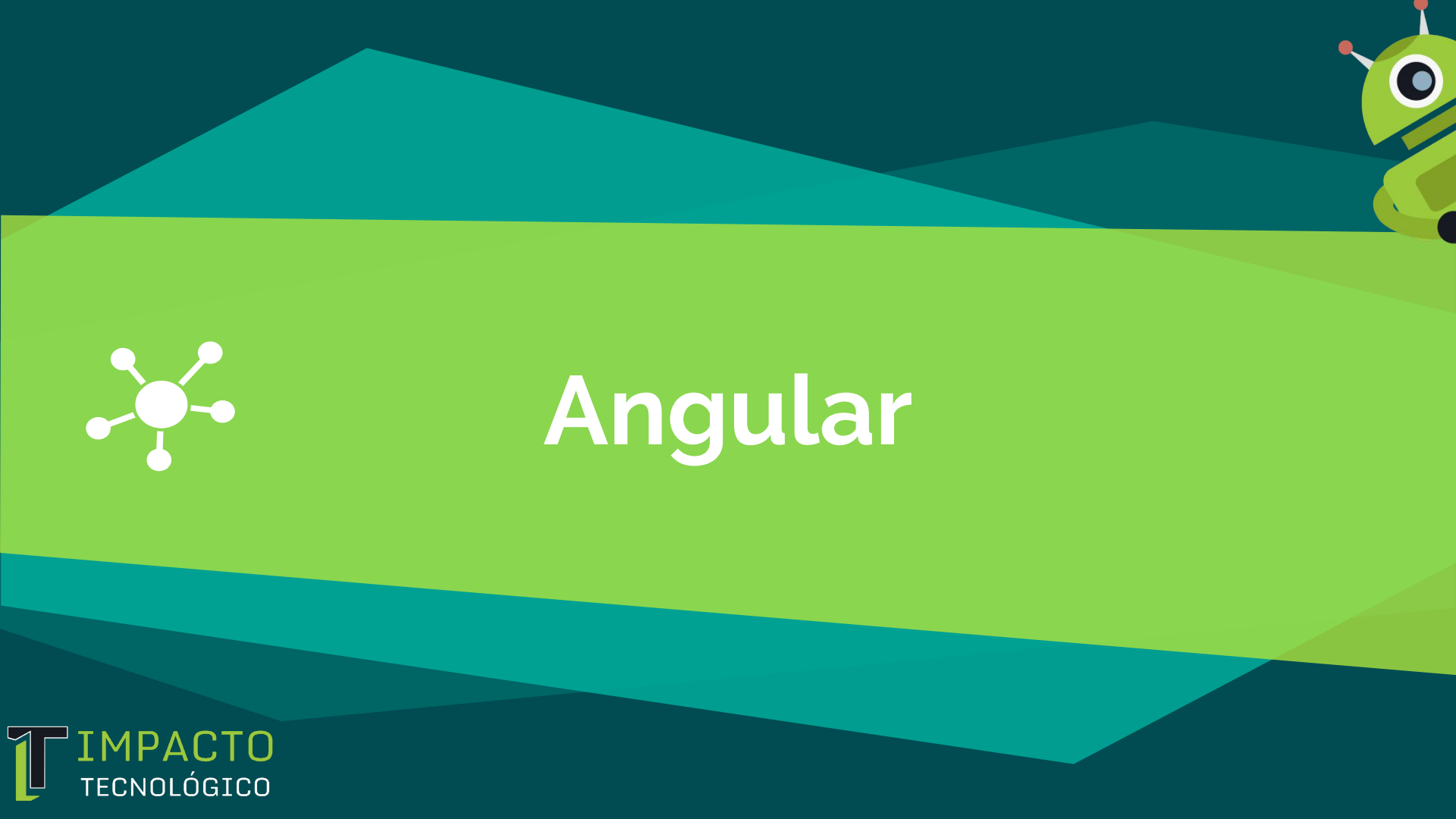
IMPACT
A<ADEMY>


Angular
IMPACTO
TECNOLÓGICO

IMPACTO TECNOLÓGICO

Hola!

Soy Pedro Plasencia

Full Stack Web Developer

@pedrovelasquez9 – pjpv9011@gmail.com

1.

Configuración de entorno

Manejo de entornos con Angular

¿Para qué podemos usar los entornos?

Los entornos se definen en archivos, por defecto, Angular maneja 2, un entorno de desarrollo y uno para producción donde podemos definir características y propiedades cambiantes de nuestra aplicación.

Manejo de entornos con Angular

Demostración práctica, accediendo a propiedades de un entorno de desarrollo.

