

Angular: Guía práctica

Desde 0 hasta producción

Rubén Aguilera

Angular: Guía práctica

Desde 0 hasta producción

Rubén Aguilera

Este libro está a la venta en <http://leanpub.com/angular-guia-practica>

Esta versión se publicó en 2018-05-03



Este es un libro de [Leanpub](#). Leanpub anima a los autores y publicadoras con el proceso de publicación. [Lean Publishing](#) es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener feedback del lector hasta conseguir tener el libro adecuado.

© 2016 - 2018 Rubén Aguilera

A Lucía y Mateo por todas las horas que les robo y a todos mis compañeros de Autentia, en especial a los tres socios. Gracias a todos por apoyar mis pasiones

Índice general

¿Por qué Angular?	1
JavaScript	1
Web Components	3
Programación reactiva (RxJS)	4
Angular	4
Simplicidad	5
Rendimiento óptimo	6
Ahead of Time Compilation	6
Lazy Loading	7
Inyección de dependencias y testing	7
Mejoras en el SEO	7
Integración con otras tecnologías	8
Multi soporte a navegadores	9
Internet de las cosas y la web física	9
Conclusión	10
Entorno de desarrollo	11
Instalación de las herramientas necesarias	11
Git	11
NodeJS y npm	12
Angular-cli	12
Editor de texto: Visual Studio Code	13
Chrome	14
Gestión del proyecto con Angular-Cli (v1.7.4)	14
Creación del proyecto	14
Ejecución de la aplicación en desarrollo	15
Ejecución de los tests unitarios	18
Ejecución de los tests de aceptación o e2e	19
Creación de elementos	19
Descubrimiento de errores de estilo y codificación	25
Construcción del proyecto	25
Acceder a la documentación	26
Consejo para la ejecución de los comandos	26

ÍNDICE GENERAL

Módulos	28
@NgModule	28
Creación de un módulo secundario	29
Arranque de la aplicación automático	30
Arranque manual de la aplicación	32
Directivas	34
Componentes	34
Directivas de atributo	35
ngClass y class binding	35
ngStyle y style binding	35
Directivas estructurales	36
ngIf	36
ngIf - then - else	37
ngSwitch	37
ngFor	37
Creación de una directiva de atributo propia	38
Pipes	41
Pipes de serie	41
Parametrización y encadenado	44
Creación de pipes propios	44
Data Binding	46
Interpolation	46
Property binding	46
Element Property	46
Component Property	47
Directive property	47
Event binding	48
NgModel	49
Servicios e inyección de dependencias	50
Elemento Injector	50
Injector principal	51
Injectores secundarios	51
Localización de dependencias	52
Declaración de providers	53
useClass	53
useValue	54
useFactory	55
Simplificación en la inyección	55

ÍNDICE GENERAL

Routing	57
Configuración del router en el módulo principal	57
Configuración del router en módulo secundario o de feature	58
Configuración con Lazy Loading	59
Para el módulo secundario o de feature	59
Para el módulo principal	60
Router outlet y router links	60
Navegación desde código	61
Recuperación de la información	62
Query params	63
Rutas de guarda	64
CanActivate	64
CanDeactivate	66
Resolución de información en el router	68
Acceso a la información del router del padre	69
Los 7 pasos del proceso de routing	70
Formularios	72
Configuración inicial	72
Creación del formulario	72
Configuración de los elementos del formulario	73
Validaciones síncronas	74
Validaciones asíncronas	76
Configuración de cuando disparar las validaciones	77
Estados de un formulario	77
Http: comunicación con servidor remoto	82
Historia de la asincronía	82
Callbacks	82
Promesas	83
Observables	84
Servicio HttpClient	85
Recuperación de datos	86
Interceptores	90
Modificar cabeceras	91
Logging	91
Manejar errores	92
Controlar peticiones a través de timeout	93
Escucha de eventos de progreso	94
Uso de async pipe con objetos	94
Testing	96
Introducción	96

ÍNDICE GENERAL

Tests unitarios y de integración con Jasmine	96
Test Suite	97
Test Case	97
Matchers	98
beforeEach() y afterEach()	98
Karma	99
Arquitectura de una aplicación testeable	101
TestBed	103
Situaciones de testing	104
Test de un pipe / servicio sin dependencias	104
Test de un pipe/servicio con dependencias	105
Tests asíncronos	106
Test de componente sin dependencias	107
Test de componente con dependencias	108
Tests de aceptación con Protractor	109
Protractor	109
Caso práctico	112
Conclusión	125
Integraciones	126
Integración con PrimeNG	126
Instalación de PrimeNG en angular-cli	126
Utilización de la librería	127
Integración con Polymer 2.0	128
Nos ponemos la gorra de arquitectos y diseñadores	129
Nos ponemos la gorra de desarrolladores	132
Integración con StencilJS	132
Empezando con StencilJS	133
Creación del primer componente	134
Puesta en producción	142
Análisis del tamaño de la aplicación	142
Verifica que solo se incluye lo obligatorio	142
Verifica el tamaño del bundle principal	143
No subas los .map	143
Genera la documentación	144
Aplicar técnicas de optimización	145
Entendiendo el “Browser rendering”	145
Cómo mejorar el “Browser rendering”	146
Técnicas de optimización en Angular	147
Eliminación de espacios en blanco	147
Uso de web workers	147
Extracción de la configuración de la aplicación	150

ÍNDICE GENERAL

Despliegue de la solución en nginx	156
Empaquetado con Docker	157
Programación reactiva (RxJS)	160
Conceptos básicos	161
DataSource	161
Observable	161
Formas de crear un Observable	161
Subscriber	162
Observer	162
Subscription	162
Operadores	162
.map(dato ⇒ function(dato))	163
.filter(dato ⇒ condicion booleana)	163
.do(x ⇒ function(x))	163
.reduce()	164
.scan()	164
.concat(otherObservable)	165
.debounceTime(x ms)	165
.distinct()	166
.distinctUntilChanged()	166
.switchMap()	166
.catch()	167
.buffer(obs)	167
.bufferTime(x)	167
.bufferCount(n)	167
Subject	167
BehaviorSubject	167
ReplaySubject	167
AsyncSubject	168
Casos de uso	168
Evitar problema al llamar dos veces a un Observable	168
Encadenar dos peticiones donde la segunda depende de los datos de la primera	170
Sistema de notificaciones a través de bus	171
Repetir una llamada Http cada cierto tiempo con actualización de datos sin parpadeo de pantalla	173
Combinar datos de dos llamadas distintas	174
Gestión del estado	176
Model Pattern	177
Creación de una librería para Angular	182
Usando @angular/cli	182

ÍNDICE GENERAL

Creación de la librería desde cero	182
Uso de packagr para la distribución	185
Server rendering	188
Introducción	188
Pasos para hacer server rendering	188
Añadimos pre-rendering	193
Conclusión	195
Internacionalización	196
Haciendo uso del core de Angular	196
Indicar los textos que se quieren internacionalizar	196
Extracción a fichero de mensajes	197
Creación de los ficheros de mensajes por idioma	200
Puesta en producción	200
Uso de la librería de ngx-translate	201
Instalación de las dependencias	201
Configuración en el módulo principal	201
Estableciendo el idioma por defecto	202
Creación de los ficheros de idiomas	203
Uso de las traducciones en los componentes	204
Uso de las traducciones en los servicios/pipes	204
Integración y despliegue continuo	205
Integración continua con TravisCI y despliegue en Firebase	205
Ionic	215
Como empezar a crear una aplicación (Ubuntu 16.04)	215
Conectar con los servicios gratuitos de Ionic PRO	217
Establecer monitoring	218
Captura automática de los errores	218
Captura manual	219
Añadir los sourcemaps	220
Crear nueva página en el sidemenu	220
Eventos de navegación	221
Gestión del almacenamiento	222
Declaración de múltiples almacenamientos	224
Gestión de las variables de entorno	226
Creación y carga dinámica de temas	228
Implementación de hot deploy	230
¿Qué es el hot deploy?	230
Como aplicarlo a los proyectos	231
Configuración para distintos entornos de ejecución	233

ÍNDICE GENERAL

Usando Firebase como servidor de producción	234
Probando el despliegue en caliente	236
NX: cómo abordar proyectos complejos	237
NX al rescate	237
Empezando con NX	238
Creación de una nueva aplicación	239
Creación de una librería como módulo secundario	239
Creación de una librería	240
Arranque/distribución de la aplicación	240
Publicación de una librería	240
Cosas a tener en cuenta	241
Actualizar el workspace a la última versión de Angular	242
Visualizar en un gráfico las dependencias entre aplicaciones y librerías	242
Integración de Ionic en un workspace de NX	245
Principios de diseño y buenas prácticas	248
Componentes pequeños y tontos	248
No subestimes el poder de los pipes	248
Uso de la herencia en los componentes	248
No uses funciones para calcular datos dentro de los templates	250
Los datos siempre deben fluir en una única dirección	250
No dejes a tus componentes jugar con todos los juguetes ni saber nada del estado de la aplicación	251
Utiliza servicios proxy para acceder a los datos de un API sin nada más de lógica	251
Pasos para abordar una aplicación del mundo real	252

¿Por qué Angular?

Vamos a tratar de explicar por qué tienes que usar Angular en tus próximos desarrollos de frontend y ponerte un poco en contexto.

JavaScript

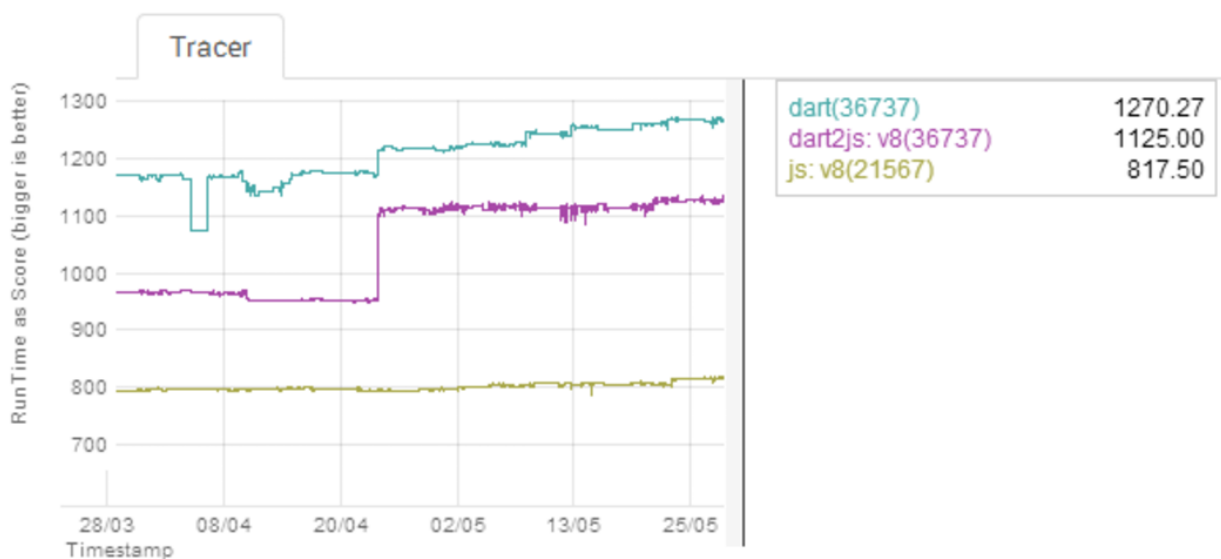
El desarrollo frontend tiene un único denominador común: JavaScript pero siempre se le ha puesto la etiqueta de ser un lenguaje pobre, no apto para abordar grandes proyectos debido a que no es tipado y no presenta una arquitectura de desarrollo bien definida de serie.

Al principio de los tiempos los desarrolladores trabajábamos con JavaScript “a pelo”, seguramente con la versión ES3. A medida que se fue estandarizando el desarrollo, cada uno se creaba sus propias librerías JS con recursos que intentaba reutilizar de la mejor manera posible en sus proyectos.

La siguiente mejora fue la aparición de la librería JQuery, que estandarizaba un poco más el desarrollo ofreciéndonos librerías de componentes que directamente podíamos utilizar en nuestros desarrollos y simplificando en gran medida la sintaxis de JavaScript, esto hizo que prácticamente se nos olvidará trabajar con la sintaxis nativa de JavaScript y abrazáramos el \$.

Pero, aun así, todavía se le achacaba la falta de estructuración en los desarrollos que impedían que se pudiera utilizar en proyectos de gran envergadura. He aquí que surgieron con fuerza frameworks como: Backbone.js, Marionette.js, Ember.js, ... que de algún modo imponían tener una estructura en los desarrollos pero que aparecían como setas, cada uno con sus ventajas e inconvenientes, obligando a los desarrolladores a tener que aprender cada uno de estos nuevos frameworks en función del desarrollo.

El verdadero cambio se empezó a fraguar en 2014 con la llegada de Angular Dart que usaba una pequeña librería llamada Polymer para la implementación de la parte visual. Maravillados quedamos cuando Google presentó un nuevo lenguaje de programación con tipado y que se podía entender a la primera, gracias a las clases, la herencia simple, las interfaces, además incluía un IDE que permitía “compilar” el lenguaje con su propia máquina virtual llamada “DartVM” con lo que se minimizaban los errores de sintaxis en tiempo de ejecución y tenía un rendimiento muy superior a JavaScript, incluso en su versión compilada.



Fuente: <https://www.adictosaltrabajo.com/tutoriales/dart> (Junio, 2014) (Moisés Belchín)

Entonces, ¿cuál fue su problema? Su problema es que Google pretendió competir con JavaScript para hacerlo desaparecer y el resto de navegadores no estuvieron por la labor de implementar la máquina virtual de Dart, con lo que su uso se restringía al V8 de Google Chrome.

Os imagináis si solo hubiera un único navegador para el que desarrollar, todo sería super rápido y la cantidad de pastillas que nos ahorraríamos los desarrolladores, pero pura utopía :-)

Después de este primer paso fallido, Google aprendió la lección de que no se puede ir contra JavaScript, y se unió a él. Pero para adaptar JavaScript al mundo empresarial creo AngularJS, que es un framework que proporciona una forma estructurada y totalmente testeable de abordar aplicaciones de mucha envergadura con JavaScript y que aprendió de la experiencia de otros frameworks como Backbone.js, Ember.js o Marionette.js. El problema es que para hacer su “magia” necesita realizar una serie de procesos que empeoran el rendimiento a medida que la aplicación se hace más compleja y va cargando el DOM con nuevos elementos.

Con ES6 se ha dado un paso muy importante en la legibilidad del lenguaje añadiendo conceptos de la programación orientada a objetos como las clases, la herencia, los métodos estáticos, etc... pero los navegadores actuales no soportan nativamente todas las características de ES6 por lo que se introduce el concepto de “transpilador” que permite convertir código con sintaxis ES6 a código ES5 entendible por el navegador “al vuelo”. Los más conocidos son Traceur y Babel.

Pero aún con este gran avance el lenguaje sigue con el problema fundamental de no ser un lenguaje compilado ni tipado, por lo que muchos de los errores de sintaxis o las

refactorizaciones grandes de código se tienen que seguir haciendo “a ciegas”, es decir probando una y otra vez en tiempo de ejecución.

En vista de este problema la gente de Microsoft inventa TypeScript como un complemento a ES6 (ya sabemos que contra JavaScript no se puede ir) que fundamentalmente proporciona tipado e incluye el concepto de interfaces (vaya esto me suena un poco a lo de antes con Dart), lo que da pie a que herramientas como Atom, WebStorm, Visual Studio puedan ofrecer distintas utilidades al programador como el marcado de errores, facilidad de refactoring, compilación al vuelo, etc... que hace que las aplicaciones grandes con JavaScript sean mucho más mantenibles y los equipos que las desarrollan mucho más productivos.

Web Components

Por otro lado, gracias a la fuerza de Google, se iba modificando la especificación de HTML para dar cabida a la tecnología de web components cuya principal ventaja es que nos permite extender el HTML a nuestro antojo, pudiendo crear etiquetas personalizadas que encapsulan el contenido HTML, la funcionalidad en JavaScript y el estilo en CSS, para conseguir esto se apoyan en estas cuatro APIs JavaScript que están alrededor de la especificación de HTML:



Custom Elements: es la tecnología que nos permite crear las etiquetas semánticamente correctas para la funcionalidad que le queramos dar. Es una recomendación que la etiqueta incluya un ‘-‘ para reducir el riesgo de colisiones con otras etiquetas del estándar HTML, ya que el W3C se compromete a que nunca va a crear etiquetas que incluyan este carácter.



HTML Imports: es la tecnología que nos permite importar un código HTML en otro HTML, se utiliza para la distribución de las librerías de web components. Esta tecnología se encuentra en desuso dado que su funcionalidad de importación de código se prefiere abordar con los módulos de JavaScript.



Templates: es la tecnología que permite crear la estructura visual del web component. No tiene ningún efecto sobre la página, es decir, no se incluye en el DOM hasta que algún proceso de JavaScript lo procesa, hace una copia y la incluye.



Shadow DOM: es la tecnología que permite encapsular el web component para que no se vea afectado por el DOM general de la página a no ser que lo permitamos explícitamente.

Programación reactiva (RxJS)

Otro de los avances más importantes ocurridos en los últimos tiempos en el mundo de la programación y que ha afectado especialmente a la programación web es la llegada de la programación reactiva.

