Angualar 17

Directivas estructurales nuevas

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

Ejemplo de uso de @for

```
@for(card of cards; track $index)
  <div>{{card}}</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:

- **afterRender:** 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

Angular SSR

SSR es un proceso que implica renderizar páginas en el servidor, lo que resulta en un contenido HTML inicial que contiene el estado de la página inicial. Una vez que el contenido HTML se entrega a un navegador, Angular inicializa la aplicación y utiliza los datos contenidos en el HTML. En Angular 17 se han realizado mejoras en el rendimiento de Angular SSR y SSG, que viene de Angular Universal.

Ventajas de Angular SSR:

- Mejora el rendimiento de la aplicación, especialmente en dispositivos móviles y conexiones lentas
- Mejora la experiencia del usuario con tiempo de carga más rápido
- Mejora el SEO de la aplicación al permitir que los motores de búsqueda indexen el contenido de la aplicación

Se puede añadir Angular SSR a una aplicación Angular existente con el comando ng add:

```
ng add @angular/ssr
```

Este comando hace modificaciones en la aplicación Angular existente para añadir Angular SSR. Algunos de los cambios que se realizan son:

- **server.ts:** el servidor de Node.js que se utiliza para renderizar la aplicación Angular en el servidor
- **angular.json:** el archivo de configuración de Angular que se utiliza para configurar la aplicación Angular
- **/src/main.server.ts:** el archivo que se utiliza para inicializar la aplicación Angular en el servidor
- **/src/app/app.config.server.ts:** el módulo de la aplicación Angular que se utiliza para inicializar la aplicación Angular en el servidor

Configuración de Angular SSR

El archivo server.ts configura un servidor Node.js Express y el renderizado del lado del servidor de Angular. CommonEngine se utiliza para renderizar una aplicación Angular.

Un ejemplo de server.ts:

```
// All regular routes use the Angular engine
server.get('*', (req, res, next) => {
  const {protocol, originalUrl, baseUrl, headers} = req;

  commonEngine
      .render({
        bootstrap,
        documentFilePath: indexHtml,
        url: `${protocol}://${headers.host}${originalUrl}`,
        publicPath: browserDistFolder,
        providers: [{provide: APP_BASE_HREF, useValue: req.baseUrl}],
      })
      .then((html) => res.send(html))
      .catch((err) => next(err));
});
```

En este ejemplo:

- bootstrap: la función de inicialización de la aplicación Angular
- documentFilePath: la ruta del archivo index.html
- document: el DOM inicial que se utilizará para inicializar la aplicación del servidor.
- url: la URL de la solicitud
- publicPath: la ruta pública para los archivos del navegador y los assets
- providers: un array de proveedores de nivel de plataforma para las requests
- inlineCriticalCss: un booleano que indica si se debe incluir CSS crítico (default: true)

Hydration

Cuando se habilita SSR, las respuestas de HttpClient se almacenan en caché mientras se ejecutan en el servidor. Esta información se serializa y se transfiere a un navegador como parte del HTML inicial enviado desde el servidor. En un navegador, HttpClient comprueba si tiene datos en la caché y, si es así, los reutiliza en lugar de hacer una nueva solicitud HTTP durante el renderizado inicial de la aplicación. HttpClient deja de usar la caché una vez que una aplicación se vuelve estable mientras se ejecuta en un navegador.

El almacenamiento en caché se realiza de forma predeterminada para todas las solicitudes HEAD y GET. Puedes configurar esta caché utilizando withHttpTransferCacheOptions al proporcionar la hidratación.

En el archivo main.server.ts, se puede configurar la caché de transferencia HTTP:

