Angualar 17

Directivas estructurales nuevas

@for, @if, @switch sustituyen a ngfor, ngif, ngswitch

Ejemplo de uso de @for

```
@for(let i=0; i<10; i++)
  <div>{{i}}</div>
@empty
  <div>No hay elementos</div>
```

Ejemplo de uso de @if con @else if y @else

```
@if(condicion)
  <div>Verdadero</div>
@else if(otraCondicion)
    <div>Verdadero</div>
@else
    <div>Falso</div>
```

Ejemplo de uso de @switch

```
@switch(condicion)
  @case(1)
    <div>Uno</div>
  @case(2)
    <div>Dos</div>
  @default
    <div>Otro</div>
```

Vistas diferibles

Aprovechando la nueva sintaxis de bloques (@for, @if, @switch), hay un nuevo mecanismo para hacer tus aplicaciones más rápidas. **@defer** la carga diferida declarativa y potente con una ergonomía sin precedentes.

Supongamos una aplicación de un blog y queremos cargar de forma diferida los comentarios de los usuarios. Actualmente, habría que usar ViewContainerRef mientras también se gestiona toda la complejidad para limpiar, manejar errores de carga, mostrar un marcador de posición, etc. Cuidar de varios casos especiales puede resultar en algún código no trivial, que será difícil de probar y depurar.

Las nuevas vistas diferibles, te permiten cargar perezosamente la lista de comentarios y todas sus dependencias transitivas con una sola línea de código declarativo:

```
@defer{
  <app-comments [postId]="postId"></app-comments>
}
@loading{
  <div>Loading comments...</div>
}
@error{
  <div>Failed to load comments</div>
}
@placeholder{
  <div>No comments yet</div>
}
```

Las vistas diferibles ofrecen algunos disparadores adicionales:

- **on idle:** carga la vista cuando el navegador está inactivo
- **on immediate:** carga la vista inmediatamente sin bloquear el navegador
- **on timer(<time>):** carga la vista después de un tiempo especificado
- **on viewport and on viewport(<ref>):** carga la vista cuando el elemento está visible en el viewport.
- **on interaction and on interaction(<ref>):** muestra la vista cuando el usuario interactúa con un elemento
- **on hover and on hover(<ref>):** lanza la carga diferida cuando el usuario pasa el ratón por encima de un elemento
- **when <expr>:** te permite especificar tu propia condición a través de una expresión booleana

*Un ejemplo de uso de **on viewport**:*

```
@defer (on viewport){
  <app-comments [postId]="postId"></app-comments>
}
@loading{
  <div>Loading comments...</div>
}
@error{
  <div>Failed to load comments</div>
}
@placeholder{
  <div>No comments yet</div>
}
```

Nuevos métodos de ciclo de vida

Para mejorar el rendimiento de Angular SSR y SSG, a largo plazo nos gustaría alejarnos de la emulación del DOM y de las manipulaciones directas del DOM. Al mismo tiempo, a lo largo del ciclo de vida de la mayoría

de las aplicaciones, es necesario interactuar con los elementos para instanciar bibliotecas de terceros, medir el tamaño de los elementos, etc.

Para conseguir ésto, se han desarrollado un conjunto de nuevos ganchos de ciclo de vida:

- **afterNextRender:** registra un callback para ser invocado la próxima vez que la aplicación termine de renderizar
- **afterNextRender:** registra un callback para ser invocado la próxima vez que la aplicación termine de renderizar

Únicamente el navegador invocará estos ganchos, lo que te permite enchufar lógica DOM personalizada de forma segura directamente dentro de tus componentes.

Por ejemplo, puedes usar afterNextRender:

```
@Component({
  selector: 'my-chart-cmp',
  template: `<div #chart>{{ ... }}</div>`,
})
export class MyChartCmp {
  @ViewChild('chart') chartRef: ElementRef;
  chart: MyChart|null;

  constructor() {
    afterNextRender(() => {
      this.chart = new MyChart(this.chartRef.nativeElement);
    }, {phase: AfterRenderPhase.Write});
  }
}
```

Signals

Un signal es un contenedor de un valor que puede notificar a los consumidores interesados cuando ese valor cambia. Los signals pueden contener cualquier valor, desde primitivos simples hasta estructuras de datos complejas.

El valor de un signal siempre se lee a través de una función getter, lo que permite a Angular rastrear dónde se usa el signal.

Los **signals** pueden ser de solo lectura o de escritura.

Writable signals

Los writable signals proporcionan una API para actualizar sus valores directamente.

Se crean writable signals llamando a la función de señal con el valor inicial de la señal:

```
const count = signal(0);
// set the value of the signal
count.set(1);
// get the value of the signal
console.log(count()); // 1
// use update()
count.update((value) => value + 1);
console.log(count()); // 2
```

Computed signals

Una computed signal deriva su valor de otros signals.

Podemos definir una señal computada utilizando la función computed y especificando una función de derivación:

```
const count: WriteableSignal<number> = signal(0);
const doubleCount = computed(() => count() * 2);
console.log(doubleCount()); // 0
count.set(1);
console.log(doubleCount()); // 2
```

Otro ejemplo de señal computada:

```
const showCount = signal(false);
const count = signal(0);
const conditionalCount = computed(() => {
  if (showCount()) {
    return `The count is ${count()}.`;
  } else {
    return 'Nothing to see here!';
  }
});
console.log(conditionalCount()); // 'Nothing to see here!'
showCount.set(true);
console.log(conditionalCount()); // 'The count is 0.'
count.set(42);
console.log(conditionalCount()); // 'The count is 42.'
```

Effects

Un effect es una operación que se ejecuta cada vez que cambia uno o más valores de señal. Los effects siempre se ejecutan al menos una vez. Cuando se ejecuta un efect, realiza un seguimiento de cualquier lectura de valores de señal. Si cambia alguno de estos valores de señal, el efect se ejecuta de nuevo. Al igual que las señales computadas, los efects realizan un seguimiento de sus dependencias de forma dinámica y solo realizan un seguimiento de las señales que se leyeron en la ejecución más reciente.

Se puede crear un efecto con la función effect:

```
const count = signal(0);
const doubleCount = computed(() => count() * 2);
const effect = effect(() => {
  console.log(`The count is ${count()} and the double count is ${doubleCount()}.`);
});
count.set(1);
// The count is 1 and the double count is 2.
count.set(2);
// The count is 2 and the double count is 4.
```

Por defecto, registrar un nuevo efecto con la función effect() requiere un contexto de inyección (acceso a la función de inyección). La forma más sencilla de proporcionar esto es llamar a effect **dentro de un constructor** de componente, directiva o servicio.

Un ejemplo de uso de effect en un componente:

```
@Component({
  selector: 'my-component',
  template: `...`,
})
export class MyComponent {
  constructor() {
    const count = signal(0);
    const doubleCount = computed(() => count() * 2);
    const effect = effect(() => {
      console.log(`The count is ${count()} and the double count is ${doubleCount()}.`);
    });
    count.set(1);
    // The count is 1 and the double count is 2.
    count.set(2);
    // The count is 2 and the double count is 4.
  }
}
```

Casos de uso comunes para los efectos:

- Mostrar datos en la consola y cuando cambian, ya sea para análisis o como una herramienta de depuración
- Mantenimiento de datos sincronizados con window.localStorage
- Aádir comportamiento de DOM personalizado que no se puede expresar con la sintaxis de plantillas
- Realizar renderizado personalizado en un <canvas>, biblioteca de gráficos u otra biblioteca de interfaz de usuario de terceros