La librería RxJS representa en la actualidad un estándar de facto para la implementación de la programación reactiva en la web que nos permite a los desarrolladores solucionar problemas comunes con el mínimo código posible, lo que aumenta la mantenibilidad de los proyectos.

Hablaremos de ella más en profundidad en su capítulo dedicado.

Angular

Mucha gente todavía se pregunta por qué, con el éxito que tenía AngularJS, Google implementó una solución tan radicalmente diferente con Angular 2 y posteriores. La razón es que AngularJS intenta hacer uso de los web components de forma artificial lo que le obliga a realizar procesos en el DOM que requieren de mucho procesamiento, es por eso que cuando la aplicación se va complicando y se van introduciendo más elementos en el DOM, AngularJS adolece de serios problemas de rendimiento.

Por eso en el momento en el que los Web Components se introdujeron de forma nativa en los navegadores (algunos todavía con polyfills, a pesar de ser el estándar), era el momento

idóneo de reescribir la solución para aumentar el rendimiento, gracias a la programación reactiva, y hacerla más mantenible, gracias a TypeScript.

Muchos de los elementos principales de Angular hacen uso intensivo de la programación reactiva; con lo que se convierte en un framework que permite aprovechar todos estos nuevos estándares para la creación de aplicaciones SPA, además de ser el único que de serie incluye la inyección de dependencias para favorecer el testing y se integra a la perfección con tecnologías de todo tipo.

Se puede decir que es el Spring de la Web, y que como en el caso de Java con Spring, ya está marcando un antes y un después en el desarrollo web.

A continuación vamos a describir sus principales características:

Simplicidad

El equipo de Angular en la nueva versión (Angular) ha perseguido simplificar el anterior framework (AngularJS), quedándose solo con la bueno y mejorando o eliminando lo que se ha visto que funciona peor.

Entre lo malo de AngularJS cabe destacar la utilización directa del \$scope para sincronizar los elementos de la vista y el modelo, el sobre diseño de los servicios (cuenta con 5 tipos que se pueden reducir a uno solo), la eliminación de directivas propias (ng-click, ng-bind, ...) y la simplificación en la construcción de directivas remplazando el DDO, que es el objeto que define la directiva, por el concepto de componente que extiende el HTML haciendo uso de los Web Components que nativamente logran un rendimiento muy superior.

Además TypeScript le proporciona una sintaxis mucho más entendible para desarrolladores cercanos a lenguajes de programación orientados a objetos como Java o C#, y los templates definen claramente las características de la vista de un componente, sus relaciones con otros componentes y las interacciones que manejan.

Todo esto redundará en una curva de aprendizaje mucho más suave que hace que la productividad de los equipos aumente, permitiendo a los desarrolladores centrarse en la lógica de negocio de la aplicación y la presentación; del resto ya se encarga el framework, favoreciendo tareas comunes como el routing de la aplicación, las llamadas a servidores externos, la validación de formularios, etc...

Otro punto que simplifica el proceso de desarrollo es la creación de angular-cli, que es una herramienta por línea de comandos que nos permite comenzar con un proyecto de cero perfectamente configurado para que solo nos tengamos que preocupar de la programación y no de la configuración básica del proyecto.

Rendimiento óptimo

Esta es otra de las máximas del equipo de Angular donde han llegado a mejorar el rendimiento de AngularJS en un tiempo de respuesta cinco veces más rápido.

Esto lo han conseguido gracias al aprovechamiento nativo de los Web Components y la mejora del sistema de detección de cambios en la vista, donde en AngularJS se hacía con un ciclo de digest que consumía muchos ciclos de CPU y en esta versión se implementa con un sistema reactivo mucho más eficiente.

Relacionado con esta mejora, ahora el data binding es unidireccional, para hacerlo bidireccional como en AngularJS, tenemos que hacerlo de forma explícita.

Todo el framework está optimizado para móviles, que es el escenario más limitado en hardware, donde los ciclos de CPU y memoria son parámetros críticos para su correcto funcionamiento.

Además, no hay que olvidar que es un framework que se ejecuta en un navegador, el cual podría presentar problemas de “congelación” al ejecutar procesos pesados que bloqueen el hilo principal de ejecución. Para evitar esto, de forma transparente al desarrollador, todo el framework se ejecuta utilizando web workers, que abren nuevos hilos de ejecución, evitando el efecto de congelación de la pantalla.

De cara a reducir el tamaño de la aplicación en producción que hace que la primera carga sea mucho más rápida, el equipo de Angular ha introducido dos nuevos conceptos, que son:

Ahead of Time Compilation

Consiste en hacer la compilación del código en el servidor en tiempo de “build”, con lo que conseguimos un feedback rápido de los posibles errores y un fichero “bundle” optimizado para su ejecución.

En este proceso de “build” con AOT durante el proceso de minificado de los ficheros se hace un procesamiento llamado “tree-shaking” que consiste en “agitar” el árbol DOM para que las referencias que no están ligadas, las que no se usan, se eliminen reduciendo el tamaño del bundle al mínimo posible.

Es importante destacar la necesidad de eliminar los imports no utilizados en nuestro proyecto, ya que los elementos que importen serán incluidos en el bundle a pesar de no utilizarse en el código.

Otra reducción del tamaño se produce porque como la compilación se hace en servidor, el navegador no necesita el compilador para interpretar el código, cosa que sí ocurre en JIT (Just In Time), con lo que reducimos varios megas el tamaño del bundle resultante.

Esta reducción del tamaño y la optimización del JavaScript hace que, utilizando AOT, la primera carga de la aplicación sea mucho más rápida; en contrapartida, el proceso de “build” es mucho más lento; por tanto JIT se deja para el entorno de desarrollo y AOT para producción. Aunque, actualmente, el equipo de ingenieros de Google están trabajando en que las siguientes versiones del framework tengan como objetivo acelerar el proceso de AOT tanto que haga desaparecer la necesidad de utilizar JIT incluso en desarrollo.

Lazy Loading

Otra de las mejoras en rendimiento es la carga perezosa, es decir, podemos configurar el router de tal forma que el bundle inicial solo contenga lo mínimo e imprescindible para que la aplicación se pueda ejecutar en el navegador, haciendo que esta carga sea super rápida, y a medida que el usuario va interactuando con otros módulos de la aplicación, se van descargando los bundles asociados de los que incluso se puede configurar que ya estén precargados.

Esta mejora viene favorecida por la inclusión del concepto de módulo y el cambio radical que sufrió la librería de router antes de sacar la versión final del framework.

Inyección de dependencias y testing

A estas alturas todos debemos saber las implicaciones que supone poder trabajar con un framework que te ofrece de serie inyección de dependencias e inversión de control.

Esta característica es heredada de AngularJS, nos permite reducir el acoplamiento entre clases favoreciendo el uso de dobles en el testing de la aplicación y el diseño con TDD.

Además cuenta con módulos específicos de testing que gracias a la integración con frameworks como Jasmine y Karma, nos permiten implementar y ejecutar una red de seguridad de tests que favorecen que las modificaciones futuras no introduzcan daños colaterales en nuestra aplicación.

También es posible su integración con Protractor para la implementación y ejecución de pruebas de aceptación gracias al web driver de Selenium.

Mejoras en el SEO

Las aplicaciones SPA no son SEO-friendly por defecto, ya que necesitan de una carga previa, que por muy rápida que sea, hace que las arañas de los buscadores no puedan leer el contenido de nuestra aplicación por lo que nuestras páginas nunca estarán en los primeros puestos de los rankings.

Para los casos en los que el SEO sea un elemento determinante del negocio de la aplicación, el framework te proporciona un módulo llamado “Angular Universal”, que

permite el renderizado de la aplicación SPA en servidor de forma que cuando el usuario solicita una página ésta ya está renderizada en HTML, haciendo que la carga de la aplicación sea instantánea y favoreciendo el trabajo de las arañas.

Actualmente Angular Universal se encuentra en fase de maduración y solo está comprobada su funcionalidad con backend escrito en NodeJS, se espera que pronto se tenga una versión con backend en .NET y Java.

Integración con otras tecnologías

Lo que hace especial a este framework es la integración con otras tecnologías que siguen los estándares en distintos ámbitos del desarrollo, y que en mi opinión, hará que el framework se consolide como en su día se consolidó Spring en el mundo de Java.

Angular se integra perfectamente con librerías webs que favorecen la implementación de web components como: [Polymer](https://www.polymer-project.org)¹ (Google) y [StencilJS](https://stenciljs.com/)² (Ionic) pudiendo utilizar cualquiera de estos web components directamente en los templates de los componentes creados con Angular.

Además, ya muchas empresas exportan sus web components como directivas que se integran en el framework de Angular. El caso más representativo es el de la empresa Prime, que se dedica a liberar componentes empresariales (Grid, Tablas, Botones, Gráficos, ..) con su librería [PrimeNG](http://www.primefaces.org/primeng/#/)³. El propio equipo de Angular también mantiene sus propias directivas en la librería [Angular Material](https://material.angular.io/)⁴ que, obviamente mantiene una integración perfecta con Angular, pero actualmente, tiene menos componentes que PrimeNG.

También podemos utilizar Angular con [Ionic](https://ionicframework.com/)⁵ para implementar una aplicación móvil híbrida que de serie se ajuste a las normas de estilo de la plataforma donde se esté ejecutando: Android o IOS.

En el caso de necesitar la ejecución nativa en los dispositivos móviles se puede integrar con [NativeScript](https://www.nativescript.org/)⁶, de forma que no necesitamos conocer el lenguaje específico de la plataforma (Java, Swift, ObjectiveC, C#,...) para implementar aplicaciones que se ejecuten de forma nativa en cada una de ellas.

En el caso de necesitar implementar aplicaciones que se vayan actualizando en tiempo real también contamos con la integración con [Meteor](https://www.meteor.com)⁷, que es una plataforma que integra de serie la actualización automática de la aplicación cuando en el servidor se modifican los datos.

¹<https://www.polymer-project.org>

²<https://stenciljs.com/>

³<http://www.primefaces.org/primeng/#/>

⁴<https://material.angular.io/>

⁵<https://ionicframework.com/>

⁶<https://www.nativescript.org/>

⁷<https://www.meteor.com>

Además, podemos encapsular nuestra aplicación SPA implementada con Angular en [Electrón](http://electron.atom.io/)⁸, lo que nos permite ejecutar una aplicación de escritorio con la apariencia y los accesos nativos del sistema operativo en el que se esté ejecutando.

Multi soporte a navegadores

Un punto crítico a la hora de adoptar este tipo de tecnologías es la compatibilidad con los distintos dispositivos en sus distintas versiones y plataformas; sobre todo si tiene o no compatibilidad con dispositivos antiguos, por aquello de no dejar “colgados” a un número significativo de usuarios.

En este punto os puedo decir, después de muchas pruebas empíricas, que este tipo de tecnologías funcionan perfectamente (sin hacer ninguna configuración adicional) en versiones iguales o superiores a la 4.4 de Android, a la 8.0 de IOS, en Windows 10 y en todos los navegadores Evergreen, es decir, los que son auto-actualizables (Chrome, Firefox, Edge y Safari). En la versión IE11 hay que habilitar polyfills y modificar algunas implementaciones al no soportar todo el estándar, y por debajo de esta versión en IE, mejor no lo intentes.

En caso de necesitar compatibilidad con versiones 4.2 y 4.3 de Android, se puede añadir el plugin de [Crosswalk](https://crosswalk-project.org/)⁹ a través de [Cordova](https://cordova.apache.org/)¹⁰, que añade un navegador “moderno” a nuestra aplicación para que pueda ejecutarse en estas versiones antiguas. La única pega es que el tamaño de la aplicación aumenta considerablemente, he probado con una aplicación de 4 Mb, y el APK ha pasado a 25 Mb y una vez instalada ocupa 35 Mb; así que puede ser una buena solución si el espacio no es un problema y avisamos fehacientemente al usuario que descargue la aplicación a través de una red Wifi. Pero ya os digo que esto solo afectaría a dispositivos 4.2 y 4.3 de Android, que a día de hoy tienen ya muy poca cuota de mercado.

Para Windows Phone se considera que no merece la pena mantener compatibilidad con versiones anteriores a Windows 10, normal ya no las mantienen ni ellos. Esto viene motivado por la poca cuota de mercado y por la mejora drástica, en rendimiento y compatibilidad, del navegador en su versión 10.

En conclusión esta tecnología es muy válida siempre que se impongan al cliente ciertas restricciones en las versiones de los dispositivos soportados. Android 4.4+, IOS 8+ y Windows 10+ (Phone y Desktop) y navegadores Evergreen.

Internet de las cosas y la web física

Otra de las grandezas de Angular es que su diseño está pensado para no limitarlo a dispositivos que tengan un navegador si no que puede ser utilizado para programar

⁸<http://electron.atom.io/>

⁹<https://crosswalk-project.org/>

¹⁰<https://cordova.apache.org/>

cualquier tipo de dispositivo en lo que ahora se conoce como el Internet de las Cosas y la web física.

Existen numerosos casos de éxito de desarrolladores que han utilizado Angular para programar robots, juegos infantiles y drones, así que con este framework los límites los pone la imaginación.

Conclusión

En conclusión, Angular es una buena opción para la implementación de aplicaciones, porque:

- Sigue y aprovecha todos los estándares web de la W3C con lo que no nos estamos casando con la tecnología.
- Permite tener un código único y ejecutar en multidispositivos.
- Tiene un rendimiento óptimo y es más simple y comprensible.
- Permite tener todo nuestro código bajo una red de seguridad de tests y un marco de integración y despliegue continuo.
- Su flexibilidad nos permite interactuar con cualquier tipo de dispositivo físico.

Entorno de desarrollo

En este apartado vamos a hablar de las herramientas necesarias para la creación de aplicaciones con Angular y su correcta gestión de la configuración.

Instalación de las herramientas necesarias

Para la creación de aplicaciones con Angular vamos a necesitar:

- Git
- NodeJS y npm
- Angular-cli
- Editor de texto: Visual Studio Code
- Chrome

En función del sistema operativo que tengas estas herramientas se instalarán de una forma o de otra, lo que es seguro es que todas ellas son multiplataforma, probado empíricamente.

A continuación se describe la forma de instalarlas en Ubuntu, que seguramente sea el sistema operativo libre más usado en el mundo y al que todos tenemos acceso de forma legal y gratuita. A mi personalmente me encanta su versión de escritorio LXDE, conocida como LUbuntu, que consume menos memoria RAM y es especialmente apta para máquinas virtuales.

Git

Para instalar Git solo tenemos que ejecutar en un terminal:

```
1 $> sudo apt-get install git -y
```

NodeJS y npm

La forma más flexible de instalar NodeJS y npm en cualquier sistema operativo es a través de la instalación de [NVM](https://github.com/creationix/nvm)¹¹ que es tan sencilla como ejecutar el siguiente comando con la versión que queramos: `bash $> wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.33.2/install.sh | bash`

Para Windows descargar el ejecutable desde [aquí](https://github.com/coreybutler/nvm-windows/releases)¹²

Una vez finalizado el proceso, cerramos el terminal y abrimos uno nuevo, y ya estamos listos para instalar la versión de NodeJS que queramos, podemos ver el listado completo de las distintas versiones que poder instalar ejecutando:

```
1 $> nvm ls-remote
```

Lo normal es que estemos en la última versión estable:

```
1 $> nvm install stable
```

O en el caso de Windows `bash $> nvm install latest`

Para comprobar la versión actual en uso, tanto de NodeJS como de NPM:

```
1 $> node --version
```

```
2 $> npm --version
```

Angular-cli

Angular-cli es la herramienta por línea de comandos que nos ofrece el equipo de Angular y que nos crea un proyecto completamente configurado para comenzar a trabajar.

Para la instalación de esta herramienta simplemente ejecutamos en un terminal:

```
1 $> npm install -g @angular/cli
```

¹¹<https://github.com/creationix/nvm>

¹²<https://github.com/coreybutler/nvm-windows/releases>

Editor de texto: Visual Studio Code

Necesitamos un editor de textos que nos proporcione ayuda a la hora de implementar nuestras funcionalidades. Actualmente el editor gratuito más completo es Visual Studio Code, el cual podemos descargar de la página [oficial](https://code.visualstudio.com/)¹³. Es un editor multiplataforma así que no tendrás problema en encontrar la versión que se ajuste a tu sistema operativo.

Descargamos la versión para Ubuntu y la instalamos como un paquete Debian. Ahora abrimos una instancia del programa a través del launcher y vamos a extensiones (icono de más abajo en la barra de la izquierda) donde vamos a instalar las siguientes:

- **TSLint by egamma (eg2.tslint):** nos señala en el propio código las violaciones de tslint que tengamos.
- **Angular Language Service (Angular.ng-template):** añade la capacidad de auto-completado y señalización de errores en los templates.
- **Types auto installer by Joao Vitor Paes de Barros do Carmo (jvitor83.types-autoinstaller):** descarga automáticamente los @types con las definiciones de las librerías que usamos.
- **Auto import by steoates (steoates.autoimport):** permite añadir automáticamente los imports necesarios a medida que vamos escribiendo en el código.
- **Sort TypeScript Imports by Michael Loughry (miclo.sort-typescript-imports):** extensión que permite organizar los imports de un fichero de manera determinista para que todo el equipo lo haga de la misma forma.
- **TypeScript Hero by Christoph Bühler (rbbit.typescript-hero):** extensión que permite borrar los imports de un fichero que no se estén utilizando.
- **vscode-icons by Roberto Huertas (robertohuertasm.vscode-icons):** añade un icono representativo a cada fichero para que se pueda identificar mejor de que trata.
- **bma-coverage by brunomartens (brunomartens.bma-coverage):** muestra en el propio código que líneas tienen o no cobertura de test.
- **Import Cost by wix (wix.vscode-import-cost):** esta extensión nos permite conocer el tamaño en bytes de los imports que hacemos de librerías de terceros. Muy útil para darnos cuenta de que librerías añaden más peso de forma visual.