```
bootstrapApplication(AppComponent, {
  providers: [
    provideClientHydration(
      withHttpTransferCacheOptions({
        includePostRequests: true,
      }),
    ),
  ],
});
```

Compatibilidad con API del navegador

Algunas funcionalidades del navegador no están disponibles en el servidor. Las aplicaciones no pueden hacer uso de objetos globales específicos del navegador como window, document, navigator o location, así como ciertas propiedades de HTMLElement.

In general, code which relies on browser-specific symbols should only be executed in the browser, not on the server. This can be enforced through the afterRender and afterNextRender lifecycle hooks. These are only executed on the browser and skipped on the server. En general, el código que depende del navegador sólo debe ejecutarse en éste, no en el servidor. Esto se puede hacer con los nuevos kooks en el ciclo de vida afterRender y afterNextRender. Éstos sólo se ejecutan en el navegador y se omiten en el servidor.

Dónde se ejecutan los logs:

- **constructor:** se ejecuta en el servidor y en el navegador
- **oninit:** se ejecuta en el servidor y en el navegador
- **onchanges:** se ejecuta en el servidor y en el navegador
- **ondestroy:** se ejecuta en el servidor y en el navegador
- **afterrender:** se ejecuta sólo en el navegador
- **afternextrerender:** se ejecuta sólo en el navegador
- **Eventos de DOM:** se ejecutan sólo en el navegador

Utilizando PLATFORM_ID con isPlatformBrowser y isPlatformServer, podemos obtener y manejar funcionalidades específicas del navegador.

```
import { PLATFORM_ID, Inject } from '@angular/core';
import { isPlatformBrowser, isPlatformServer } from '@angular/common';

@Component({
  selector: 'my-cmp',
  template: `...`,
})
export class MyComponent {
  constructor(@Inject(PLATFORM_ID) private platformId: Object) {
    if (isPlatformBrowser(this.platformId)) {
      console.log('This will only be logged in the browser');
    }
    if (isPlatformServer(this.platformId)) {
      console.log('This will only be logged on the server');
    }
  }
}
```

Ejemplo de uso de afterNextRender:

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

@Component({
  selector: 'my-cmp',
  template: `<span #content>{{ ... }}</span>`,
})
export class MyComponent {
  @ViewChild('content') contentRef: ElementRef;

  constructor() {
    afterNextRender(() => {
      // Safe to check `scrollHeight` because this will only run in the browser, not the
server.
      console.log('content height: ' + this.contentRef.nativeElement.scrollHeight);
    });
  }
}
```

Angular Service Worker

Con angular SSR, el comportamiento de service worker cambia. La solicitud inicial del servidor se renderizará en el servidor como se espera. Sin embargo, después de esa solicitud inicial, las solicitudes posteriores son manejadas por el service worker y siempre se renderizan en el lado del cliente.

Habilitar la recopilación de datos de rendimiento

El CommonEngine ofrece una opción para iniciar la recopilación de datos de perfil de rendimiento y mostrar los resultados en la consola del servidor.

Esto se puede hacer estableciendo en el archivo server.ts:

```
content_copy
const commonEngine = new CommonEngine({
  enablePerformanceProfiler: true,
});
```

NgRx

NgRx es una biblioteca de gestión de estado para Angular. Proporciona una forma de gestionar el estado de la aplicación de forma reactiva y unidireccional.

Instalar NgRx

Para instalar NgRx, se puede utilizar el comando ng add:

