Nx, mono repositorios en Angular

Empiezo este **tutorial de Angular Avanzado** con la frase con la que acabé un artículo de opinión sobre arquitectura de software acerca de Angular para grandes aplicaciones.

> Angular y las decisiones de diseño que le acompañan tienen como objetivo facilitar el desarrollo y mantenimiento a medio y largo plazo de aplicaciones web no triviales.

Las empresas de desarrollo y los clientes finales que escogen **Angular**, suelen ser de tamaño medio o grande. Cuanto mayor sea el problema más destaca este *framework*. Y tarde o temprano esos grandes proyectos necesitarán compartir o reutilizar código. La herramienta Nx de Nrwl ayuda en esa tarea facilitando la creación de espacios de trabajo multi proyecto: **los mono repositorios.**

Partiendo de cero y usando las herramientas de Nrwl.io/ crearemos un *blueprint* para desarrollar grandes aplicaciones. Al finalizar tendrás, en el mismo repositorio, un par de aplicaciones y varias librerías reutilizables creadas con los *Nx power-ups*.

> Código asociado a este tutorial en *GitHub*: AcademiaBinaria/angular-boss

1. Crear el repositorio

```
Como: arquitecto de software
quiero: disponer de un espacio de trabajo único
para: crear aplicaciones y librerías.
```

Lo primero será preparar las herramientas. **Nx** es un complemento del **CLI** así que debemos tener este último disponible. Voy a emplear yarn para la instalación de paquetes y la ejecución de comandos. Pero se muestran las instrucciones alternativas con `npm`. El repositorio siempre lo creo vacío y después agrego las capacidades específicas para **Angular**.

```
# Add latest Angular CLI
yarn global add @angular/cli
# Sets yarn as default packager for cli
ng config -g cli.packageManager yarn

# Creates empty repository
yarn create nx-workspace angular-blueprint

# also with NPM...
npm i -g @angular/cli
npx create-nx-workspace@latest angular-blueprint
```

2. Generar una SPA con Angular

```
As a: customer,
  I want: to see a shop
  so that: I can buy products

 As a: seller,
  I want: to see a warehouse
  so that: I can take control
```

Los próximos comandos te sonarán a los mismo del **angular-cli**. Es normal, pues **Nx** utiliza y mejora las capacidades de la herramienta original. La diferencia está en que la recién creada aplicación, en lugar de nacer en la raíz del *workspace*, va la carpeta específica `/apps`.

```
# Add Angular capabilities
yarn add --dev @nrwl/angular

# Generate an Angular application with nx power-ups
ng g @nrwl/schematics:application shop --inlineStyle --routing --directory= -p ab-
shop
# Start default !!!
yarn start
# Generate an Angular application with nx power-ups
ng g @nrwl/schematics:application warehouse --inlineStyle --routing --directory= -
p ab-warehouse
# Start especific !!!
ng serve warehouse --port=4202 -o
```

Ambas aplicaciones comparten la configuración del `angular.json` y las demás herramientas de ayuda como **tslint** y **prettier**. Destaca mucho que también que compartan `/node_modules`, lo cual se agradece en el tiempo y en el espacio.

Con nx puedes crear también aplicaciones de Backend. Por ejemplo un API REST hecha con NestJS. Para acceder a los generadores específico tienes que instalarlos antes. por ejemplo:

```
yarn add --dev @nrwl/nest
ng g @nrwl/nest:application api
ng serve api
```

Al tener varias aplicaciones es conveniente disponer de scripts específicos para cada una. Por ejemplo yo en el `package.json` tendía algo así.

```
{
  "start:shop": "ng serve shop --port=4201 -o",
  "build:shop": "ng build shop --prod",
  "test:shop": "ng test shop",
```

```
    "start:warehouse": "ng serve warehouse --port=4202 -o",
    "build:warehouse": "ng build warehouse --prod",
    "test:warehouse": "ng test warehouse",
    "start:api": "ng serve api",
    "build:api": "ng build api --prod",
    "test:api": "ng test api",
  }
```

Por supuesto que todos estos comando se pueden lanzar visualmente mediante la extensión Angular Console para VSCode. Te recomiendo además estas otras extensiones:

```
{
  "recommendations": [
    "nrwl.angular-console",
    "angular.ng-template",
    "ms-vscode.vscode-typescript-tslint-plugin",
    "esbenp.prettier-vscode",
    "pkief.material-icon-theme",
    "christian-kohler.path-intellisense",
    "ban.spellright",
    "johnpapa.angular-essentials"
  ]
}
```