Una vez terminado de instalar todos los plugins, reiniciamos el entorno para poder usarlos. Esto se puede hacer cerrando y abriendo el editor o pulsando en cualquiera de los botones “Enable” de los plugins instalados.

¹³<https://code.visualstudio.com/>

Chrome

Necesitamos un navegador para poder ejecutar y depurar nuestras aplicaciones con Angular. El mejor actualmente y más compatible con este tipo de tecnologías es Chrome, que además nos aporta las developer tools, un conjunto de herramientas que nos permiten depurar y optimizar las aplicaciones.

Para instalarlo en Ubuntu navegamos a la página [oficial](https://www.google.com/chrome/browser/desktop/index.html)¹⁴ y seleccionamos la versión con el paquete de Debian, la descargamos y la ejecutamos en nuestro escritorio. El acceso quedará en la sección “Internet” del menú principal.

Gestión del proyecto con Angular-Cli (v1.7.4)

Creación del proyecto

Una vez tenemos instaladas todas las herramientas necesarias ya estamos en disposición de crear nuestro primer proyecto.

Para ello abrimos un terminal, nos posicionamos en la ruta donde queramos almacenar nuestro proyecto y ejecutamos:

```
1 $> ng new nombre_proyecto [opciones]
```

Donde nombre_proyecto sería el nombre que le queremos dar al proyecto, por ejemplo, hello. En las opciones podemos definir:

- **-dry-run / -d (Boolean) (Default: false):** simula ejecución pero no la materializa.
- **-verbose / -v (Boolean) (Default: false):** muestra información por consola del proceso.
- **-collection:** permite definir una nueva “schematics collection” con la estructura por defecto que queramos crear. Similar al “archetype” de Maven.
- **-directory / -dir (String):** si queremos cambiar el directorio donde se crea la aplicación.
- **-inline-style / -is (Boolean) (Default: false):** si queremos que no se generen ficheros independientes con el estilo CSS.
- **-inline-template / -it (Boolean) (Default: false):** si queremos que no se generen ficheros independientes con los templates HTML.
- **-view-encapsulation (string):** permite especificar la estrategia de encapsulación por defecto (native, emulated o none).

¹⁴<https://www.google.com/chrome/browser/desktop/index.html>

- **-routing (Boolean) (Default: false):** si desde el principio sabemos que la aplicación va a necesitar de un router te lo genera automáticamente.
- **-prefix / -p (String) (Default: app):** si queremos darle un prefijo distinto a “app” a nuestra aplicación.
- **-style (String) (Default: css):** que tipo de hojas de estilo vamos a utilizar por defecto (css) pero podría ser SCSS o SASS.
- **-skip-tests / -st (Boolean) (Default: false):** si no queremos que genere automáticamente los ficheros .spec
- **-skip-install / -si (Boolean) (Default: false):** permite omitir el paso de instalar las dependencias definidas en package.json.
- **-skip-git / -sg (Boolean) (Default: false):** si no queremos que nos cree la configuración local de git.
- **-minimal (Boolean) (Default: false):** crea una aplicación con lo mínimo para ejecutar la aplicación, eliminando la configuración para los tests y con los templates y styles inline.
- **-service-worker (boolean):** instalar @angular/service-worker para facilitar la creación de aplicaciones PWA.

Una vez finaliza el proceso podemos situarnos dentro de la carpeta recién creada y ejecutar una instancia de Visual Studio Code, de esta manera:

```
1 $> cd nombre_proyecto
2 $> code .
```

Con el editor abierto vamos a “View” -> “Integrated Terminal” para habilitar el terminal embebido y no tener que estar cambiando de pantalla para ejecutar los comandos.

Ejecución de la aplicación en desarrollo

Para la ejecución de la aplicación en desarrollo tenemos que ejecutar el siguiente comando:

```
1 $> ng serve [opciones]
```

Este comando realiza una build del proyecto y publica el resultado en el puerto 4200. De forma que si abrimos una instancia de Chrome y navegamos a la url: <http://localhost:4200>¹⁵ veremos el resultado de la ejecución del proyecto en desarrollo.

Como estamos en desarrollo podemos hacer cualquier cambio en la aplicación que inmediatamente se reflejará en el navegador.

Podemos definir las siguientes opciones:

¹⁵<http://localhost:4200>

- **-target / -t (String) (Default: Development):** para cambiar el modo de generar los ficheros, si queremos que sea en modo desarrollo (-dev) o si queremos que ya estén optimizados para producción (-prod)
- **-environment (String):** para que definir el fichero de variables de entorno a tener en cuenta. Los ficheros de entorno se encuentran en el directorio “environments”.
- **-output-path / -op (Path):** podemos cambiar la ruta donde se almacenan los ficheros de salida.
- **-aot / -aot (Boolean):** para hacer uso de Ahead of Time Compilation
- **-sourcemaps / -sm (Boolean):** para generar los ficheros .map que nos permiten depurar mejor nuestro código, ver qué dependencias se están utilizando y que peso aportan al bundle.
- **-vendor-chunk / -vc (Boolean):** genera un bundle separado con todas las librerías de terceros de nuestro proyecto.
- **-common-chunk / -cc (Boolean) (Default: true):** separa en bundles código que es utilizado en múltiples bundles.
- **-base-href / -bh (String):** nos permite cambiar el meta href del index.html para ajustarlo apropiadamente allá donde se vaya a ejecutar la aplicación.
- **-deploy-url / -d (String):** define la URL donde se van a desplegar los ficheros.
- **-verbose / -v (Boolean) (Default: false):** añade más detalle al log de salida.
- **-progress / -pr (Boolean) (Default: true):** muestra el progreso de la fase de despliegue.
- **-i18n-file (String):** define la localización de los ficheros de i18n.
- **-i18n-format (String):** define el formato de i18n de los ficheros especificados en el anterior.
- **-locale (String):** define el locale para el i18n.
- **-missing-translation (String):** permite establecer cómo manejar las traducciones olvidadas.
- **-extract-css / -ec (Boolean):** extrae a ficheros css el estilo que de otra forma vendría incluido en los ficheros .js
- **-watch (Boolean) (Default: true):** hace el rebuild después de cualquier cambio de código.
- **-output-hashing=none|all|media|bundles / -oh (String):** define el formato de hash para los ficheros de salida a fin de evitar el problema de cacheo en los navegadores.
- **-poll / -poll (Number):** permite definir el periodo de tiempo del watch para hacer el polling.
- **-app / -a (String):** especifica el nombre de la aplicación o el index a utilizar, cuando tenemos más de una aplicación definida en el proyecto.
- **-delete-output-path / -dop (Boolean) (Default: true):** borra la carpeta de output antes de hacer la build.

- **-preserve-symlinks (Boolean) (Default: false):** no utiliza el path real cuando resuelve los módulos.
- **-extract-licenses (Boolean) (Default: true):** solo en producción, extrae todas las licencias a un fichero separado.
- **-show-circular-dependencies (Boolean) (Default: true):** muestra los avisos de dependencias circulares.
- **-build-optimizer (Boolean):** establece las optimizaciones de compilación cuando se acompaña de **-aot**
- **-named-chunks (Boolean):** si se establece muestra el nombre del módulo lazy en el nombre del fichero generado para el chunk.
- **-subresource-integrity / -sri (Boolean) (Default: false):** permite habilitar la validación de integridad del subrecurso.
- **-bundle-dependencies (none, all) (Default: none):** solo está disponible en la plataforma de servidor. Incluye todas las dependencias externas en un bundle dentro del módulo. Por defecto, todo `node_modules` se mantendrá como requerido.
- **-service-worker (Boolean) (Default: true):** genera la configuración de service worker para producción, si la aplicación tiene el flag de service worker habilitado.
- **-skip-app-shell (Boolean) (Default: false):** flag que previene la construcción de la app shell.
- **-port / -p (Number) (Default: 4200):** permite cambiar el número de puerto donde se despliega la aplicación.
- **-host / -H (String) (Default: localhost):** permite modificar el host donde se despliega la aplicación. Para permitir acceder desde cualquier dispositivo con conexión tenemos que utilizar la ip 0.0.0.0
- **-proxy-config / -pc (Path):** define la configuración de proxy necesaria en algunos casos para evitar el CORS.
- **-ssl / -ssl(Boolean) (Default: false):** levanta la aplicación con https.
- **-ssl-key (String) (Default: ssl/server.key):** define el fichero key para servir https.
- **-ssl-cert (String) (Default: ssl/server.crt):** define el certificado para servir https.
- **-open / -o (Boolean) (Default: false):** para abrir automáticamente el navegador.
- **-live-reload / -lr (Boolean) (Default: true):** para que una rebuild provoque automáticamente el refresco del navegador.
- **-public-host (String):** especifica la URL que el navegador usará.
- **-disable-host-check (Boolean) (Default: false):** No verifica si los clientes conectados pertenecen a una lista de hosts permitidos.
- **-serve-path (String):** permite definir el pathname donde se va a servir la aplicación, muy útil cuando queremos desplegarla en los GHPAGES de GitHub, por ejemplo.
- **-hmr / -hrm (Boolean) (Default: false):** habilita el cambio de módulos en caliente.

Ejecución de los tests unitarios

Esta herramienta ya nos proporciona un entorno correctamente configurado con Jasmine y Karma para la ejecución de todos los tests unitarios y de integración, para lo que simplemente tenemos que ejecutar:

```
1 $> ng test [opciones]
```

Esto hace que por consola se muestre el resultado de la ejecución de los tests unitarios que tengamos implementados.

Cualquier cambio que queramos hacer en la configuración, por ejemplo, para definir otro navegador o hacer que no se quede esperando cambios en los ficheros de tests para volver a ejecutarse, lo podemos realizar en el fichero karma.conf.js de la raíz del proyecto. Podemos especificar algunos de estos cambios a través de las siguientes opciones:

- **-watch / -w (Boolean) (Default: true):** si es true, o simplemente se indica el flag, determinamos que se vuelva a ejecutar cuando haya algún cambio de código en los tests.
- **-code-coverage / -cc (Boolean) (Default: false):** si queremos obtener un informe de cobertura.
- **-config / -c (String):** permite cambiar la ruta del fichero de configuración de karma. ** **
- **-single-run (Boolean):** si es true ejecuta los tests y detiene el proceso, si es false no detiene el proceso.
- **-progress (Boolean):** muestra el progreso por consola.
- **-browsers (String):** si queremos definir otro navegador no esté configurado en el fichero de configuración. Lo que si vamos a necesitar es tener los plugins necesarios instalados y definidos.
- **-colors (Boolean):** si queremos o no mostrar el texto con colores.
- **-log-level (String):** si queremos cambiar el nivel de log de las trazas para una determinada ejecución
- **-port / -port(Number):** si queremos cambiar el puerto que viene por defecto.
- **-reporters (String):** si queremos especificar el reporter que queremos ejecutar.
- **-sourcemaps / -sm (Boolean) (Default: true):** genera los ficheros .map asociados
- **-poll (Number):** define el periodo de watching.
- **-environment / -e (String):** define el entorno de compilación.
- **-preserve-symlinks (Boolean) (Default: false):** no usa los path reales cuando resuelve los módulos.
- **-app / -a (String):** especifica la aplicación a ejecutar los tests, cuando hay más de una definida en el proyecto.

Ejecución de los tests de aceptación o e2e

De la misma forma también viene preparada para la ejecución de los tests de aceptación o e2e que tengamos implementados, simplemente tenemos que ejecutar:

```
1 $> ng e2e [opciones]
```

Podemos utilizar cualquiera de las opciones vistas en `serve`, más las siguientes:

- **-config / -c (String)**: permite definir el fichero de configuración. Por defecto, tomará el que define `angular-cli`.
- **-specs / -sp (Array) (Default:)**: permite definir los specs a ejecutar, se puede añadir múltiples (`ng e2e -specs=spec1.ts -specs=spec2.ts`)
- **-suite (String)**: Sobrescribe la propiedad `suite` de la configuración de Protractor. Permite indicar más de un suite separado por comas.
- **-element-explorer / -ee (Boolean) (Default: false)**: permite habilitar los `element explorer` para debug.
- **-webdriver-update / -wu (Boolean) (Default: true)**: actualiza `webdriver`.
- **-serve / -s (Boolean) (Default: true)**: compila y levanta la aplicación. Cualquier comando de `serve` es aceptado.

Cualquier cambio en la configuración de Protractor lo podremos hacer en el fichero `protractor.conf.js` de la raíz del proyecto o el que hayamos especificado en la propiedad `config`.

Creación de elementos

La herramienta nos ofrece código ya preconfigurado para reducir el “boilerplate” y ayudarnos a centrarnos en la lógica de nuestra aplicación. Para ello ejecutamos en el terminal el comando:

```
1 $> ng g tipo opciones
```

Donde el tipo y las opciones pueden ser:

- **application**: permite crear una nueva aplicación dentro de nuestro proyecto. Podemos crear `n` aplicaciones dentro de un mismo proyecto. Podemos verlas definidas dentro del fichero `.angular-cli.json`
 - **-dry-run / -d**: ejecuta los comandos sin materializar los cambios.
 - **-force / -f**: sobrescribe todos los ficheros.

- **-app / -a**: especifica el nombre de la nueva aplicación.
- **-collection / -c**: especifica el “schematics” a utilizar para la nueva aplicación.
- **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.
- **-directory / -dir (string)**: especifica el nombre del directorio donde crear la nueva aplicación.
- **-inline-style / -is (boolean)**: no crea un fichero separado para los estilos del componente.
- **-inline-template / -it (boolean)**: no crea un template separado con la estructura del componente.
- **-view-encapsulation (string)**: especifica la estrategia de encapsulación del componente (native, emulated o none).
- **-version (string)**: permite definir la versión de Angular CLI a usar.
- **-routing (boolean)**: genera el módulo de routing.
- **-prefix / -p (string) (Default: app)**: especifica el prefijo para los selectores.
- **-style (string) (Default: css)**: especifica la extensión de los ficheros de estilo puede ser css o sass.
- **-skip-tests / -st (boolean)**: no crea el fichero separado con la estructura del .spec correspondiente.
- **-skip-install (boolean)**: no realiza la instalación de las dependencias.
- **-skip-git / -sg (boolean)**: no crea la configuración de git.
- **-minimal (boolean)**: crea una aplicación con el contenido mínimo, sin tests y con template y style inline.
- **-service-worker (boolean)**: instala @angular/service-worker.
- **class / cl**
 - **-dry-run**: ejecuta los comandos pero sin materializar los cambios.
 - **-force / -f**: sobrescribe todos los ficheros.
 - **-app / -a (String)**: permite definir en qué aplicación de las múltiples que haya irá la clase.
 - **-collection / -c**: especifica el “schematics” a utilizar para la nueva aplicación.
 - **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.
 - **-spec (Boolean)**: si es true crea el fichero .spec asociado.
 - **-type (string)**: especifica el tipo de clase.
- **component / c**
 - **-dry-run**: ejecuta los comandos pero sin materializar los cambios.
 - **-force / -f**: sobrescribe todos los ficheros.
 - **-app / -a (String)**: especifica el nombre de la app a utilizar, cuando en el proyecto hay más de una.
 - **-collection / -c**: especifica el “schematics” a utilizar para la nueva aplicación.
 - **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.

- **-inline-style / -is (Boolean)**: si es true no se crea un fichero CSS separado.
- **-inline-template / -it (Boolean)**: si es true no se crea un fichero .html separado.
- **-view-encapsulation (string)**: especifica la estrategia de encapsulación del componente (native, emulated o none).
- **-change-detection / -cd (String)**: especifica la estrategia de change detection del componente. Puede ser: OnPush.
- **-prefix (String) (Default: null)**: si queremos especificar al selector un prefijo distinto al definido en la aplicación.
- **-styleext (string) (Default: css)**: especifica la extensión de fichero para los ficheros de estilo. Podría ser: css o sass
- **-spec (Boolean)**: si queremos crear el fichero .spec con los tests asociados.
- **-flat (Boolean)**: si es true no se crea un directorio con los ficheros asociados.
- **-skip-import (Boolean) (Default: false)**: no establece forma automática la importación en el correspondiente módulo.
- **-selector (string)**: especifica el selector que va a utilizar el componente.
- **-module / -m (String)**: permite especificar el módulo asociado.
- **-export (Boolean) (Default: false)**: permite decidir si queremos establecer el componente en la sección exports del módulo asociado.
- **directive / d**
 - **-dry-run**: ejecuta los comandos pero sin materializar los cambios.
 - **-force / -f**: sobrescribe todos los ficheros.
 - **-app (String)**: cuando se tiene más de una aplicación en el proyecto, permite especificar la aplicación sobre la que queremos actuar.
 - **-collection / -c**: especifica el "schematics" a utilizar para la nueva aplicación.
 - **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.
 - **-prefix (String) (Default: null)**: especifica el prefijo utilizado en el selector.
 - **-spec (Boolean)**: indica que si se tiene que crear el fichero .spec asociado.
 - **-skip-import (Boolean) (Default: false)**: permite decidir que si se quiere añadir el import en el módulo correspondiente.
 - **-selector (string)**: especifica el selector que va a utilizar el componente.
 - **-flat (Boolean)**: si es true no se crea directorio asociado.
 - **-module / -m (String)**: permite definir el módulo donde queremos crear la directiva.
 - **-export (Boolean) (Default: false)**: especifica si la directiva se tiene que añadir al conjunto de elementos de la propiedad exports.
- **enum / e**
 - **-dry-run**: ejecuta los comandos pero sin materializar los cambios.
 - **-force / -f**: sobrescribe todos los ficheros.
 - **-app (String)**: cuando se tiene más de una aplicación en el proyecto, permite especificar la aplicación sobre la que queremos actuar.

