

# Tema de Investigación sobre la comunicación de componentes y los ciclos de vida.

Nombre : Juan Carlos Jara` `Curso: 3 nivel` `Materia: Aplicaciones Móviles` `Tema: Comunicacion de componentes y los ciclos de vida

## Comunicación de Componentes en Desarrollo de Software

La comunicación de componentes es crucial en el desarrollo de software, ya que los sistemas modernos suelen estar compuestos por múltiples componentes que necesitan interactuar de manera efectiva.

### Definición y Importancia de la Comunicación de Componentes

La comunicación de componentes en el desarrollo de software se refiere a cómo los diversos módulos, servicios o componentes de un sistema se comunican entre sí para lograr funcionalidades complejas y coherentes. Esta comunicación puede ocurrir a través de interfaces bien definidas, protocolos de comunicación, llamadas a API, eventos, entre otros mecanismos.

La importancia de una comunicación efectiva entre componentes radica en la capacidad de construir sistemas modulares, flexibles y mantenibles. Una comunicación clara y eficiente entre componentes facilita la escalabilidad, la reutilización de código, la depuración y la evolución continua del software.

También tenemos casos donde un componente hijo necesita enviar información a un componente padre. Los casos de uso de la comunicación entre componentes pueden ser innumerables, así que es necesario estudiarlo y comprenderlos para poder construir aplicaciones realmente potentes.

### De componente padre a hijo

Dentro de nuestro componente hijo necesitamos una librería llamada `Input` que nos permitirá utilizar un decorator mediante el cual declaramos las variable donde recibiremos los datos enviados por el componente padre.

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

@Component({
  ...
```

```

})
export class CardComponent{

    // Declaramos la variable que recibirá la variable enviada por el padre.
    @Input() character:any = {}

    constructor(){

    }

}

```

En el HTML del componente padre solo tenemos que crear una directiva personalizada del mismo nombre que la variable donde recibiremos los datos en el componente hijo.

## De componente hijo a padre

Para esto dentro de nuestro componente hijo utilizaremos dos librerías de Angular `Output` y `EventEmitter`. El mensaje será enviado a forma de evento emitido por el componente hijo. En el componente hijo usamos el siguiente código que pasamos a explicar a continuación.

```

import { Component, Output, EventEmitter } from "@angular/core";

@Component({
  selector: "app-hijo",
  templateUrl: "./hijo.component.html",
})
export class HijoComponent {
  // Creamos la variable a enviar al padre
  @Output() mensaje: EventEmitter<string>;

  constructor() {
    // Inicializamos la emision de eventos
    this.mensaje = new EventEmitter();
  }

  emitirMensaje() {
    // Usando la variable emitimos el valor que queremos enviar
    this.mensaje.emit("Gaaaaaa Este es el mensaje");
  }
}

```

```
}  
}
```

Ahora dentro del código HTML del componente Hijo utilizamos el evento click para ejecutar la función `emitirMensaje()`.

```
<div>  
  <h1>Hijo</h1>  
  <button (click)="emitirMensaje()">click me</button>  
</div>
```

Ya emitimos el mensaje ahora solo falta que el componente padre lo reciba para esto debemos crear una función dentro del componente padre de la siguiente manera.

```
msj:string;  
  
recibirMensaje(mensaje:string){  
  this.msj = mensaje;  
}
```

Ahora debemos ejecutar esta función pero en el parámetro obtener el evento emitido por el componente hijo, para esto hacemos lo siguiente.

```
<app-hijo (mensaje)="recibirMensaje( $event )"></app-hijo>
```

En este código `(mensaje)` es el nombre de la variable output del componente hijo y `$event` es la captura del `.emit()` que enviamos en el componente hijo.

### 3. Rutas con parámetros

Las rutas con parámetro son aquellas en las que dentro de la ruta se envía un parámetro determinado para que el componente lo procese o muestre alguna información relacionada a dicho parámetro. Un ejemplo de ruta con parámetro sería el siguiente.

```
https://www.miweb.com/personaje/1
```

Para poder trabajar con este tipo de rutas lo principal es obtener dicha ruta en el componente que este abre, de tal modo realizar alguna lógica con ese parámetro.

Para poder realizar esto tenemos que cambiar un poco la ruta dentro de nuestro archivo de rutas `app.routes.ts` de tal forma que pueda aceptar algún parámetro. El código de la ruta acepta un parámetro `id` pero abre un único componente. Ese componente es el que se encargará de obtener el parámetro para procesarlo.

```
{ path: 'character/:id', component: CharacterComponent }
```

Ahora para poder obtener la ruta desde un componente tenemos que hacer uso de una librería llamada `ActivatedRoute` que mediante el constructor de nuestro componente obtendremos la ruta activa y su respectivo parámetro.

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

@Component({
  ...
})
export class CharacterComponent {
  constructor( private activatedRoute : ActivatedRoute){
    console.log(activatedRoute);
  }
}
```

El objeto `ActivatedRoute` contiene información acerca de la ruta que está activa en el navegador. Este objeto tiene la propiedad `params` que a su vez tiene información sobre los parámetros enviados a la ruta. Sin embargo, si bien podemos ver esta información usando un `console.log()` no es posible ingresar al valor de la ruta de manera directa.

Si buscamos en la documentación de Angular descubriremos que el `params` es un objeto de la interface `Observable<Params>`. Pues bien, la forma de obtener este parámetro será utilizando un método presente en un `Observable` llamado `subscribe` al cual enviaremos como parámetro una función flecha para obtener el parámetro como se muestra en el código siguiente.

```
constructor( private activatedRoute : ActivatedRoute){
  activatedRoute.params.subscribe( prm => {
    console.log(`El id es: ${prm['id']}`);
  })
}
```

**Listo!!** Ya sabemos crear rutas con parámetros y recibir el parámetro desde un componente. Ahora lo único que nos falta para terminar esta sección es saber como ingresar a una ruta enviando un determinado parámetro. Para esto existen dos métodos. El primero es desde el HTML y el segundo es desde el TypeScript.