Manejo de entornos con Angular

Agregando más entornos.
Para manejar más entornos en la app, debemos seguir los siguientes pasos:

- Crear un archivo environment, por ejemplo, environment.qa.ts

```
export const environment = {
  production: true
  APIEndpoint: "https://google.com"
};
```

Manejo de entornos con Angular

Una vez tengamos en environment, debemos indicar a Angular que se trata de un ambiente en nuestro angular.json

Una vez definidos los ambientes que manejaremos en la app, podemos crear builds para cada uno con el flag --configuration:

ng build --configuration=qa

ng build (dev por defecto)

ng build --prod (usa el env para producción)

Manejo de entornos con Angular

Nota: los ambientes en angular se manejan de esta forma y no con, por ejemplo, un .env ya que los process.env son propios de aplicaciones que se ejecutan en base a nodejs.

Bonus:

cambiando el base href

Manejo de entornos con Angular

Si, en vez de generar un build lo que quiero es servir mi aplicación en local para probar la configuración de cada ambiente, debo definir esta configuración en el objeto serve->configurations de mi angular.json, una vez hecho esto, podemos ejecutar:

ng serve -c=qa

Y nuestra aplicación se servirá con la configuración del ambiente nuevo sin necesidad de generar un build.

2.

Cambiando variables en caliente

Manejo de parámetros dinámicos

Si bien, como hemos visto, los entornos en angular sirven para definir ciertos parámetros propios de cada uno, quizás se dé el caso en el que debamos cambiar parámetros “en caliente”, aunque no es recomendado, una solución para realizar este tipo de cambios es sirviendo archivos de propiedades a través de servicios.

Por ejemplo, creemos una nueva app con el CLI de angular y, en assets, definamos un archivo JSON con las propiedades que queramos.

Manejo de parámetros dinámicos

Una vez tengamos nuestro json con lo que queremos parametrizar, lo leeremos a través de una llamada http que lo cargue a nuestro componente:

```
constructor(private http: HttpClient) {
   this.getJSON().subscribe(data => {
     console.log(data);
   });
 }
 public getJSON(): Observable<any> {
   return this.http.get(this._jsonURL);
 }
```

Manejo de parámetros dinámicos

Nota: no olvidar importar el módulo http en el módulo de la app.

Una vez tengamos esto listo, podemos generar un build y probar el cambio dinámico de propiedades desde el json cada vez que recargamos la app.

3.
Comunicación entre componentes

Comunicación entre componentes

A medida que se desarrolla una app con Angular será necesario que algunos de nuestros componentes se comuniquen, la medida estándar para transferir datos entre componentes es a través de servicios usando getters y setters u objetos a los que ambos tengan acceso (localStorage, etc.).

Sin embargo, cuando tenemos una relación “más estrecha” entre componentes que comparten un flujo de trabajo, podemos optar por ciertos métodos para la comunicación entre componentes padre e hijo.

Comunicación entre componentes

La que más se suele utilizar. Consiste en usar la etiqueta @Input de Angular. Esta etiqueta se pone en el componente hijo para indicar que esa variable proviene desde fuera del componente, es decir desde el componente padre usamos el selector del hijo para incluirlo en el html y le pasamos el valor que queremos pasar al hijo:

Comunicación entre componentes

Componente hijo:

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

@Component({
  selector: 'app-child',
  template: `
     Message from parent:
  `,
  styleUrls: ['./child.component.css']
})
export class ChildComponent {

  @Input() childMessage: string;

  constructor() { }
```

Comunicación entre componentes

Componente padre:

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

@Component({
  selector: 'app-parent',
  template: `
    <app-child [childMessage]="parentMessage"></app-child>
  `,
  styleUrls: ['./parent.component.css']
})
export class ParentComponent{
  parentMessage = "message from parent"
  constructor() { }
}
```

Comunicación entre componentes

Otro método usado es el ViewChild. Mediante ViewChild, el padre crea el componente hijo y tiene acceso a sus datos y atributos:

Componente hijo

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

@Component({
  selector: 'app-child',
  template: `
  `,
  styleUrls: ['./child.component.css']
})
export class ChildComponent {

  message: string = "Hola Mundo!"

  constructor() { }
```

Comunicación entre componentes

Componente padre:

```
import { Component, ViewChild, AfterViewInit } from '@angular/core';
import { ChildComponent } from "../child/child.component";
@Component({
  selector: 'app-parent',
  template: `
    Message:
    <app-child></app-child>
  `,
  styleUrls: ['./parent.component.css']
})
export class ParentComponent implements AfterViewInit {

  @ViewChild(ChildComponent) child;

  constructor() { }

  message:string;

  ngAfterViewInit() {
    this.message = this.child.message
  }
}
```

Comunicación entre componentes

La particularidad de éste método es que tenemos que esperar a que la vista esté totalmente cargada para acceder a los atributos del hijo. Para ello creamos un método de Angular llamado ngAfterViewInit() en el que simplemente inicializamos la variable con el valor del atributo del hijo (el hijo lo declaramos como @ViewChild(ChildComponent)).

Comunicación entre componentes

Comunicación desde el componente padre al hijo mediante output y eventos

Éste método es útil cuando queremos queremos informar de cambios en los datos desde el hijo, por ejemplo, si tenemos un botón en el componente hijo y queremos actualizar los datos del padre.

Comunicación entre componentes

Componente hijo:

```
import { Component, Output, EventEmitter } from '@angular/core';
@Component({
  selector: 'app-child',
  template: `
     <button (click)="sendMessage()">Send Message</button>
  `,
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  message: string = "Hola Mundo!"