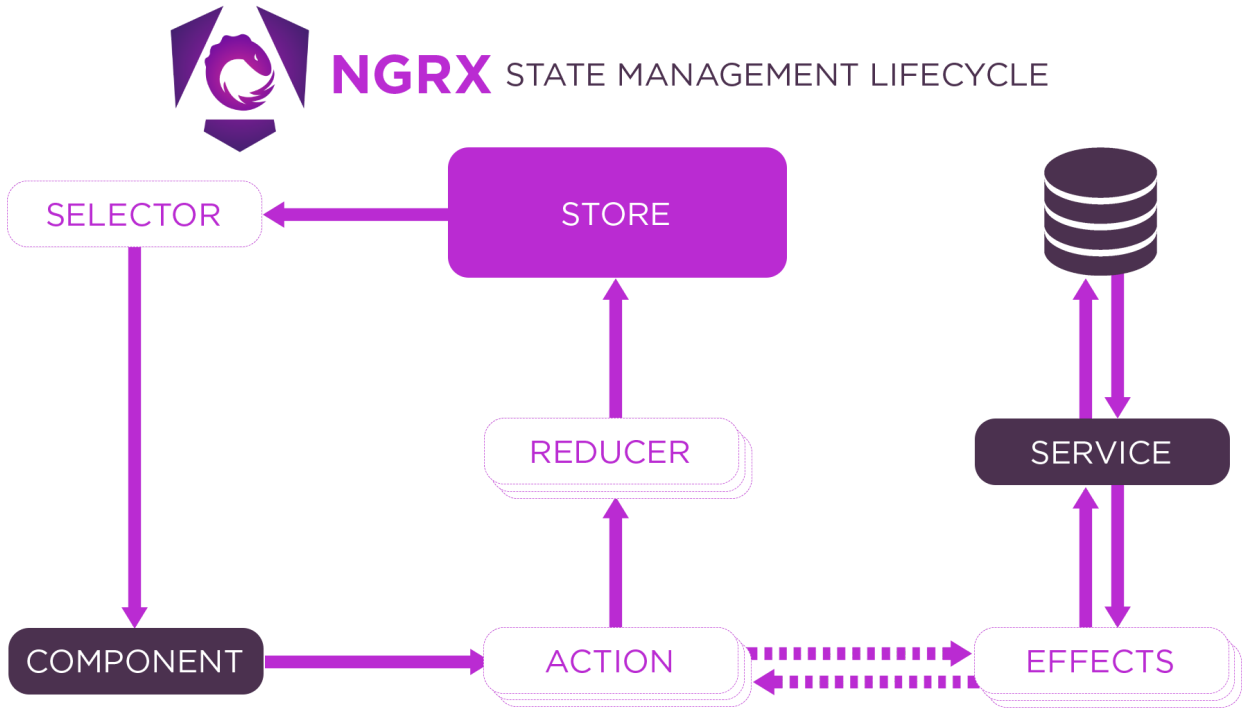
```
ng add @ngrx/store@latest
```

Este comando instala NgRx y realiza las siguientes tareas:

- Actualiza el archivo package.json con las dependencias de NgRx
- Modifica app.config.ts para añadir provideStore()

Conceptos clave de NgRx:

- **Store** es un contenedor de estado que contiene el estado de la aplicación.
- **Actions** describen eventos únicos que se envían desde componentes y servicios.
- **Cambios de estado** son manejados por funciones puras llamadas reductores que toman el estado actual y la última acción para calcular un nuevo estado.
- **Selectors** son funciones puras utilizadas para seleccionar, derivar y componer piezas de estado.
- **State** se accede con el Store, un observable de estado y un observador de acciones.



Ejemplo de uso de NgRx

Hay que definir un estado inicial y los reducers en un archivo (reducer.ts):

```
import { Demo } from "../modal/demo.modal";
import * as DemoActions from './../actions/demo.actions';

const initialState: Demo = {
    name: 'Harisudhan',
    gender: 'Male'
}

export function reducer(state: Demo[] = [initialState], action: DemoActions.Actions) {
    switch(action.type) {
        case DemoActions.ADD_DEMO:
            return [...state, action.payload];
        case DemoActions.REMOVE_DEMO:
            state.splice(action.payload, 1);
            return state;
        default:
            return state;
    }
}
```

Un ejemplo de uso de NgRx en un componente:

```
import { Component } from '@angular/core';
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs';
import { Demo } from '../modal/demo.modal';
import * as DemoActions from './../actions/demo.actions';

@Component({
  selector: 'app-demo',
  templateUrl: './demo.component.html',
  styleUrls: ['./demo.component.css']
})
export class DemoComponent {
    constructor(private store: Store<{ demo: Demo[] }>) {
        this.demo = store.select('demo');
    }

  // Dispatch actions
    addDemo(name) {
        this.store.dispatch(new DemoActions.AddDemo({name: name}));
    }

    removeDemo(index) {
        this.store.dispatch(new DemoActions.RemoveDemo(index));
    }

}
```