- **-collection / -c**: especifica el “schematics” a utilizar para la nueva aplicación.
- **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.
- **guard / g**
 - **-dry-run**: ejecuta los comandos pero sin materializar los cambios.
 - **-force / -f**: sobrescribe todos los ficheros.
 - **-app (String)**: cuando se tiene más de una aplicación en el proyecto, permite especificar la aplicación sobre la que queremos actuar.
 - **-collection / -c**: especifica el “schematics” a utilizar para la nueva aplicación.
 - **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.
 - **-spec (Boolean)**: si es true se crea el fichero .spec asociado.
 - **-flat (Boolean)**: indica si se tiene que crear el directorio.
 - **-module / -m (String)**: especifica el módulo donde se va a crear el provider asociado.
- **interface / -i**
 - **-dry-run**: ejecuta los comandos pero sin materializar los cambios.
 - **-force / -f**: sobrescribe todos los ficheros.
 - **-app (String)**: cuando se tiene más de una aplicación en el proyecto, permite especificar la aplicación sobre la que queremos actuar.
 - **-collection / -c**: especifica el “schematics” a utilizar para la nueva aplicación.
 - **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.
 - **-prefix (string)**: especifica el prefijo a utilizar.
 - **-type (string)**: especifica el tipo de interface.
- **module / -m**
 - **-dry-run**: ejecuta los comandos pero sin materializar los cambios.
 - **-force / -f**: sobrescribe todos los ficheros.
 - **-app (String)**: cuando se tiene más de una aplicación en el proyecto, permite especificar la aplicación sobre la que queremos actuar.
 - **-collection / -c**: especifica el “schematics” a utilizar para la nueva aplicación.
 - **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.
 - **-routing (Boolean) (Default: false)**: indica si se tiene que crear un módulo de routing asociado.
 - **-routing-scope (string) (Default: Child)**: especifica el scope para el router generado.
 - **-spec (Boolean)**: especifica si se tiene que crear el fichero .spec asociado.
 - **-flat (Boolean)**: si es true crea un directorio asociado.
 - **-common-module (boolean) (Default: true)**: flag que controla si se quiere o no añadir el import de CommonModule al módulo especificado.
 - **-module / -m (String)**: especifica donde el módulo debería ser importado.

- **pipe / -p**
 - **-dry-run**: ejecuta los comandos pero sin materializar los cambios.
 - **-force / -f**: sobrescribe todos los ficheros.
 - **-app (String)**: cuando se tiene más de una aplicación en el proyecto, permite especificar la aplicación sobre la que queremos actuar.
 - **-collection / -c**: especifica el “schematics” a utilizar para la nueva aplicación.
 - **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.
 - **-flat (Boolean)**: indica si se tiene que crear el directorio asociado.
 - **-spec (Boolean)**: indica si se tiene que crear el fichero .spec asociado.
 - **-skip-import (Boolean) (Default: false)**: permite decidir si se quiere o no hacer el import automático en el módulo asociado.
 - **-module / -m (String)**: permite especificar el módulo asociado.
 - **-export (Boolean) (Default: false)**: especifica si el pipe se debe añadir como elemento de la propiedad exports.
- **service**
 - **-dry-run**: ejecuta los comandos pero sin materializar los cambios.
 - **-force / -f**: sobrescribe todos los ficheros.
 - **-app (String)**: cuando se tiene más de una aplicación en el proyecto, permite especificar la aplicación sobre la que queremos actuar.
 - **-collection / -c**: especifica el “schematics” a utilizar para la nueva aplicación.
 - **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.
 - **-flat (Boolean)**: indica si se tiene que crear el directorio asociado.
 - **-spec (Boolean)**: indica si se tiene que crear el fichero .spec asociado.
 - **-module / -m (String)**: permite especificar el módulo asociado.
- **universal**
 - **-dry-run**: ejecuta los comandos pero sin materializar los cambios.
 - **-force / -f**: sobrescribe todos los ficheros.
 - **-app (String)**: cuando se tiene más de una aplicación en el proyecto, permite especificar la aplicación sobre la que queremos actuar.
 - **-collection / -c**: especifica el “schematics” a utilizar para la nueva aplicación.
 - **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.
 - **-client-app (string) (Default: 0)**: indica el nombre o el número de índice de la aplicación cliente relacionada.
 - **-app-id (string) (Default: serverApp)**: especifica el “appId” que se va a utilizar en la definición de “withServerTransition”.
 - **-out-dir (string) (Default: dist-server/)**: especifica el directorio de salida del resultado de la build.
 - **-root (string) (Default: src)**: especifica el directorio raíz de la aplicación.
 - **-index (string) (Default: index.html)**: especifica el nombre del fichero html

principal.

- **-main (string) (Default: main.server.ts)**: especifica el nombre del fichero de entrada principal.
- **-test (string)**: especifica el nombre del test del fichero de entrada principal.
- **-tsconfig-file-name (string) (Default: tsconfig.server)**: especifica el nombre del fichero de configuración de TypeScript.
- **-test-tsconfig-file-name (string) (Default: tsconfig.spec)**: especifica el nombre del fichero de configuración de TypeScript para los tests.
- **-app-dir (string) (Default: app)**: especifica el nombre del directorio de la aplicación.
- **-root-module-file-name (string) (Default: app.server.module.ts)**: especifica el nombre del archivo del módulo principal.
- **-root-module-class-name (string) (Default: AppServerModule)**: especifica el nombre del módulo principal.

- **appShell**

- **-dry-run**: ejecuta los comandos pero sin materializar los cambios.
- **-force / -f**: sobrescribe todos los ficheros.
- **-app (String)**: cuando se tiene más de una aplicación en el proyecto, permite especificar la aplicación sobre la que queremos actuar.
- **-collection / -c**: especifica el “schematics” a utilizar para la nueva aplicación.
- **-lint-fix / -lf**: utiliza lint para solucionar los problemas de estilo de los ficheros después de generarlos.
- **-client-app (string) (Default: 0)**: indica el nombre o el número de índice de la aplicación cliente relacionada.
- **-universal-app (string)**: especifica el nombre o el número de índice de la aplicación de tipo univesal relacionada.
- **-route (string) (Default: shell)**: especifica el path del router usado para generar la app shell.
- **-app-id (string) (Default: serverApp)**: especifica el “appId” que se va a utilizar en la defición de “withServerTransition”.
- **-out-dir (string) (Default: dist-server/)**: especifica el directorio de salida del resultado de la build.
- **-root (string) (Default: src)**: especifica el directorio raíz de la aplicación.
- **-index (string) (Default: index.html)**: especifica el nombre del fichero html principal.
- **-main (string) (Default: main.server.ts)**: especifica el nombre del fichero de entrada principal.
- **-test (string)**: especifica el nombre del test del fichero de entrada principal.
- **-tsconfig-file-name (string) (Default: tsconfig.server)**: especifica el nombre del fichero de configuración de TypeScript.
- **-test-tsconfig-file-name (string) (Default: tsconfig.spec)**: especifica el nombre del fichero de configuración de TypeScript para los tests.

- **–app-dir (string) (Default: app):** especifica el nombre del directorio de la aplicación.
- **–root-module-file-name (string) (Default: app.server.module.ts):** especifica el nombre del archivo del módulo principal.
- **–root-module-class-name (string) (Default:AppServerModule):** especifica el nombre del módulo principal.

Descubrimiento de errores de estilo y codificación

Para ver los errores de estilo que podamos cometer en el momento de desarrollar los distintos componentes, contamos con el plugin TSLint que nos marca y nos ayuda a corregir los distintos errores en el propio editor. Angular CLI ya viene configurado con la librería [Codelyzer](http://codelyzer.com/)¹⁶, que mantiene actualizadas todas las reglas de estilo que aplica el equipo de Angular.

En caso de no tener editor podemos ver un informe de los errores por consola ejecutando el comando:

```
1 $> ng lint
```

Este comando no se suele utilizar por línea de comando, lo que se hace es instalar la extensión TSLint que hace que Visual Code marque los errores por pantalla.

Además podemos configurar la extensión para que automáticamente aplique las correcciones que pueda al código, de forma que no lo tengamos que hacer a mano. Para ello editamos las settings de Visual Studio Code y añadimos la propiedad:

```
1 {  
2     "tslint.autoFixOnSave": true  
3 }
```

Construcción del proyecto

La fase de construcción del proyecto es la encargada de traducir nuestros ficheros TypeScript a código JavaScript que entienda el navegador, entre otras cosas. Para ejecutar una build simplemente ejecutamos el comando:

```
1 $> ng build [opciones]
```

¹⁶<http://codelyzer.com/>

Este comando creará una carpeta “dist” en el proyecto con el resultado de la build. Las distintas opciones que admite este comando son las mismas que serve pero sin las relativas a live-reload, además añade estas propias:

- **–stats-json (Boolean) (Default: false):** genera el fichero stats.json necesario para analizar la aplicación con la herramienta: webpack-bundle-analyzer.

Acceder a la documentación

Cuando queramos acceder directamente a la documentación de algún elemento en particular podemos utilizar el comando:

```
1 $> ng doc <keyword>
```

Esto abrirá un navegador con los resultados de la búsqueda por keyword en la documentación oficial de Angular.

Consejo para la ejecución de los comandos

Si te das cuenta la mayoría de comandos vistos hasta ahora se han ejecutado directamente con ng, lo que hace que se ejecute la instancia de angular-cli que tengamos instalada de forma global.

Esto es una mala práctica ya que podemos estar ejecutando los comandos con una versión distinta a la que tengamos definida en el package.json de nuestra aplicación. Imagina que descargas un proyecto hecho con angular-cli de tu empresa o de un tercero, este proyecto va con una versión específica que tenemos que respetar para no tener incongruencias difíciles de depurar.

La clave es ejecutar siempre la versión local del proyecto y esto se consigue con el comando npm seguido de “run” y el comando de la sección de “scripts” que queramos ejecutar seguido de “–” y las opciones permitidas para ese comando.

Así si, por ejemplo, queremos crear un nuevo componente en vez de ejecutar:

```
1 $> ng generate component foo
```

Haremos:

```
1 $> npm run ng -- generate component foo
```

Esto lo podemos aplicar con cualquier comando definido en la sección “scripts” del fichero package.json. Ahora si necesitamos incluir otros procesos ya sea con gulp, grunt, webpack, empaquetado con docker, ... es buena práctica crear un nuevo comando en la sección de scripts de forma que solo te tienes que acordar del comando npm y no del detalle de cada uno de ellos.

Así si queremos arrancar la aplicación solo tendremos que ejecutar:

```
1 $> npm run start
```

De esta forma en un proyecto más o menos complejo podríamos encontrarnos con una sección de script similar a esta:

```
1  "scripts": {  
2    "ng": "ng",  
3    "start": "ng serve -e int",  
4    "build": "ng build",  
5    "test": "ng test --code-coverage",  
6    "test-it": "ng test --config=karma.conf.it.js --single-run",  
7    "test:ci": "xvfb-run npm run all-tests -- --browser=Headless_Chrome --code-co\  
8 verage=true",  
9    "lint": "ng lint",  
10   "e2e": "ng e2e",  
11   "e2e-env": "protractor protractor.conf.js --baseUrl",  
12   "e2e:ci": "xvfb-run npm run e2e",  
13   "ship": "ng build --prod && export APP_VERSION=${version} && docker-compose b\  
14 uild",  
15   "rest": "cd ../api-rest; mvn spring-boot:run -Drun.jvmArguments='-Dserver.port\  
16 t=8282'",  
17   "dev": "npm run rest && npm run mock && npm start",  
18   "compodoc": "compodoc -p src/tsconfig.app.json",  
19   "build:prod": "ng build --aot --build-optimizer",  
20   "all-tests": "npm run lint && npm run test -- --single-run && npm run test-it\  
21 ",  
22   "sm": "source-map-explorer",  
23   "release": "generate-release",  
24   "precommit": "npm run lint"  
25 }
```

Módulos

Casi in-extremis la versión final de Angular ha incluido el tan necesario concepto de módulo, que permite estructurar y organizar la aplicación en áreas funcionales a fin de que puedan ser reutilizadas en otras aplicaciones.

El propio framework modulariza todas sus funcionalidades de forma que solo tenemos que importar, en el módulo principal de la aplicación, los elementos del framework que necesitamos, es decir, si nuestra aplicación no cuenta con ningún formulario, no vamos a incluir el FormsModule, pero si hacemos uso de llamadas externas si vamos a incluir el HttpClientModule. Con esto conseguimos que el bundle resultante ocupe lo menos posible.

@NgModule

Cualquier módulo en Angular, ya sea principal (root) o secundario (feature), se define con el decorador @NgModule y tiene las siguientes propiedades:

- **providers:** donde se define un array con las clases que pueden ser inyectadas en este módulo o en sus hijos.
- **declarations:** especifica un array de directivas y pipes que pertenecen al módulo y no pueden estar declarados en ningún otro módulo relacionado. Si esto ocurre Angular nos informará del error en tiempo de ejecución.
- **imports:** donde especificamos la lista de módulos de los que vamos a hacer uso dentro de este módulo, ya sean del framework, propios o de terceros.
- **exports:** es la lista de directivas y pipes que permitimos que puedan ser utilizados por otros módulos que importen a éste. Solo tiene sentido en módulos secundarios.
- **entryComponents:** define la lista de componentes que serán compilados cuando el módulo sea definido. En este caso Angular crea de forma transparente un ComponentFactory por cada uno de ellos y los almacena en un ComponentFactoryResolver para mejorar el rendimiento. Es imprescindible cuando queremos carga dinámica de componentes en la aplicación.
- **bootstrap:** define la lista de componentes principales del módulo. Estos componentes automáticamente se añaden a la lista de entryComponents de forma transparente al desarrollador.
- **schemas:** se utiliza para definir propiedades que no son de Angular, por ejemplo, si queremos incluir web components de terceros debemos indicar la propiedad CUSTOM_ELEMENTS_SCHEMA.