---

3. Tener una biblioteca TypeScript con lógica de dominio.

```
Como: arquitecto
quiero: tener una biblioteca en TypeScript con lógica de dominio
de modo que: pueda usarla con varios frameworks o incluso en puro JavaScript.
```

Más temprano que tarde aparecerán funcionalidades comunes a distintas aplicaciones. Validadores genéricos, utilidades o casos concretos de un cliente pero que se usan en todos sus desarrollos. En este ejemplo partimos de la necesidad común de un sistema de saludos (un mensaje, vaya). Y para ello empezamos por definir una `interface` pública reutilizable.

Un poco de arquitectura de software. Todo lo que podamos programar y que no dependa de un *framework* debemos encapsularlo en librerías independientes. De esa forma puede reutilizarse con otras tecnologías o sobrevivir dignamente a la evolución o desaparición de Angular.

Lo primero será crear la librería. Pero esta vez no usaremos los *schematics* del **cli**, si no los propios de **nrwl**. La idea es usarla como la **capa de dominio de la arquitectura**. En ella pondremos los modelos y servicios de lógica de negocio con las menores dependencias posibles. Repito lo fundamental: minimizar las

dependencias. En concreto no dependeremos de Angular, lo cual permitiría usarlo con otros *frameworks* actuales o futuros.

```
# Generate a Type Script library with nx power-ups
ng generate @nrwl/workspace:library domain --directory=shared
```

Por ahora no te preocupes de la implementación. La muestro para destacar las dos cosas que considero más importantes:

- No hay ninguna referencia explícita a *Angular*.
- Lo que quieras exportar debe indicarse en el fichero `index.ts`.

Por lo demás es puro *TypeScript*; en una carpeta con intenciones bien claras: `models/` creo de forma manual el siguiente fichero:

`libs\shared\domain\src\lib\models\greetings.interface.ts`

```
export interface Greetings {
  message: string;
}
```

al no disponer del sistema de módulos de angular tengo que exportarlo en el índice de la librería.

`libs\shared\domain\src\index.ts`

```
export * from './lib/models/greetings.interface';
```

Para localizarlo, Nx crea un alias en el `tsconfig.json`, que puedes retocar a voluntad

```
"paths": {
  "@a-boss/domain": ["libs/shared/domain/src/index.ts"]
}
```

Ya está listo para seer consumido desde distintos proyectos (Lo haré después en el API y ya mismo en la próxima librería de componentes).

---

4. Tener una biblioteca Angular con componentes de interfaz

```
As a: customer,
  I want: to be greeted
  so that: I feel at home

As a: seller,
  I want: to be greeted
  so that: I feel at home
```

Si eres una empresa consultora es posible que te encuentres repitiendo funciones o pantallas una y otra vez para distintos clientes. Por supuesto que una gran empresa seguro que se hacen muchas aplicaciones similares, a las que les vendría de maravilla **compartir una biblioteca de componentes**.

Pues ahora crear librerías es igual de sencillo que crear aplicaciones. **Nx** las depositará en la carpeta `/libs` y se ocupará de apuntarlas en el `tsconfig.json` para que la importación desde el resto del proyecto use alias cortos y evidentes.

Crear componentes en un entorno multi proyecto requiere especificar a qué proyecto se asociarán. Para empezar vamos a crear los componentes básicos para esta funcionalidad en una librería compartida de para interfaz de usuario.

```
# Generate an Angular library with nx power-ups
ng g @nrwl/angular:library ui --directory=shared --prefix=ab-ui --simpleModuleName
# Generate Greetings Component
ng g component greetings --project=shared-ui --module=ui.module.ts --export --
inlineStyle --inlineTemplate
```

Y le damos contenido al componente. Fíjate en la importación de la interfaz `Greetings`.

`libs\shared\ui\src\lib\greetings\greetings.component.ts`

```
import { Greetings } from '@a-boss/domain';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'ab-ui-greetings',
  template: `
    <p>
      {{ theGreeting.message }}
    </p>
  `,
  styles: []
})
export class GreetingsComponent implements OnInit {
  public theGreeting: Greetings = { message: 'Hello world' };
  constructor() {}

  ngOnInit() {}
}
```

Puedes usarlos como cualquier otro componente y en cualquier aplicación del repositorio. Simplemente importando el módulo en el que se declaran: el `UiModule`. NX se encarga de referenciar cada proyecto en el fichero `tsconfig.json`. De esa forma se facilita su importación en cualquier otra aplicación del repositorio.

