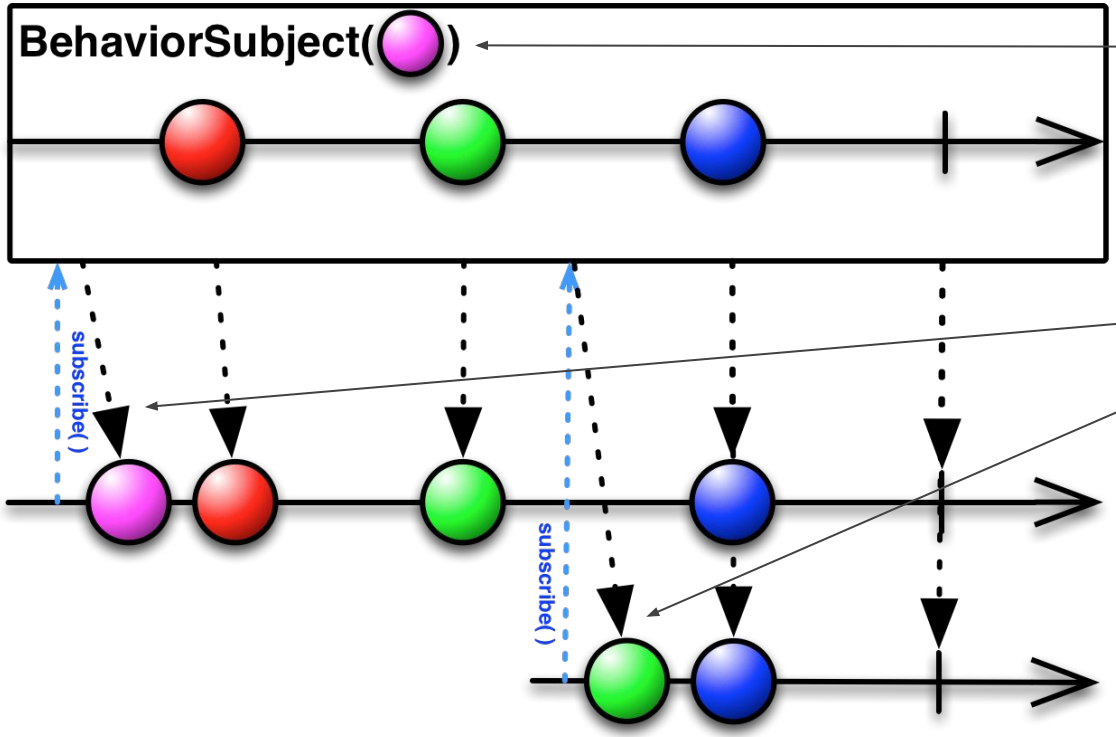

He sido creado!
Cada vez que cambie voy a disparar este evento
Uhh un nuevo evento, me voy a suscribir
Apa lala, ahora tengo una cerveza
Esa, consiguio una birra.
Hola!!
Observable
Observer

Behavior Subject

- Es un tipo de **clase** en rxjs (Reactive Extensions Javascript).
- Es un **observable**, hace de proxy (encapsula) el valor que se quiere observar.
- Puede manejar múltiples suscripciones a un único flujo.
- Emitir eventos cuando alguien se suscribe, y cada vez que cambia el valor.

Mas info:
http://reactivex.io/documentation/subject.html
http://www.introtorx.com/Content/v1.0.10621.0/02_KeyTypes.html#BehaviorSubject

Behavior Subject



Se construye con un valor inicial (por defecto)

Al suscribirse se manda el valor actual

Manos al código.

- En el servicio:
  - Incluyo Observable y BehaviorSubject
  - Creo un objeto BehaviorSubject
  - Creo una variable pública items, que es Observable.

```
import {BehaviorSubject, Observable} from "rxjs";

@Injectable()
export class CartService {

  private _items: Beer[] = [];
  private _itemsSubject: BehaviorSubject<Beer[]> = new BehaviorSubject(this._items);
  public items: Observable<Beer[]> = this._itemsSubject.asObservable();
```

Manos al código.

- También tengo que agregar al final de addToCart la llamada a next del BehaviorSubject

```
    this._items.forEach( (b: Beer) => {
      if (b.name == newBeer.name) {
        alreadyInCart = true;
        b.quantity += newBeer.quantity;
      }
    });

    // If beer doesnt exist in cart, then add it
    if (!alreadyInCart)
      this._items.push(newBeer);

    this._itemsSubject.next(this._items);
  }
```

Manos al código.

- Por último, nos suscribimos en el cart

```
beers = [];

constructor(private cartService: CartService) {
}

ngOnInit() {

  // Subscribe to cartService changes
  this.cartService.items.subscribe(data => {
    this.beers = data;
  });
}
```

Referencias

- Angular.io - Getting Started - Documentación Oficial
- Angular.io - Component Interaction
- Branch en el Repositorio