Este sería un ejemplo de definición de un módulo principal:

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule, CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';
3 import { FormsModule } from '@angular/forms';
4 import { HttpClientModule } from '@angular/http';
5 import { AppComponent } from './app.component';
6 import { UserService } from './user.service';
7
8 @NgModule({
9   providers: [ AppService ],
10  declarations: [ AppComponent ],
11  imports: [ BrowserModule, FormsModule, HttpClientModule ],
12  entryComponents: [ OtherAppComponent ],
13  bootstrap: [ AppComponent ],
14  schemas: [CUSTOM_ELEMENTS_SCHEMA]
15 })
16
17 export class AppModule {}
```

Creación de un módulo secundario

A medida que una aplicación va creciendo se van añadiendo más y más elementos (componentes, servicios, pipes, ...) al módulo principal, en este momento, tenemos que ver cuáles de estos elementos están relacionados entre sí a través de una misma funcionalidad y crear con ellos módulos secundarios a fin de poder ser importados en cualquier módulo principal de nuestra aplicación o de cualquier otra.

La diferencia con un módulo principal es que un módulo secundario tiene que ser independiente de la plataforma por lo tanto no tendrá la dependencia con el módulo `BrowserModule` sino que utilizará el `CommonModule`, además hay que tener en cuenta que todos los elementos de un módulo son por defecto privados, por lo tanto si queremos que puedan ser utilizados desde otros módulos que lo importen, tenemos que establecerlos en la propiedad **exports** de la definición del módulo secundario.

Por ejemplo, podríamos encapsular en un módulo la funcionalidad de acceso a un determinado API de acceso a los datos de usuario de GitHub. Para ello crearíamos una interfaz `User` con los datos de dominio, un servicio proxy de acceso al API REST (`GitHubProxyService`), un servicio de adaptación de los datos recuperados con el dominio de la aplicación (`UserService`) y un componente de visualización de datos (`UserComponent`).

Con estos elementos crearíamos el correspondiente módulo secundario de esta forma:

```
1 @NgModule({
2   imports: [CommonModule],
3   declarations: [UsersComponent],
4   providers: [UserService, GithubProxyService],
5   exports: [UsersComponent]
6 })
7
8 export class UsersModule { }
```

En este caso permitimos, que al importar el módulo en otro, solo se pueda acceder al componente de visualización de datos, dejando el resto de elementos en el funcionamiento interno del módulo.

Esta característica hace que cualquier funcionalidad y lógica de negocio sea altamente reutilizable en otras aplicaciones pero sin comprometer el funcionamiento original.

A medida que la aplicación crece en el número de módulos tenemos que tener cuidado con la jerarquía que se establece entre ellos, sobre todo, a la hora de hacer uso de elementos comunes, ya que el compilador de Angular se quejará cuando detecte que dos módulos declaren el mismo elemento. La solución más sencilla es declarar un nuevo módulo llamado “SharedModule” que declare estos elementos compartidos (pipes, directivas, componentes) e importarlo en los módulos en los que sean necesarios.

Arranque de la aplicación automático

Toda aplicación Angular al menos tiene que tener un módulo principal que será el que se defina en el arranque de la aplicación y tendrá indicado en la propiedad bootstrap cuál es el componente que tiene que cargar en base al selector que se establezca en el fichero index.html.

Si vamos al detalle, cuando arrancamos una aplicación en el scope de angular, lo primero que hace el navegador es cargar el fichero main.ts, que tendrá este contenido:


```
1  import { enableProdMode } from '@angular/core';
2  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
3
4  import { AppModule } from './app/app.module';
5  import { environment } from './environments/environment';
6
7  if (environment.production) {
8    enableProdMode();
9  }
10
11 platformBrowserDynamic().bootstrapModule(AppModule)
12   .catch(err => console.log(err));
```

En este fichero se indica cuál es la plataforma específica con la que estamos trabajando, el navegador en nuestro caso, y a través de la función `bootstrapModule` definimos cuál de nuestros módulos es el principal, en este caso `AppModule`, que se implementa en el fichero `app.module.ts` con el siguiente contenido:

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4
5  import { AppComponent } from './app.component';
6
7
8  @NgModule({
9    declarations: [
10      AppComponent
11    ],
12    imports: [
13      BrowserModule
14    ],
15    providers: [],
16    bootstrap: [AppComponent]
17  })
18  export class AppModule { }
```

El proceso de arranque lee la propiedad `bootstrap` donde determinamos cuál es el componente principal de nuestra aplicación y que se va a implementar en la clase `AppComponent` en el fichero `app.component.ts` con el siguiente contenido:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'app';
10 }
```

El proceso de arranque lee el selector de nuestro componente principal y trata de encontrarlo en el fichero index.html de nuestra aplicación. Si no lo encuentra lanza el siguiente error:

The selector “app-root” did not match any elements

En caso de encontrarlo renderiza el template del componente principal mostrando el mensaje de bienvenida en pantalla.

Arranque manual de la aplicación

Nos tenemos que dar cuenta que el index.html no es parte de nuestra aplicación Angular, es simplemente el contenedor donde instanciamos nuestra aplicación. Por lo que nuestra aplicación podría ser instanciada en cualquier otro contenedor como un portal de aplicaciones tipo Liferay o una página estática de Wordpress.

Pero claro ya hemos visto que si el arranque no encuentra en el DOM el selector de nuestro componente principal, nos mostrará un error en todas las páginas del portal donde no esté definido nuestro selector.

Para resolver esta situación tenemos que modificar la implementación del fichero app.module de esta forma:

```
1  import { BrowserModule, DOCUMENT } from '@angular/platform-browser';
2  import { NgModule, ApplicationRef, Inject } from '@angular/core';
3  import { AppComponent } from './app.component';
4
5  @NgModule({
6    declarations: [
7      AppComponent
8    ],
9    imports: [
10     BrowserModule
11   ],
12   providers: [ ],
13   entryComponents: [AppComponent],
14   schemas: []
15 })
16
17 export class AppModule {
18
19   private browser_document;
20
21   ngDoBootstrap(appRef: ApplicationRef) {
22     if (this.browser_document.getElementsByTagName('app-root').length > 0) {
23       appRef.bootstrap(AppComponent);
24     }
25   }
26
27   constructor(@Inject(DOCUMENT) private document: any) {
28     this.browser_document = document;
29   }
30
31 }
```

Como se ve hemos eliminado la propiedad “bootstrap” de la declaración del módulo principal y dentro de la lógica de la clase hemos hecho uso de la función `ngDoBootstrap` donde se implementa nuestra lógica y cuando se cumple la condición de que el selector existe, se llama manualmente a la función `bootstrap` del elemento de tipo `ApplicationRef` indicando el componente principal.

Es muy importante, si no el navegador nos arrojará un error, que declaremos el componente en la propiedad “**entryComponents**” de la declaración del módulo principal, ya que esto es lo que hace implícitamente Angular cuando lo declaramos en el array de `bootstrap`.

Directivas

En Angular tenemos tres tipos de directivas.

- Componentes
- Directivas de atributo
- Directivas estructurales

Componentes

Son directivas con template. Los componentes se definen como un conjunto de elementos HTML que forman un único elemento común con su propia lógica. Son partes reutilizables de una aplicación que están formados por dos partes diferenciadas: la vista y la lógica de negocio.

En todo componente vamos a diferenciar dos partes: la definición del componente que vendrá identificada por el decorador **@Component** y la parte de la lógica que será la clase y vendrá identificada por la palabra reservada **"class"**.

En el decorador **@Component** podemos definir las siguientes propiedades obligatorias:

- **selector:** define el nombre de la etiqueta HTML con el que le vamos a hacer referencia, es buena práctica que este nombre contenga un guión y un identificador corporativo para evitar futuras colisiones.
- **template o templateUrl:** que contendrá el contenido HTML que mostrará el componente al renderizarse.

Los componentes tienen un ciclo de vida del cual podemos hacer hook y meter código gracias a estas funciones que por defecto tenemos disponibles y que tienen que implementar cada una de las interfaces, entre paréntesis el nombre del método a implementar:

- **OnChanges (ngOnChanges):** se llama cuando un valor de output o input cambia lanzando un evento.
- **OnInit (ngOnInit):** se llama después del primer OnChanges y del constructor.
- **DoCheck (ngDoCheck):** se lanza después de cambiar un valor o una dirección de memoria (se puede personalizar el change detection).
 - **AfterContentInit (ngAfterContentInit):** se llama justo después de inicializar el componente.

- **AfterContentChecked (ngAfterContentChecked)**: después de cada check del contenido del componente.
- **AfterViewInit (ngAfterViewInit)**: después de que la vista del componente haya sido inicializada.
- **AfterViewChecked (ngAfterViewChecked)**: después de cada check del contenido de la vista.
- **OnDestroy (ngOnDestroy)**: justo antes de que la vista haya sido destruida.

Directivas de atributo

Son un tipo de directiva sin template que se utiliza para cambiar la apariencia o el comportamiento de un elemento del DOM. Angular proporciona varias de serie como: `ngClass`, `ngStyle` y `ngModel` (de la última hablaremos en un capítulo siguiente), además de permitirnos la creación de propias.

ngClass y class binding

Esta directiva nos permite cambiar de forma dinámica la asociación de clases CSS a un elemento HTML determinado.

En este primer ejemplo, dependiendo del valor de la variable “errorCount” se va a establecer una clase u otra en el elemento `div`.

```
1 <div [ngClass]="{active: errorCount > 0, inactive: errorCount=0}>  
2   Contenido  
3 </div>
```

En caso de querer solo establecer una clase en base a una propiedad del modelo, sería mejor hacerlo con class binding.

```
1 <div [class.special]="isSpecial">Contenido</div>
```

Solo se va a establecer la clase “special” cuanto el atributo “isSpecial” del componente sea true.

Esta directiva no nos permite modificar el valor del CSS, solo aplicar una clase determinada que debe estar definida en CSS, ya sea en la propiedad `styles` o `styleUrls` del componente o en el fichero principal de la aplicación, siempre y cuando la encapsulación del componente lo permita.

ngStyle y style binding

Es similar a la anterior pero añadiendo al elemento HTML de forma dinámica contenido a la propiedad `style`.

```
1 <h1 [ngStyle]="{'font-style': style, 'font-size': size, 'font-weight': weight}">
2     Contenido
3 </h1>
```

Donde los valores de las variable style, size y weight se establecen en base a la lógica del componente. Cuando se trata de un único atributo es preferible utilizar el style binding.

```
1 <div [style.fontSize]="isSpecial ? 'x-large' : 'smaller'">
2     Contenido
3 </div>
```

Cuando el atributo “isSpecial” sea true se aplicará el estilo “x-large”, en caso contrario, “smaller”.

Directivas estructurales

Son el tipo de directivas que alteran el layout pudiendo añadir, eliminar o reemplazar elementos del DOM.

ngIf

Permite mostrar o no un elemento HTML en función del resultado de una condición.

```
1 <div *ngIf="errorCount > 0" class="error">
2     {{errorCount}} errors detected
3 </div>
```

Este sería el típico ejemplo cuando queremos mostrar el número de errores que tiene un formulario pero solo cuando exista algún error.

Date cuenta que no estamos usando los corchetes si no un * en esta directiva, esto es la simplificación de esta sintaxis:

```
1 <ng-template [ngIf]="errorCount > 0">
2   <div class="error">
3     {{errorCount}} errors detected
4   </div>
5 </ng-template>
```

ngIf - then - else

A partir de Angular 4 podemos hacer uso de then y else en un ngIf; donde en función de la condición se va a renderizar un elemento ng-template u otro, los cuales están referenciados gracias a la #, como vemos en el siguiente ejemplo:

```
1 <div *ngIf="errorCount == 0; then ok else error"></div>
2
3 <ng-template #ok>No errors</ng-template>
4
5 <ng-template #error>
6   {{errorCount}} errors
7 </ng-template>
```

ngSwitch

Permite agrupar un conjunto de condiciones y ejecutar solo la primera que se cumpla.

```
1 <div [ngSwitch]="value">
2   <p *ngSwitchCase="'init'">increment to start</p>
3   <p *ngSwitchCase="0">0, increment again</p>
4   <p *ngSwitchCase="1">1, increment again</p>
5   <p *ngSwitchCase="2">2, stop incrementing</p>
6   <p *ngSwitchDefault>< 2, STOP!</p>
7 </div>
```

Con la directiva “ngSwitch” interrogamos el valor, con la directiva “ngSwitchCase” establecemos una condición y con “ngSwitchDefault” establecemos la condición por defecto que solo se ejecutará en caso de que no se cumpla ninguna de las anteriores.

ngFor

Permite repetir un elemento HTML tantas veces como marque la iteración de un conjunto de variables. En esta iteración podemos tener acceso a las siguientes variables locales:

- **index:** devuelve el número de iteración que se esté procesando.
- **last:** devuelve un boolean indicando si es o no el último elemento de la iteración.
- **even:** devuelve boolean indicando si index tiene valor par.
- **odd:** devuelve boolean indicando si index tiene valor impar.

```
1 <ul>
2   <li *ngFor="let item of items; let i = index; let o = odd; let l = last; let\
3   e = even">
4       {{i}} {{item}}
5   </li>
6 </ul>
```

Creación de una directiva de atributo propia

Si estás leyendo en orden el libro te aconsejo que saltes esta sección ya que incluye aspectos que se ven en capítulos siguientes.

Algo muy común y que nos puede pasar es que necesitemos resaltar un texto o cualquier otro elemento al pasar por encima con el ratón, es lo que se conoce como highlight o resaltado.

Esta operación requiere de modificar el DOM por lo que el lugar idóneo para hacer esto es en una directiva sin template, que identificamos con el decorador @Directive.

Como hemos dicho que no tiene template tampoco el usuario podrá interactuar de forma directa con esta directiva es por eso que utilizamos el decorador @HostListener para definir los métodos manejadores en base a una serie de interacciones.

En el caso de querer dejar al usuario de la etiqueta poder definir alguna propiedad de la directiva, hacemos uso del decorador @Input. Podemos poner tantos @Input como necesitemos.

El código quedaría de esta forma:


```
1 import { Directive, HostListener, ElementRef, Renderer2, Input } from '@angular/\n2 core';\n3\n4 @Directive({\n5   selector: '[appHighlight]'\n6 })\n7 export class HighlightDirective {\n8\n9   @Input('appHighlight') appHighlight: string;\n10\n11   @HostListener('mouseenter') onMouseEnter() {\n12     this.highlight(this.appHighlight || 'yellow');\n13   }\n14\n15   @HostListener('mouseleave') onMouseLeave() {\n16     this.highlight(null);\n17   }\n18\n19   constructor(private element: ElementRef, private renderer: Renderer2) {\n20     console.log(this.appHighlight);\n21   }\n22\n23   highlight(value: string | null) {\n24     this.renderer.setStyle(this.element.nativeElement, 'backgroundColor', value);\n25   }\n26\n27 }
```

Es muy importante que la modificación del DOM se realice a través del servicio `renderer` para que no quede acoplado exclusivamente al navegador y otras tecnologías renderizadoras (como `NativeScript`) puedan ajustarse a esta directiva y el proceso de AOT funcione sin problemas.

Estos serían ejemplos de modificaciones erróneas del DOM que darían al traste con la independencia del render y por tanto con el AOT.

```
1 $('<div>.bad-with-jquery').click();\n2 this._elementRef.nativeElement.xyz = 'bad with native element';\n3 document.getElementById('bad-with-document');
```

Ahora en cualquier template de nuestra aplicación podemos aplicar la directiva dando o no valor inicial. Ejemplo

```
1  template: `2  or azul</p>`
```

Pipes

Los pipes se utilizan para hacer transformaciones sobre la forma de visualización de los datos en la vista. Si has trabajado con AngularJS es el equivalente al concepto de “filter”.

Existen dos tipos de pipe en función de si tienen o no estado, los stateless que no tienen estado: aplican la lógica del elemento que corresponda y se olvidan; y los stateful: que si manejan estado. Más o menos el 90% de los pipes son y deben ser de tipo stateless o también conocidos como puros.

Pipes de serie

Antes de plantearte la implementación de un pipe propio conviene conocer los que Angular ya trae de serie y que cubren la mayoría de casos típicos.

- **DatePipe:** `date_expression | date[:format[:timezone[:locale]]]` es utilizado para formatear fechas. Ejemplos: `{{today | date:'medium'}} //Oct 10, 2017, 5:33:09 PM` `{{today | date:'yyyy-MM-dd'}} //2017-10-10` A partir de Angular 5 se puede especificar también el timezone y el locale, ejemplos:

```
1 {{today | date:'yyyy-MM-dd HH:mm a z':'+0900'}} //2017-12-08 08:20 AM GMT+9
2 {{today | date:'medium':'+0900':'fr'}}
```

- **DecimalPipe:** `number_expression | number[:digitInfo[:locale]]` para formatear variables de tipo number estableciendo el número de enteros y número mínimo y máximo de decimales. `{{34.676 | number:'2.0-2'}} //34.68` A partir de Angular 5 también se puede especificar el locale, por ejemplo:

```
{{34.676 | number:'2.0-2':'es'}} //34,68
```

- **CurrencyPipe:** `number_expression | currency[:currencyCode[:display[:digitInfo[:locale]]]]` para pintar correctamente el formato de moneda, en función de tipo de moneda y si se quiere mostrar el símbolo asociado.

```
1 {{34.68 | currency:'EUR':'code'}} //EUR34.68
```

A partir de Angular 5 también se puede especificar el locale, y en vez de establecer el símbolo con un boolean ahora “display” admite los valores:

- **code:** para usar el nombre del código (USD, EUR, ...)
- **symbol:** (por defecto) para usar el símbolo (\$, €, ...)
- **symbol-narrow:** para algunos países como Canada que tienen dos símbolos (CA\$)

```
1 {{34.68 | currency:'EUR':'symbol':'1.2-2':'es'}} //34,68€
```

- **PercentPipe: number_expression | percent[:digitInfo[:locale]]** para pintar el cálculo del tanto por ciento de un número.

```
1 {{34.68 | percent}} //3.468%
```

A partir de Angular 5 también se puede especificar el locale, por ejemplo:

```
1 {{34.68 | percent:'es'}} //3,468%
```

- **UpperCasePipe: string | uppercase** para poner en mayúsculas un texto.

```
1 {{'Texto de prueba' | uppercase}} //TEXTO DE PRUEBA
```

- **LowerCasePipe: string | lowercase** para poner en minúsculas un texto.

```
1 {{'Texto de prueba' | lowercase}} //texto de prueba
```

- **TitleCasePipe: string | titlecase** para poner cada primera letra de palabra en mayúscula.

```
1 {{'Texto de prueba' | titlecase}} //Texto De Prueba
```

- **I18nPluralPipe: expression | i18nPlural:mapping[:locale]** para ajustar el texto en base a la cardinalidad de un valor. Por ejemplo, cuando mostramos la cantidad de resultados y queremos un texto distinto en función del número de resultados devueltos. A partir de Angular 5 también admite especificar un locale.

```
1  -- En la clase de negocio
2  messages: string[];
3  messagesMapping: {[key: string]: string};
4
5  ngOnInit() {
6    this.messages = [];
7    this.messagesMapping = {'=0': 'No hay mensajes', '=1': 'Tienes un mensaje', 'o\
8  ther': 'Tienes # mensajes'};
9  }
```

```
1  -- En el template
2  {{messages.length | i18nPlural:messagesMapping}} //No hay mensajes
```

Nota: la # indica que se va a mostrar el valor de la cardinalidad, es decir, si el array tuviera 3 mensajes, se mostraría por pantalla “Tienes 3 mensajes”.

- **i18nSelectPipe: expression | i18nSelect:mapping** para definir un objeto clave/valor y pintar por pantalla el valor que coincida con la clave. Es útil cuando queremos poner un texto distinto en base a si el usuario es hombre o mujer, por ejemplo.