Primero importamos el módulo.

`apps\shop\src\app\app.module.ts`

```
import { UiModule } from '@a-boss/ui';
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule } from '@angular/router';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    RouterModule.forRoot([], { initialNavigation: 'enabled' }),
    UiModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Y ya podemos incrustar sus componentes públicos.

`apps\shop\src\app\app.component.html`

```
<ab-ui-greetings></ab-ui-greetings>
<router-outlet></router-outlet>
```

`apps\shop\src\app\app.component.ts`

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

@Component({
  selector: 'ab-shop-root',
  templateUrl: './app.component.html',
  styles: []
})
export class AppComponent {
  title = 'shop';
}
```

5. Tener una biblioteca Angular con servicios de datos

```
As a: customer,
  I want: to be greeted by the API :-)
  so that: I know I am not alone.

As a: seller,
  I want: to be greeted by the API :-)
  so that: I know I am not alone.
```

Además de componentes visuales, podemos tener librerías con servicios de lógica o de acceso a datos. Por ejemplo un servicio para comunicarnos con el API podría ser utilizado en diversos proyectos (aplicaciones o librerías).

Con lo ya sabido vamos a crear una librería compartida para acceso a datos.

```
ng g @nrwl/angular:library data --directory=shared --prefix=ab-data --
simpleModuleName
ng g service greetings --project=shared-data --no-flat
```

El servicio realiza la llamada http y devuelve un observable.

libs\shared\data\src\lib\greetings\greetings.service.ts

```
import { Greetings } from '@a-boss/domain';
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
@Injectable({
  providedIn: 'root'
})
export class GreetingsService {
  private apiUrl = 'http://localhost:3333/api';
  constructor(private httpClient: HttpClient) {}
  public getGrettings$(): Observable<Greetings> {
    return this.httpClient.get<Greetings>(this.apiUrl);
  }
}
```

Para consumir el servicio no hay que hacer nada más. Pero, para importarlo en TypeScript, necesitmaos que nos lo exporten adecuadamente.

libs\shared\data\src\index.ts

```
export * from './lib/data.module';
export * from './lib/greetings/greetings.service';
```

tsconfig.json

```
"paths": {
  "@a-boss/domain": ["libs/shared/domain/src/index.ts"]
}
```

Y ahora consumirlo ya no es un problema. Por ejemplo directamente en el componente.

libs\shared\ui\src\lib\greetings\greetings.component.ts

```
import { GreetingsService } from '@a-boss/data';
import { Greetings } from '@a-boss/domain';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'ab-ui-greetings',
  template: `
    <p>
      {{ theGreeting.message }}
    </p>
  `,
  styles: []
})
export class GreetingsComponent implements OnInit {
  public theGreeting: Greetings = { message: 'Hello world' };
  constructor(private greetingsService: GreetingsService) {}
  public ngOnInit() {
    this.greetingsService.getGrettings$().subscribe(this.appendApiMessage);
  }
  private appendApiMessage = (apiGreetings: Greetings) =>
    (this.theGreeting.message += ' and ' + apiGreetings.message);
}
```

Ya que estamos accediendo al API, podemos aprovechar para adecuar sus tipos a la interfaz declarada en el dominio. Fíjate lo familiar que resulta este código **NodeJS** gracias al framework nest.

apps\api\src\app\app.controller.ts

```
import { Greetings } from '@a-boss/domain';
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
```

```
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getData(): Greetings {
    return this.appService.getData();
  }
}
```

Resumen

En definitiva, los grandes desarrollos demandados por bancos, multinacionales o administración pública requieren soluciones avanzadas. **Angular** es una plataforma ideal para esos grandes proyectos, pero requiere conocimiento y bases sólidas para sacarle partido.

Con este tutorial empiezas tu formación avanzada en Angular para poder afrontar retos de tamaño industrial. Continúa aprendiendo a crear y ejecutar pruebas automatizadas creando tests unitarios con Jest y e2e con Cypress en Angular.

> Aprender, programar, disfrutar, repetir. -- *Saludos, Alberto Basalo*