## Utilizando el HTML

Para dirigirnos a una ruta desde el HTML es bastante sencillo, solo es necesario enviar un parámetro por la directiva `routerLink` de la siguiente manera.

```
<button [routerLink]="['/character', '1']">Click me</button>
```

Utilizamos `/` porque queremos que la nueva ruta comience desde la raíz de nuestra web y no de `rutaactual/character/1`. También debes tener en cuenta que el parámetro es un string.

## Utilizando el TypeScript

Ahora para dirigirnos a una ruta desde el TypeScript tenemos que hacer uso de la librería `Router` en el TypeScript del componente para luego utilizar el método `navigate` de la siguiente forma.

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

@Component({
  ...
})
export class CharactersComponent{

  verPagina(id: number){
    this.router.navigate(['character', id])
  }

  constructor(private router:Router){

  }

}
```

Luego solo debemos ejecutar el método `verPagina(1)` desde alguna etiqueta HTML con la directiva `(Click)`.

**Listo!!** Ya sabemos todo lo necesario para utilizar rutas con parámetros en nuestros proyectos Angular.

#### 4. Los servicios

Los servicios son muy importantes en Angular, estos nos permiten brindar información al componente que lo requiera. Podemos utilizarlos para mantener la persistencia de datos en nuestra aplicación web o realizar peticiones asíncronas hacia alguna API. Además como son independiente a los componentes son totalmente reutilizables.

Los servicios en Angular tenemos que crearlos dentro de la carpeta `src/app/services`. Dentro de la carpeta usaremos la convención `datos.service.ts` para los archivos de los servicios. Dentro de estos archivos crearemos la siguiente estructura de código.

```
// Importamos la libreria necesaria para que los servicios puedan ser usados
en Angular mediante la inyección de dependencias
import { Injectable } from "@angular/core";

// Usamos el decorator que le indica a Angular que esta clase podrá ser
inyectada en cualquier parte del proyecto
@Injectable()
// Creamos la clase para nuestro servicio
export class DataService {
  constructor() {
    console.log("Servicio listo para usarse");
  }
}
```

Ahora tenemos que agregar nuestro servicio al archivo `app.module.ts` para que Angular lo pueda reconocerlo y podamos usarlo en nuestros componentes. Para esto debemos importar el servicio y luego colocarlo en la propiedad `providers`.

```
// Importando el servicio
import { HeroesService } from '../services/heroes.service';

@NgModule({
  ...
  providers: [
    HeroesService
  ],
  ...
})
```

```
})  
export class AppModule { }
```

Para poder utilizar un servicio dentro de un componente primero debemos importarlo luego debemos declararlo dentro del constructor de la siguiente manera.

```
import { Component } from '@angular/core';  
// Importamos el servicio  
import { DataService } from '../services/data.service';  
  
@Component({  
  ...  
})  
export class CompComponent {  
  
  // Enviamos el servicio como parametro al constructor de nuestro componente  
  constructor(private dataService:DataService) {  
  
  }  
}
```

Ahora ya estamos usando el servicio y lo podemos comprobar si dentro del servicio colocamos un `console.log()` y vemos en la consola lo que sucede cuando ejecutamos el componente. En el siguiente ejemplo, vamos como utilizarlo creando un un arreglo de objetos estático con una interface y obteniendo los datos desde algún componente.

```
@Injectable()  
export class HeroesService {  
  
  private heroes:Heroe[] = [  
    {  
      nombre: "IronMan",  
      bio: "Es uno de los Avenger más conocidos",  
    },  
    {  
      nombre: "SpiderMan",  
      bio: "Es el increíble hombre araña",  
    }  
  ];  
  
  constructor(){
```

```

    }

    public getHeroes():Heroe[]{
        return this.heroes;
    }
}

```

```

export interface Heroe{
    nombre: string;
    bio: string;
}

```

```

// Importamos el servicio y la interface
import { ServService, Heroe } from '../../services/heroes.service';

// Usamos la interface para crear el arreglo de heres
heroes:Heroe[] = [];

constructor(private _heroesService:HeroesService) {

}

// El OnInit se ejecuta cuando el constructor termina de renderizarse
ngOnInit(): void {
    this.heroes = this._heroesService.getHeroes();
    console.log(this.heroes);
}

```

## Ciclos de Vida en el Desarrollo de Software

Los ciclos de vida en el desarrollo de software describen las etapas y actividades que se llevan a cabo desde la concepción de un proyecto hasta su entrega, mantenimiento y eventual desuso. Aquí hay una investigación sobre los ciclos de vida junto con algunas referencias bibliográficas destacadas:

### Importancia y Modelos de Ciclos de Vida

Los ciclos de vida en el desarrollo de software son fundamentales para la planificación, ejecución y control de proyectos de software. Diferentes modelos de ciclo de vida, como el



modelo en cascada, el modelo en espiral, los modelos iterativos e incrementales, entre otros, ofrecen enfoques variados para abordar las complejidades del desarrollo de software.

Comprender y seleccionar el modelo de ciclo de vida adecuado para un proyecto específico es crucial para garantizar la calidad, la eficiencia y la satisfacción del cliente a lo largo del proceso de desarrollo.



#### Referencias Bibliográficas

**"Software Engineering: A Practitioner's Approach"** de Roger S. Pressman

**"Agile Software Development: Principles, Patterns, and Practices"** de Robert C. Martin y Micah Martin.

**Component-Based Software Engineering: Putting the Pieces Together"** de George T. Heineman y William T. Councill.