  @Output() messageEvent = new EventEmitter<string>();
  constructor() { }
  sendMessage() {
    this.messageEvent.emit(this.message)
  }
}
```

Comunicación entre componentes

Componente padre:

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

@Component({
  selector: 'app-parent',
  template: `
    Message:
    <app-child (messageEvent)="receiveMessage($event)"></app-child>
  `,
  styleUrls: ['./parent.component.css']
})
export class ParentComponent {

  constructor() { }
  message:string;

  receiveMessage($event) {
    this.message = $event
  }
}
```

Comunicación entre componentes

En el hijo declaramos un evento de tipo EventEmitter y con la pulsación del botón ejecutamos un método para lanzar el evento al padre. Desde el padre creamos una función para recibir el mensaje desde el evento y incluimos en la etiqueta del html (messageEvent)=”receiveMessage($event) para conectar el evento al método que hemos creado.

4.
Desarrollo de directivas

Desarrollo de directivas

El concepto de Angular custom directive es uno de los conceptos que tenemos que conocer de Angular ya que nos permite extender la funcionalidad del framework de una forma muy sencilla. ¿Para que sirve un Angular custom directive? . Sirve como su nombre indica para construir una directiva personalizada es decir en vez de usar una directiva existente como ngClass o *ngIf podemos construir nuestras propias directivas y aumentar la extensibilidad de nuestro código.

Desarrollo de directivas

Angular soporta dos tipos de custom directive attribute directive y structure directive. Cada una de las cuales asume una responsabilidad diferente . Vamos a construir un ejemplo de cada una de ellas para poder entenderlas mejor.

Desarrollo de directivas

Angular Custom Directive (Attribute)

Lo primero que vamos a hacer es construir un custom directive de tipo atributo que nos ilumine el texto de cualquier etiqueta. El primer paso es usar el cliente de angular y hacer

ng generate directive destacar

Esto nos construira una nueva directiva con un selector por defecto “appDestacar”

Desarrollo de directivas

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

@Directive({
  selector: '[appDestacar]'
})
export class DestacarDirective {

  constructor() { }

}
```

Desarrollo de directivas

Acabamos de construir una directiva personalizada pero en estos momentos la directiva no hace nada . Es momento de modificar el constructor y añadir el código necesario para que la directiva se encarga de iluminar nuestro código.

```
constructor(private el:ElementRef, private renderer:Renderer) {
    renderer.setElementStyle(el.nativeElement,'fontSize','50px');
    renderer.setElementStyle(el.nativeElement,'color','blue');
    renderer.setElementStyle(el.nativeElement,'border','2px solid blue');
   }
```

Desarrollo de directivas

En este bloque de código la directiva usa a nivel de constructor dos variables el y renderer . La primera de ellas nos da acceso de forma directa al elemento en el cual la directiva se va a aplicar. El segundo parámetro es el objeto renderer que permite aplicar estilos al elemento que manipulados.

Desarrollo de directivas

En este caso nos queda aplicar la directiva a cualquier etiqueta

```
<p appDestacar>
componente hola
</p>
```

Desarrollo de directivas

Angular Custom Directive (Structural)

Las directivas estructurales funcionan de otra forma ya que son capaces de cambiar por completo el elemento en el cual se aplican. Ejemplos de directivas estructurales son *ngFor y *ngIf. Vamos a proceder a construir una directiva estructural. Para ello ejecutamos el comando:

ng generate directive multiplicar

Esto nos genera una nueva directiva vacía que nosotros deberemos rellenar en este caso la diferencia es que vamos a acceder a través del constructor a la plantilla a la que la directiva puede acceder .Es decir a la plantilla interna de la etiqueta y que nosotros podemos necesitar modificar:

Desarrollo de directivas

```
import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
@Directive({
  selector: '[appMultiplicar]'
})
export class MultiplicarDirective {
  constructor(private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) {
  }

  @Input() set appMultiplicar(numero: number) {
    for (var i = 0; i < numero; i++)
      // If condition is true add template to DOM
      this.viewContainer.createEmbeddedView(this.templateRef);
  }
}
```

Desarrollo de directivas

En este caso lo que estamos haciendo son varias cosas en primer lugar permitir que la directiva tenga un parámetro de entrada utilizando @Input. Este parámetro de entrada permitirá que la directiva pueda variar su comportamiento de forma dinámica. En nuestro caso la directiva usa el parámetro para realizar un bucle for y apoyándose en la plantilla por defecto multiplicar su contenido n veces.

Vamos a ver como la directiva se puede aplicar en un componente:

```
<p *appMultiplicar=”5”>
componente hola
</p>
```

5.
Deployments remotos

Deployment remoto

https://angular.io/guide/deployment

IMPACT
A<ADEMY>