```
1  -- En la clase de negocio
2  genero: string;
3  generoMapping: {[key: string]: string};
4
5  ngOnInit() {
6    this.genero = 'hombre';
7    this.generoMapping = {'hombre': 'Sr.', 'mujer': 'Sra.'};
8  }
```

```
1  -- En el template
2
3  {{genero | i18nSelect:generoMapping}} //Sr.
```

- **JSONPipe: expression | json** para pintar un objeto en formato json, útil en fase de depuración.

```
1 {{obj | json}}
```

- **SlicePipe: array_or_string_expression | slice:start[:end]** para hacer subcadenas o subarrays.

```
1 {{'Texto de prueba' | slice:0:11}} //Texto de pr
```

- **AsyncPipe: observable_or_promise_expression | async** para suscribirnos a un observable o una promesa y pintar el último valor emitido. Este es el único de los que hemos visto que no es stateless y veremos su utilización en el tema de Http.

Todos ellos se encuentran en el módulo CommonModule de la librería '@angular/common'

Parametrización y encadenado

Son dos de las características de cualquier pipe. Por un lado se pueden incluir parámetros para especializar su funcionamiento y, por otro, se pueden encadenar en un valor tantos pipes como tengan sentido.

En este ejemplo vemos como se parametriza el pipe de fechas con un valor que se establece en el modelo y como a ese resultado le podemos aplicar el pipe de escribir en mayúsculas.

```
1 @Component({
2     selector: 'test-pipe',
3     template: `My birthday is {{birthday | date:format | uppercase}}`
4 })
5
6 export class MyBirthday{
7
8     birthday = new Date(1982, 6, 2) //Jul 2, 1982
9     format: string = 'fullDate'
10
11 }
```

Creación de pipes propios

Si necesitamos de un pipe que no venga de serie podemos implementarlo nosotros. Por ejemplo, si queremos decorar un texto con unos caracteres determinados.

Lo primero que tenemos que hacer es decorar la clase con el decorador @Pipe donde realizamos la definición de las propiedades del pipe:

- **name:** será el nombre que identifique al pipe en el template de un componente.
- **pure:** true (por defecto) si es stateless o false si es stateful.

Ahora tenemos que crear una clase que implemente la interfaz PipeTransform y dentro del método transform ejecutar la lógica que queramos aplicar al valor, teniendo en cuenta que en el primer argumento vamos a recibir el valor al que queremos aplicar la transformación y en el segundo argumento y sucesivos, los parámetros de parametrización; en este caso solo el string que vamos a utilizar para decorar el texto y que será establecido al usar el pipe en el template después de los ‘.’. Esta sería la implementación del pipe:

```
1  import {Pipe, PipeTransform} from '@angular/core';
2
3  @Pipe({
4    name: 'decorator', pure: true
5  })
6
7  export class DecoratorPipe implements PipeTransform {
8    transform(value: string, dec: string):string{
9      return dec + value + dec
10   }
11 }
```

Y este sería un ejemplo de aplicación en un componente:

```
1  @Component({
2    selector: 'app',
3    template: `Texto: {{texto | decorator:'**'}}` /**Angular**
4  })
5
6  export class App implements OnInit {
7
8    texto: string;
9
10   ngOnInit(){
11     this.texto = 'Angular';
12   }
13 }
```

No olvides declarar la clase en la propiedad declarations del módulo correspondiente.

Data Binding

El data binding es el mecanismo que mantiene sincronizado el modelo y la vista. Angular tiene cuatro tipos de data binding: tres de una sola dirección (interpolation, property binding y event binding), y uno de doble dirección con la directiva ngModel.

Interpolation

Es la forma más sencilla de mostrar un valor del modelo en el template del componente para que se refleje en la vista del navegador. Utiliza la sintaxis de la doble llave '{{valor}}'.

De esta forma el framework detecta esta parte del código y lo sustituye por el valor que tiene la variable en el modelo. Son solo de un sentido del modelo hacia la vista.

```
1 @Component({
2   selector: 'interpolation',
3   template: `<p>
4       My favourite number is {{myNumber}}
5     </p>`
6 })
7
8 export class MyFavouriteNumberComponent{
9   myNumber:number = 11;
10 }
```

Property binding

Permite asignar valores a los elementos de la vista. Se caracteriza por embeber el target entre corchetes y este target puede ser de tres tipos distintos.

Element Property

Realmente la interpolación es una forma abreviada de hacer un binding a un element property, en este caso la propiedad innerHTML de un elemento párrafo de HTML. De forma que este ejemplo sería el equivalente al anterior.


```
1 @Component({
2   selector: 'element-property',
3   template: `<p [innerHTML]="My favourite number is " + myNumber"></p>`
4 })
5
6 export class MyFavouriteNumberComponent{
7   myNumber:number = 11;
8 }
```

Component Property

Sirve para transmitir información entre componentes de padre a hijos. En este caso estamos transmitiendo cada uno de los objetos persona a un componente hijo que pinta el detalle.

```
1 @Component({
2   selector: 'component-property',
3   template: `<ul>
4     <li *ngFor="let currentPerson of people">
5       <person-detail [person]="currentPerson"></person-detail>
6     </li>
7 </ul>`
8 })
9
10 export class ListPersonComponent{}
```

De esta forma el componente hijo tiene que tener definido que puede recibir dicha información esto lo hace gracias al decorador @Input, como vemos en la continuación del ejemplo.

```
1 @Component({
2   selector: 'person-detail',
3   template: `<p>{{person | json}}</p>`
4 })
5
6 export class PersonDetailComponent{
7   @Input() person: Person;
8 }
```

Directive property

Otro target de un property binding puede ser una directiva como vemos en el caso de la directiva estructural ngIf, cuando no usamos la simplificación con el *:

```
1 <template [ngIf]="errorCount > 0">
2   <div class="error">
3     {{errorCount}} errors detectados
4   </div>
5 </template>
```

o como hemos visto cuando implementábamos la directiva propia highlight,

```
1 <h1 [appHighLight]=" 'blue' ">Es resalta en azul</h1>
```

o cuando hacemos ngClass o style binding:

```
1 <div [style.fontSize]="isSpecial ? 'x-large' : 'smaller' " >
2   Contenido
3 </div>
```

Event binding

Por el contrario, el event binding define la comunicación desde la vista hacia el modelo. Es una comunicación que responde a alguna acción del usuario, como modificar información de un input o pulsar en algún botón. La sintaxis consiste en meter entre paréntesis el target e igualarlo al método que vaya a manejar el evento.

Este sería un ejemplo cuando queremos manejar el evento onClick de un elemento HTML.

```
1 <button (click)="onSave()">Salvar</button>
```

También podemos manejar eventos propios generados por un componente hijo, como en este caso:

```
1 <person-detail (deleted)="onDelete($event)"></person-detail>
```

El evento “deleted” no existe en el estándar, es el componente person-detail quien lo ha expuesto como un EventEmitter con el decorador @Output.

```
1  @Component({
2    selector: 'person-detail',
3    template: `<p>{{person | json}}</p>`
4  })
5
6  export class PersonDetailComponent implements OnInit {
7
8    @Input() person: Person;
9    @Output() deleted = new EventEmitter();
10
11    ngOnInit() {
12      this.deleted.emit('Persona eliminada.');
```

Ahora en la lógica del padre se implementaría el método `onDelete(event)` que maneja el evento `deleted` capturado.

NgModel

Es la directiva que nos permite definir la comunicación bidireccional, donde un cambio en la vista inmediatamente se ve reflejado en el modelo y viceversa. No hay que abusar de esta característica ya que se ha demostrado que complica el `change detection` y por tanto empeora el rendimiento.

```
1  <input [( ngModel )]="firstName">
```

Para acordarnos de la sintaxis correcta podemos definirla como el “plátano dentro de la caja”. Esto realmente es como hacer un `property binding` y un `event binding` de la forma que vemos en el ejemplo.

```
1  <input [value]="firstName" (input)="firstName=$event.target.value">
```

Para hacer uso de esta directiva tenemos que importar el módulo `FormsModule` obligatoriamente.

Servicios e inyección de dependencias

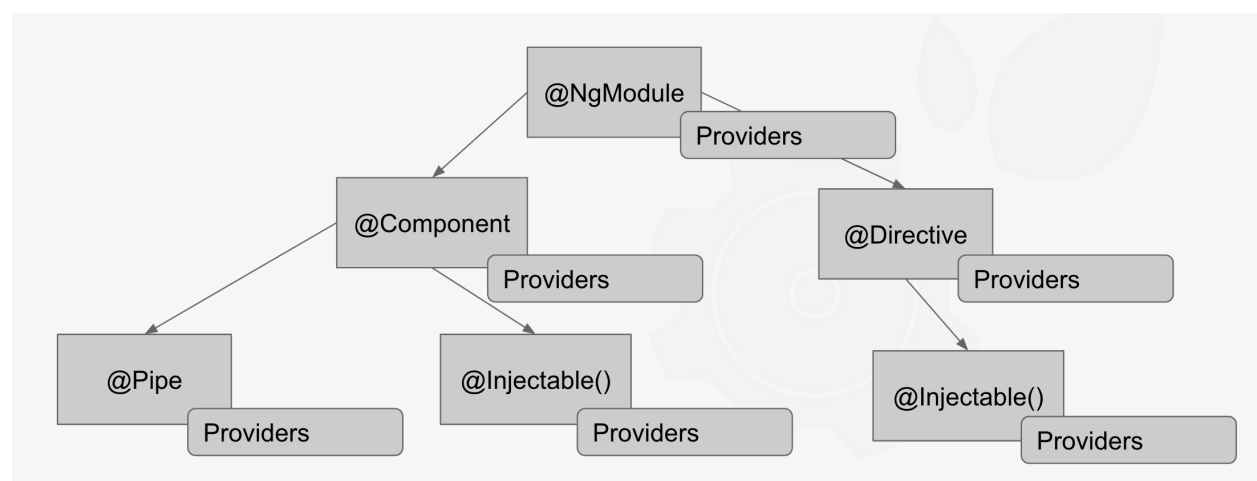
Realmente Angular no maneja estrictamente el concepto de servicio. Servicio puede ser cualquier clase que implementemos que tenga funcionalidad transversal a los componentes, es decir, que queramos que su lógica sea utilizada por varios componentes u otros servicios, estemos utilizando Angular o cualquier otro framework. De hecho es buena práctica crear una capa de servicios con la lógica de negocio de tu aplicación que sea agnóstica al framework utilizado.

Lo realmente importante es que el framework que utilices maneje el concepto de inyección de dependencias e inversión de control, que permite un mayor desacoplamiento entre clases, favoreciendo el testing unitario y de integración, con la ayuda de los dobles de pruebas; y el diseño con TDD.

Elemento Injector

Angular implementa la inyección de dependencias con el elemento injector que crea de forma transparente al desarrollador, y que se encarga de instanciar las dependencias para que puedan ser inyectadas a través del constructor de las clases.

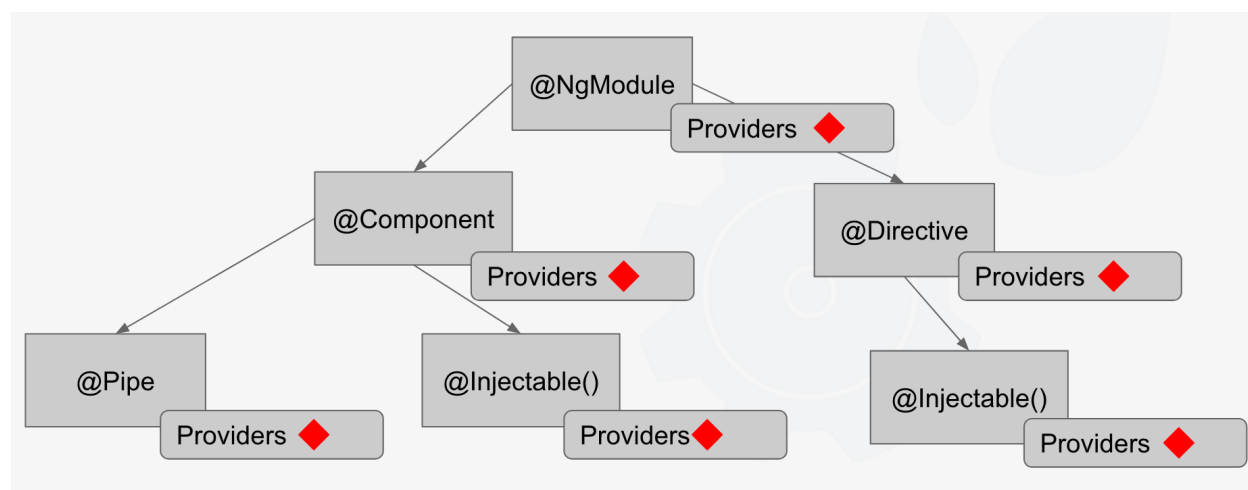
El injector principal se crea automáticamente en el proceso de arranque de la aplicación, y por cada clase decorada con `@Component`, `@Directive`, `@Pipe` o `@Injectable`, se crea un injector secundario que hereda las características de su padre; este injector tiene una doble misión por un lado permite hacer la inyección de dependencias a través del constructor y por otro crea la jerarquía de inyectores para la resolución de dependencias, como vemos en la siguiente imagen:



Nosotros como desarrolladores en lo único que nos tenemos que preocupar es en declarar las dependencias de nuestra aplicación, que en Angular se llaman providers.

Injector principal

En caso de que queramos que nuestro provider sea Singleton, es decir, tenga una única instancia y pueda ser inyectado en cualquier punto de la aplicación tenemos que declararlo en el injector principal de la aplicación.



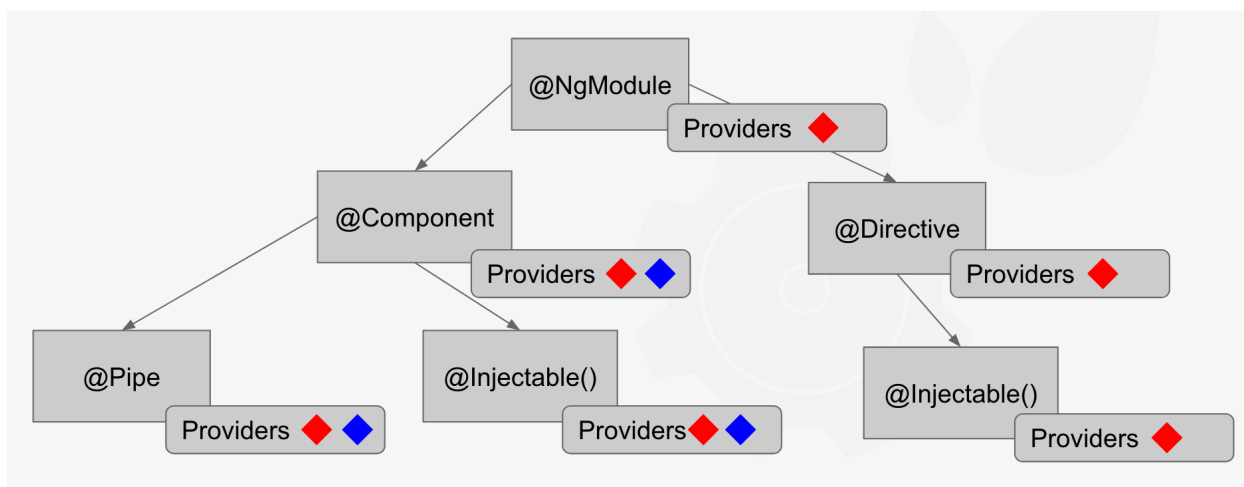
En el gráfico puedes ver como nuestro provider, el rombo rojo, se declara en el injector principal a través de la propiedad `providers` del `@NgModule` e inmediatamente cualquier elemento de la aplicación puede inyectarlo a través de su constructor.

Es importante que tengas en cuenta que cuando se inyecta este provider siempre se hace de la misma instancia por lo que si mantenemos estado en el provider va a ser compartido por todos los elementos que lo instancien, es decir, si algún elemento ha modificado ese estado el siguiente elemento que lo instancia verá esta modificación.

Hay que tener cuidado pues esto puede dar lugar a efectos colaterales y condiciones de carrera muy difíciles de depurar.

Injectores secundarios

En el caso de querer restringir la inyección del provider a una determinada parte de la aplicación, solo podemos declararlo en la propiedad `providers` de `@Component` o `@Directive`.

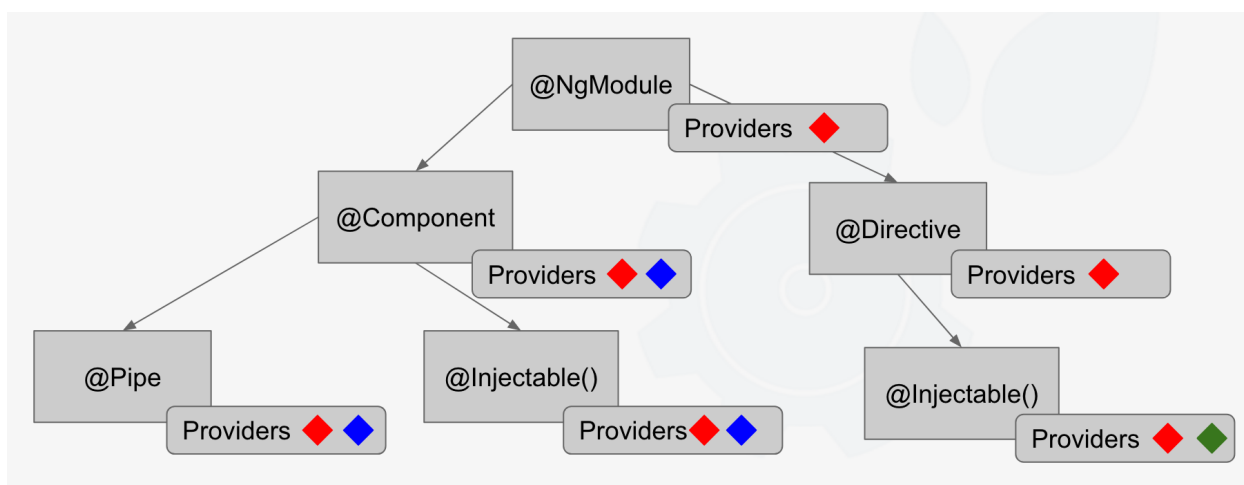


En el gráfico se muestra cómo hacemos la declaración de un nuevo provider, el rombo azul, en la propiedad providers de `@Component` y ahora solo los elementos que cuelgan de este componente pueden hacer uso del provider.

Estos providers ya no serán Singleton, es decir, el elemento que haga la inyección de estos providers recibirá una instancia nueva, con lo que no compartirán el estado evitando efectos colaterales y condiciones de carrera.

Localización de dependencias

En el caso en el que un elemento quiera inyectar un provider que no está declarado en ninguna propiedad providers, Angular nos devolverá un error del tipo “Not found provider”.



En el gráfico vemos como un servicio a través de `@Injectable` quiere inyectar otro servicio, el rombo verde, el proceso de localización de dependencia, buscará ese provider

en el inyector más cercano en este caso `@Directive`, al no encontrarlo, buscará en el siguiente elemento de la jerarquía que es el módulo principal, al no encontrarlo en el inyector principal nos dará el error mencionado de `Not found provider`.

Ten en cuenta que hasta que un servicio no se declara como `provider` no puede ser inyectado por otro elemento.

Para declarar los `providers` ya sea en el inyector principal o en los secundarios contamos con una serie de recetas.

Declaración de providers

Un `provider` es cada uno de los elementos que pueden ser inyectados en el elemento inyector. Para poder incluir `providers` en el inyector principal tenemos que hacerlo a través de propiedad “`providers`” de la definición del módulo principal de la aplicación o del `@Component/@Directive`, como se explicó anteriormente.

Para la definición del `provider` utilizamos un objeto donde indicamos dos propiedades, por un lado el token de acceso y por otro la receta para crear la instancia.

useClass

La forma más común es la declaración de un servicio como `provider`, donde simplemente incluimos en el array de la propiedad `providers` el nombre de la clase, pero no como `string`.

```
1 providers: [  
2     HelloService  
3 ]
```

Realmente esto es la simplificación de la siguiente declaración donde incluimos un objeto con las propiedades: `provide` y `useClass`, indicando en `provide` la instancia del servicio y en `useClass` que servicio la va a implementar. Cuando son iguales se puede hacer la simplificación mencionada.

```
1 providers: [  
2     {provide: HelloService, useClass: HelloserviceFake}  
3 ]
```

Esta sintaxis extendida es útil cuando la clase a implementar es distinta, como en los casos en lo que queremos cambiar la implementación, para por ejemplo, crear un `fake`.

Para poder hacer uso de la instancia de la clase utilizamos el decorador `@Inject` en el constructor de cualquier elemento, que tenga acceso, indicando el token de localización, es decir, el nombre de la clase.

```
1 constructor(@Inject(HelloService) private helloService:HelloService)
```

A partir de Angular 5 se modifica el comportamiento del inyector pasando a ser estático; de forma que si un servicio depende de otros somos nosotros los que tenemos que establecerlo en la declaración del provider con la propiedad `deps`, como vemos en el ejemplo:

```
1 providers: [  
2   {  
3     provide: CalculadoraService,  
4     useClass: CalculadoraService,  
5     deps: [{provide:SumarService, useClass: SumarServiceFake}]  
6   }  
7 ]
```

Esto es que `HelloService` necesita una instancia de `SaludoService`, es decir:

```
1 export class CalculadoraService {  
2   constructor(private sumarService: SumarService){}  
3 }
```

Antes de Angular 5 esta dependencia la resolvía el inyector con reflexión y ahora para evitar problemas conocidos de implementación, se tiene que especificar como se ha explicado con la propiedad `“deps”`.

useValue

Cuando lo que queremos es poder compartir un determinado valor u objeto de forma estática, hacemos uso de la receta `useValue`. En este caso en la propiedad `provide` establecemos un string que será utilizado como token para la localización del provider y en `useValue` establecemos el valor que queramos.

En el siguiente ejemplo vemos la típica situación en la que queremos compartir con toda la aplicación los datos de configuración general.


```
1  -- app.module.ts
2
3  const config = {
4    apiEndPoint: 'api.com'
5  }
6
7  providers: [
8    {provide: 'App.config', useValue: config}
9  ]
```

Ahora desde el constructor de cualquier elemento podemos hacer uso del decorador `@Inject` indicando el token de localización para establecer el valor en el identificador y poder hacer uso en la lógica del elemento.

```
1  constructor(@Inject('App.config') private config){}
```

useFactory

En el caso en que para establecer un valor como provider dependamos de la resolución de cierta lógica, contamos con la receta `useFactory` donde el `provide` es un token de tipo string, en `deps` definimos el servicio que va a resolver la lógica y en `useFactory` llamamos al método que devuelve el valor que queremos establecer.

En el siguiente ejemplo vemos como establecer el valor de la constante `LOCALE_ID` que existe en Angular con la resolución de un método que devuelve el valor con el lenguaje a establecer.

```
1  providers: [{provide: LOCALE_ID, deps: [SettingService], useFactory: (settingSer\
2  vice: SettingService) => settingService.getLanguage()}]
```

La forma de inyección es igualmente con el decorador `@Inject` indicando el string que hace de token de localización.

```
1  constructor(@Inject(LOCALE_ID) private LOCALE_ID){}
```

Simplificación en la inyección

Como ya se ha dicho la inyección de dependencias se puede hacer con `@Inject` en todos los casos; pero existe una simplificación solo para el caso de la inyección de servicios que es como se muestra en el primer ejemplo; donde directamente se define el identificador y la clase.

```
1 export class ExamplePipe {  
2     constructor(private otherService:OtherService){}  
3 }
```

Como se ve en el segundo ejemplo, esta forma es mucho menos verbosa y es la que se prefiere, pero tienes que tener en cuenta que solo va a funcionar cuando la opción **“emitDecoratorMetadata”** esté a **true** en la configuración del fichero tsconfig.json.

```
1 export class Service {  
2     constructor(@Inject(OtherService) private  
3         otherService:OtherService){}  
4 }
```

Routing

El router es el elemento que se encarga de definir todas las potenciales vistas de la aplicación permitiendo cambiar la vista principal de la aplicación en base a las interacciones del usuario, simulando de esta forma la navegación entre páginas dentro de una aplicación SPA. Además es el elemento que permite el paso de información entre las distintas vistas donde cada una tiene que tener un componente asociado.

Configuración del router en el módulo principal

Debemos configurar adecuadamente el router para poder hacer uso de él en nuestras aplicaciones. Lo normal es que el módulo principal solo tenga una única entrada que redirija a otro módulo secundario o de feature. Esta sería la forma habitual de configuración del router en el módulo principal:

```
1  import { Routes, RouterModule } from '@angular/router';
2  import { BrowserModule } from '@angular/platform-browser';
3
4  const ROUTES: Routes = [
5    { path: '', redirectTo: 'tutorials', pathMatch: 'full' },
6    { path: '**', redirectTo: 'tutorials' }
7  ];
8
9  @NgModule({
10
11    imports: [
12      BrowserModule,
13      TutorialModule,
14      RouterModule.forRoot(ROUTES)
15    ]
16
17  })
18
19  export class AppModule {}
```

Como ves definimos una constante que almacenará todas las posibles vistas del módulo principal, como ya se ha dicho, lo normal es que redirija a otro módulo que será el que cargaremos por defecto. Esta ruta consta de:

- **path:** que es la URL que va a manejar. En este caso definimos que manejamos la URL cuando no viene ninguna otra vista definida.
- **redirectTo:** nos permite redireccionar a otro path que tiene que estar definido en algún otro módulo relacionado.
- **pathMatch:** con valor “full” podemos indicarle que solo haga el redirect cuando el path sea exactamente igual, en este caso, a cadena vacía, esto se hace para la primera carga que siempre tendrá la cadena vacía.

También podemos definir que ante cualquier ruta desconocida (**) se haga el redirect a una página 404 específica, o como en el ejemplo que se vaya al módulo de entrada.

Ahora utilizamos el método `forRoot()` de la clase `RouterModule` para definirle a Angular cuáles son las rutas válidas de nuestro módulo principal y junto a este import, importamos también el módulo secundario al que estamos haciendo referencia.

Hay que tener mucho cuidado con no cargar muchas vistas en el módulo principal ni el secundario que carga primero, ya que todas las clases asociadas se cargarán en un único bundle el cual será muy grande y puede retrasar la carga inicial de la aplicación.

Configuración del router en módulo secundario o de feature

Podemos definir las vistas propias de un módulo secundario o de feature de forma independiente. Para ello definimos también una constante con todas las posibles rutas de este módulo y sus vistas asociadas. Vamos a ver un ejemplo:

```
1 import { Routes, RouterModule } from '@angular/router';
2 import { CommonModule } from '@angular/common';
3
4 const ROUTES: Routes = [
5   { path: 'tutorials', children: [
6     { path: '', component: TutorialesComponent }
7     { path: ':id', component: TutorialesDetailComponent }
8   ]}];
9
10 @NgModule({
11
12   imports: [CommonModule, RouterModule.forChild(ROUTES)]
13
14 })
```

```
15  
16 export class TutorialesModule {}
```

En este ejemplo definimos las posibles vistas para el módulo “tutorials” que va a tener una vista principal con una lista de tutoriales y otra vista con el detalle de cada uno de ellos.

Para poder hacer esto contamos con los siguientes elementos:

- **path:** como ya se ha dicho representa la URL que se va a manejar. En este caso estamos definiendo que este módulo va a manejar los paths que contengan “tutorials”. Fíjate como también se utiliza para definir las vistas internas del módulo, en este caso, vacío que va a mostrar el componente con la lista de tutoriales, sería “tutorials/” y para mostrar el detalle de cada uno ellos, definimos un nuevo path que admite el paso de un identificador, de forma que manejará las rutas de la forma “tutorials/1” o “tutorials/2”, etc... y mostrará la vista y la lógica del componente de detalle.
- **children:** es una array que define las distintas vistas internas del módulo.
- **component:** lo utilizamos para determinar el componente a mostrar en función del path asociado.

Para decirle a Angular cuáles son las rutas de nuestro módulo de feature que tiene que tener en cuenta, hacemos uso de la función de `forChild()` del `RouterModule` pasándole por parámetro las rutas definidas.

Configuración con Lazy Loading

En caso de querer aprovechar las ventajas de lazy loading para modularizar nuestra aplicación en distintos bundles pequeños que ir cargando bajo demanda tenemos que hacer pequeños ajustes en la configuración de los módulos que conocemos hasta ahora.

Para el módulo secundario o de feature

En este caso tenemos que tener en cuenta que el módulo tiene que ser independiente por lo que las rutas siempre van a empezar por cadena vacía, que ya hemos dicho que es la ruta por defecto. Entonces el caso anterior quedaría de esta forma:

```
1 import { Routes, RouterModule } from '@angular/router';
2 import { CommonModule } from '@angular/common';
3
4 const ROUTES: Routes = [
5   { path: '', component: TutorialComponent },
6   { path: ':id', component: TutorialDetailComponent }
7 ];
8
9 @NgModule({imports: [CommonModule, RouterModule.forChild(ROUTES)]})
10 export class TutorialModule {}
```

Para el módulo principal

Ahora en el módulo principal tenemos que eliminar la importación del módulo secundario para aliviar este peso del bundle principal, pero tenemos que indicar que queremos seguir cargando TutorialModule.

La forma de hacer esto es con la propiedad loadChildren donde, en forma de cadena, indicamos la ruta relativa al módulo que queremos cargar.

```
1 import { Routes, RouterModule } from '@angular/router';
2 import { BrowserModule } from '@angular/platform-browser';
3
4 const ROUTES: Routes = [
5   { path: 'tutorials', loadChildren: './tutorials.module#TutorialModule' },
6 ];
7
8 @NgModule({imports: [BrowserModule, RouterModule.forRoot(ROUTES)]})
9 export class AppModule {}
```

De este modo cuando un usuario haga uso de la aplicación, primero se cargará el bundle principal de forma muy rápida y acto seguido un nuevo fichero con el contenido del módulo de tutoriales.

Router outlet y router links

El router outlet es la directiva que Angular utiliza para identificar áreas de renderizado en la aplicación, mientras que, el router link es la directiva que hay que utilizar para que Angular sepa montar los enlaces a las distintas rutas.

Lo más habitual en una aplicación es que el componente principal, es decir, el que se define dentro de la propiedad bootstrap del módulo principal, tenga un template asociado que presente un aspecto similar a este:

```
1 template: `  
2   <h1>Aplicación principal</h1>  
3   <nav>  
4     <a [routerLink]="['/tutoriales']">Tutoriales</a>  
5   </nav>  
6  
7  
8   <router-outlet></router-outlet>  
9 `,
```

En pantalla veremos algo como esto:

Aplicación principal

[Tutoriales](#)

Lista de tutoriales

Donde el texto “Lista de tutoriales” es renderizado por el componente “TutorialComponent” dentro del área definida por la directiva router-outlet cuando pinchamos en el enlace “Tutoriales”. Si tuviéramos otros router links configurados podríamos ver que al pinchar sobre ellos el área de renderizado cambia cargando el contenido y la lógica del componente asociado.

Navegación desde código

Los router links no son la única forma de poder cargar una u otra ruta, esto también lo podemos hacer desde la lógica de cualquier componente, como vemos a continuación:

```
1 import { Router } from '@angular/router';
2
3 export class TutorialComponent {
4
5     constructor(private router: Router) { }
6
7     onSelect(tutorial: Tutorial) {
8         this.router.navigate( ['tutoriales', tutorial.id]);
9     }
10 }
```

Esta lógica estaría dentro del componente “TutorialComponent” y lo que permite es que cuando el usuario pulse sobre un elemento de la lista, se dispare el evento `onSelect` con la información del tutorial que ha pulsado para transmitirla mediante la función “navigate” del router a la vista del detalle que como admite el paso de un id, se lo establecemos como segundo elemento del array, de forma que cuando se produzca este evento se renderice en el router-outlet el componente de detalle con la información que se obtenga a partir de su id.

Recuperación de la información

La información, ya sea transmitida a través de la directiva `router-link` o a través de la función `navigate` del servicio Router, puede ser recuperada gracias al servicio “ActivateRoute”, como vemos en el ejemplo:

```
1 import { ActivatedRoute } from '@angular/router';
2
3 export class TutorialesDetailComponent implements OnInit {
4
5     constructor(private activatedRoute: ActivatedRoute) { }
6
7     ngOnInit() {
8         const id = this.activatedRoute.snapshot.params['id'];
9     }
10
11 }
```

En este caso vemos como dentro del objeto `snapshot` se almacena un array con los parámetros de la URL, los cuales podemos recuperar a través del nombre de la variable. Ahora con el id podemos llamar a algún servicio que nos devuelva la información con el detalle de un tutorial.

Query params

Nos sirve para definir parámetros que son opcionales, es decir, que no van a formar parte del path si no que se van a añadir a partir del símbolo “?” en la URL. Estos parámetros pueden transmitirse a través de la directiva router-link de esta forma:

```
1 @Component({
2   template: `
3     <a [routerLink]="['/state']" [queryParams]="{param:1}">State</a>`,
4 })
```

O también a través de la función navigate del servicio de router de esta otra forma:

```
1 ngOnInit() {
2   this.router.navigate(['/state'], {queryParams: {param:1}});
3 }
```

Para recuperarlos hacemos uso del servicio ActivatedRoute para acceder a ellos a través de identificador en el objeto snapshot, pero en este caso se encuentran en el objeto queryParams.

```
1 ngOnInit() {
2   this.param = this.actRoute.snapshot.queryParams['param'];
3   this.router.navigate(['comp2'], {queryParams: {param:
4     this.param + 1}})
5 }
```

Tanto en el caso de params como de queryParams al hacer uso de snapshot, si el componente sigue renderizado en la pantalla, por mucho que pinche en otros enlaces para mostrar información distinta, no me va a hacer caso, ya que el método ngOnInit solo se ejecuta una vez en la carga del componente.

Para evitar esto, Angular recurre a la programación reactiva permitiendo subscribirnos tanto params,

```
1 ngOnInit() {  
2     this.activateRoute.params.subscribe(  
3         params => {  
4             const id = params['id'];  
5         }  
6     );  
7 }
```

como a queryParams:

```
1 ngOnInit() {  
2     this.activateRoute.queryParams.subscribe(  
3         queryParams => {  
4             this.param = queryParams['param'];  
5         }  
6     );  
7 }
```

Rutas de guarda

Otra de las características del router es que actúa de guardián de las rutas analizando si tenemos permiso para ver el componente que se corresponde con la URL (activar la ruta) o si tenemos permisos para salir del componente hacia otra ruta (desactivar la ruta).

Estas dos características se implementan mediante las interfaces CanActivate y CanDeactivate.

CanActivate

Así si queremos representar una condición de entrada en los componentes, por ejemplo que el usuario cumpla con un determinado rol o que tenga presente un token, podemos implementar un servicio de esta forma:

```
1 import { Injectable } from '@angular/core';
2 import { ActivatedRouteSnapshot, RouterStateSnapshot,
3       Router, CanActivate } from '@angular/router';
4
5 @Injectable()
6 export class AuthService implements CanActivate {
7
8   constructor(private router: Router) {}
9
10  canActivate(route: ActivatedRouteSnapshot,
11    state: RouterStateSnapshot): boolean {
12
13    const token = localStorage.getItem('token');
14
15    if (isValidToken(token)) {
16      return true;
17    } else {
18      this.router.navigateByUrl('/login', {queryParams: {returnUrl: state.url}});
19      return false;
20    }
21  }
22 }
```

Esta es una lógica muy común en cualquier aplicación, queremos que si el usuario no cumple una determinada condición, en este caso que tenga un token válido, le redireccionamos a la página de login pasando la URL de retorno. Si el usuario tiene token válido entonces le dejamos visualizar el componente asociado a la URL que estamos protegiendo.

Fíjate como el método recibe dos argumentos: el primero, un snapshot del router que nos permite obtener la URL actual, y el segundo, un snapshot de la URL por si queremos acceder a alguna variable de esta URL para tomar la decisión. Para poder hacer la redirección hacemos uso del servicio Router como vimos anteriormente.

Para salvaguardar una ruta con este servicio simplemente tenemos que indicarlo en la configuración de la ruta, de esta forma:

```
1 const ROUTES: Routes = [
2
3   {path: 'ruta-prottegida', component: SecretoComponent, canActivate [AuthService]}
4
5 ];
```

CanDeactivate

El otro caso es que queramos establecer una condición para abandonar cierta URL. Por ejemplo, imaginad la típica pantalla de configuración donde todavía no le hemos dado a guardar y pulsamos en otro enlace, de forma que la aplicación no sale inmediatamente a la otra URL sino que nos avisa con un mensaje de confirmación de si queremos salir sin guardar los cambios.

Para este caso implementamos la interfaz CanDeactivate de este modo, notad que la lógica se aplica a un componente en concreto.

```
1 import { CanDeactivate, ActivatedRouteSnapshot,
2         RouterStateSnapshot } from '@angular/router';
3 import { Injectable } from '@angular/core';
4 import { ExampleComponent } from '../example/example.component';
5
6 @Injectable()
7 export class ConfirmService implements CanDeactivate<ExampleComponent>{
8
9     constructor() { }
10
11     canDeactivate(component: Comp1Component, currentRoute:
12     ActivatedRouteSnapshot, currentState: RouterStateSnapshot,
13     nextState?: RouterStateSnapshot): boolean {
14         return confirm('¿Estás seguro de querer salir?');
15     }
16 }
```

En este caso como lógica implementamos un sencillo confirm que lo único que hace es preguntar siempre antes de salir del componente. Si el usuario acepta, devolverá true y saldrá del componente; mientras que, si el usuario cancela, devolverá false y se quedará dentro del componente.

Para aplicar esta guarda tenemos que utilizar la propiedad canDeactivate de la definición de rutas del router. De esta forma:

```
1 const ROUTES: Routes = [
2     {path: 'comp1', component: Comp1Component,
3     canDeactivate: [ConfirmService]
4 }];
```

Seguro que ahora te estás preguntando, pero si ahora tengo otro componente que implementa la misma lógica, ¿tengo que hacer copy&paste?

¡Eso nunca! Antes de recurrir a tan mala práctica te voy a explicar como resolver esta situación.

Lo primero que vamos a hacer es desacoplar el componente del servicio ConfirmService, para ello lo vamos a implementar de esta forma:

```
1  import { Injectable } from '@angular/core';
2  import { CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
3  import { Observable } from 'rxjs/Observable';
4
5
6  export interface CanComponentDeactivate {
7    canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
8  }
9
10 @Injectable()
11 export class ConfirmService implements CanDeactivate<CanComponentDeactivate> {
12
13   constructor() { }
14
15   canDeactivate(component: CanComponentDeactivate,
16     currentRoute: ActivatedRouteSnapshot,
17     currentState: RouterStateSnapshot,
18     nextState?: RouterStateSnapshot) {
19     return component.canDeactivate ? component.canDeactivate() : true;
20   }
21 }
22 }
```

Como ves hemos creado una interfaz del tipo que admite la clase CanDeactivate y en vez de con el componente, la parametrizamos con la interfaz. Ahora en el método canDeactivate tampoco trabajamos directamente con el componente sino con la interfaz, y si existe el método canDeactivate en el componente que tiene que implementar dicha interfaz, entonces se ejecuta el método canDeactivate del componente y sino se devuelve true, con lo que se dejaría salir de la ruta.

Ahora el componente que quiera verificar su estado antes de salir de la ruta, tiene que implementar la interfaz CanComponentDeactivate que le obliga a implementar el método canDeactivate con la lógica que requiera. Este podría ser un ejemplo:

```
1 import { Component, Input, OnInit, Output, EventEmitter } from '@angular/core';
2 import { CanComponentDeactivate } from 'app/confirm.service';
3
4 @Component({
5   selector: 'app-data-binding-child-example',
6   templateUrl: './data-binding-child-example.component.html',
7   styleUrls: ['./data-binding-child-example.component.css']
8 })
9 export class DataBindingChildExampleComponent implements OnInit, CanComponentDea\
10 ctivate {
11
12   saved: boolean;
13
14   constructor() { }
15
16   ngOnInit() {
17     this.saved = false;
18   }
19
20   save() {
21     this.saved = true;
22   }
23
24   canDeactivate(): boolean {
25     if (saved) {
26       return true;
27     } else {
28       return confirm('Seguro que quieres salir sin guardar');
29     }
30   }
31 }
```

Resolución de información en el router

En ciertos casos nos interesa hacer la resolución de cierta información antes de llegar a renderizar el componente.

Para permitir esto el router cuenta con la propiedad “resolve” seguido de un objeto con el nombre de una propiedad a la que le acompaña un servicio (que se tiene que instanciar como provider y tiene que implementar la interfaz Resolve) que es el encargado de hacer la resolución de la información.

La definición dentro del estado del router podría ser esta:

```
1 {
2   path: 'contact/:id',
3   component: ContactsDetailComponent,
4   resolve: {
5     contact: ContactResolve
6   }
7 }
```

Un ejemplo de implementación de este servicio puede ser:

```
1 import { Injectable } from '@angular/core';
2 import { Resolve, ActivatedRouteSnapshot } from '@angular/router';
3 import { ContactsService } from './contacts.service';
4
5 @Injectable()
6 export class ContactResolve implements Resolve<Contact> {
7
8   constructor(private contactsService: ContactsService) {}
9
10  resolve(route: ActivatedRouteSnapshot) {
11    return this.contactsService.getContact(route.paramMap.get('id'));
12  }
13 }
```

De esta forma en el componente asociado podemos recuperar esta información generada a través del objeto data:

```
1 @Component()
2 export class ContactsDetailComponent implements OnInit {
3
4   contact: Contact
5
6   constructor(private route: ActivatedRoute) {}
7
8   ngOnInit() {
9     this.contact = this.route.snapshot.data['contact'];
10  }
11 }
```

Acceso a la información del router del padre

Si estamos en un escenario como éste donde tenemos una ruta padre con varios hijos a la que se llega con las siguientes URLs:

```
1 /admin/courses/1234/metadata
2 /admin/courses/1234/curriculum
3 /admin/courses/1234/prices
4 /admin/courses/1234/coupons
```

Podemos representarlo con la siguiente estructura de router:

```
1 export const ROUTES: Routes = [
2   {
3     path: '',
4     canActivate: [CoursesGuard],
5     component: CoursesComponent,
6   },
7   {
8     path: ':id',
9     canActivate: [CoursesGuard],
10    component: CourseComponent,
11    children: [
12      { path: '', redirectTo: 'metadata', pathMatch: 'full' },
13      { path: 'metadata', component: CourseMetadataComponent },
14      { path: 'curriculum', component: CourseCurriculumComponent },
15      { path: 'prices', component: CoursePricesComponent },
16      { path: 'coupons', component: CourseCouponsComponent },
17    ],
18  },
19 ];
```

Como ves tenemos un padre que representa el curso con un montón de hijos que dependen del ID que les proporciona el padre.

Entonces desde cada uno de estos hijos podemos acceder a la información de la ruta del padre de esta forma:

```
1 this.id = this.activatedRoute.parent.snapshot.params['id'];
```

Los 7 pasos del proceso de routing

Cada vez que un usuario pincha en alguno de los enlaces gestionados por el router, Angular se tiene que asegurar que la aplicación reacciona correctamente.

Para ello Angular sigue estos 7 pasos en este orden: (PRIGRAM)

- **Parse:** parsea la URL del navegador a la que el usuario quiere navegar
- **Redirect:** aplica el redirect si es que está definido en el router.
- **Identify:** identifica que estado del router se identifica con la URL.
- **Guard:** ejecuta las guardas que tenga definidas ese estado del router.
- **Resolve:** resuelve el data requerido si es que lo hubiera para ese estado del router.
- **Activate:** activa el componente para mostrar el contenido.
- **Manage:** maneja la navegación y repite el proceso cuando se pincha sobre un nuevo enlace.

Formularios

Los formularios representan el punto de entrada de información del usuario y el dolor de cabeza de los desarrolladores.

La facilidad de creación y gestión de los formularios con un buen chequeo de errores y personalización de validaciones es uno de los muchos puntos fuertes de Angular, que presenta dos aproximaciones: model driven y template driven.

Ambas aproximaciones tienen sus pros y sus contras, lo mejor del model driven es que dejamos más limpio el HTML y la potencia de las validaciones síncronas y asíncronas; por contra es la aproximación menos intuitiva.

Lo peor del template driven es el uso que hay que hacer de la directiva `[(ngModel)]` en todos los elementos del formulario que debido a su bidireccionalidad penaliza el rendimiento en función del número de elementos que tenga el formulario.

Es por eso que en este libro se va a ver en detalle la aproximación model driven por ser la más potente al hacer uso de la programación reactiva y para que os resulte más intuitiva.

Configuración inicial

Para poder hacer uso de las directivas y providers específicos de los formularios en la aproximación model driven necesitamos importar el módulo `ReactiveFormsModule` de la librería forms de angular, como vemos a continuación:

```
1 import {NgModule} from '@angular/core';
2 import {ReactiveFormsModule} from '@angular/forms';
3
4 @NgModule({
5   imports: [ReactiveFormsModule]
6 })
```

Creación del formulario

En cualquier componente donde queramos tener un formulario para que el usuario pueda introducir información podemos editar el HTML de esta forma, por ejemplo, para incluir el nombre de usuario y la contraseña de acceso:

```
1  template: `
2      <form>
3          <div>Username: <input type="text"/></div>
4          <div>Password: <input type="password"/></div>
5          <div><button>Send</button></div>
6      </form>`
```

Como podemos ver es un formulario HTML donde no se le indica el action ni nada en especial.

Configuración de los elementos del formulario

Ahora en el componente asociado al template donde tengamos el formulario vamos a crear un atributo del tipo **FormGroup** que va a representar a todo el formulario y en el método `ngOnInit` lo vamos a inicializar con la definición de cada campo, donde cada campo va a ser un **FormControl**, como vemos a continuación:

```
1  form: FormGroup;
2
3  constructor() { }
4
5  ngOnInit() {
6      this.form = new FormGroup({
7          username: new FormControl(''),
8          password: new FormControl(''),
9          address: new FormGroup({
10             city: new FormControl(''),
11             postcode: new FormControl('')
12         })
13     });
14 }
```

Es posible definir subgrupos dentro de un formulario de tipo `FormGroup`, como se representa con el caso de la propiedad “address”

Ahora para asociar cada elemento de la configuración con su correspondiente elemento en el formulario, editamos el template para hacer uso de las directivas **formGroup**, **formGroupName** y **formControlName**, quedando el formulario de esta forma:

```
1  template: `
2      <form [formGroup]="form">
3          Username: <input type="text" formControlName="username"/>
4          Password: <input type="password" formControlName="password"/>
5          <fieldset formGroupName="address">
6              Username: <input type="text" formControlName="username"/>
7              Password: <input type="password" formControlName="password"/>
8          </fieldset>
9          <button>Send</button>
10 </form>`
```

Validaciones síncronas

Como segundo argumento del FormControl se puede pasar un array con el conjunto de validadores síncronos que queremos que nuestro campo supere para considerarlo válido. Para ello se hace uso de la función compose de la clase Validators de la librería forms de Angular o directamente dentro de [].

Angular ya cuenta con los siguientes validadores síncronos de serie:

- **required:** se pone cuando queremos que el campo sea obligatorio.
- **min:** valida que el valor esté por encima de un valor mínimo.
- **max:** valida que el valor esté por debajo de un valor máximo.
- **minLength:** se establece un número mínimo de caracteres para el campo.
- **maxLength:** se establece un número máximo de caracteres para el campo.
- **email:** valida que el campo cumpla con el formato de email.
- **pattern:** se establece que el campo tiene que cumplir con un determinado patrón.

En el ejemplo si queremos que el campo username y password sean obligatorios y que el username tenga un número de caracteres mínimo de 2, tendríamos que añadir la siguiente configuración:

```
1  this.form = new FormGroup({
2      username: new FormControl('',
3          Validators.compose([Validators.required, Validators.minLength(2)])),
4      password: new FormControl('', Validators.compose(
5          [Validators.required]))
6  });
```

También podemos crear nuestras propias validaciones síncronas. Para ello creamos una clase, por ejemplo, `CommonValidator`, que va contener métodos estáticos con los distintos validadores que reciben el `FormControl` al que se asocia.

En este ejemplo vemos como validar que el campo `username` no empiece por un carácter numérico.

```
1  import { FormControl } from '@angular/forms';
2
3  export class CommonValidator {
4
5      static startWithNumber(control: FormControl) {
6
7          let firstChar = control.value.charAt(0);
8          if (firstChar && !isNaN(firstChar)){
9              return {'startWithNumber': true};
10         } else {
11             return null;
12         }
13     }
14 }
15 }
```

Fíjate que cuando la validación no se cumple lo que devolvemos es un objeto con el identificador de la validación a `true` y cuando se cumple devolvemos un `null`. Esto es porque JavaScript a la hora de interpretar el resultado solo lo va a entender como `false` si es `null`.

Cosas que tiene este lenguaje. Cuando hablemos de Jasmine veremos los conceptos de `Truthy` y `Falsy`.

Ahora para asignar este método de validación al campo “username” tenemos que añadirlo dentro del array de validadores de asíncronos quedando de esta forma:

```
1  this.form = new FormGroup({
2    username: new FormControl('',
3      Validators.compose([Validators.required,
4        Validators.minLength(2),
5        CommonValidator.startsWithNumber])),
6    password: new FormControl('',
7      Validators.compose([Validators.required]))
8  });
```

Validaciones asíncronas

Las validaciones asíncronas son aquellas que requieren de un proceso asíncrono para poder validarse. Por ejemplo, cuando necesitamos validar el dato contra una fuente de datos externa. También se tratan de métodos estáticos pero con la particularidad de que siempre tienen que devolver una promesa, como vemos en el ejemplo, donde consultamos a un servicio asíncrono si el “username” existe ya o no.

```
1  static userTaken(service: UserService) {
2    return(control: FormControl) => {
3      return new Promise((resolve) => {
4        service.checkUser(control.value).subscribe(
5          (response) => {
6            resolve(null);
7          },
8          (error) => {
9            resolve({ 'userTaken': true });
10         }
11       );
12     });
13   };
14 }
```

En el ejemplo vemos como el método tiene que recibir una instancia del servicio que va a utilizar e internamente recibe el control asociado y devuelve una promesa donde si la petición es errónea es que el usuario existe y por tanto resolvemos la promesa devolviendo un objeto con el identificador de la validación a true, y si tiene éxito quiere decir que el usuario no existe y devolvemos la promesa con un null.

Para poder asociar esta validación al control que se quiera tenemos que incluirla con la función `composeAsync` de la clase `Validators` como tercer parámetro del `FormControl` o simplemente como array con [], el ejemplo quedaría de esta forma:

```
1 constructor(private service: UserService) { }
2
3 this.form = new FormGroup({
4   username: new FormControl('',
5     Validators.compose([Validators.required,
6       Validators.minLength(2),
7       CommonValidator.startWithNumber
8     ]),
9     Validators.composeAsync([
10      CommonValidator.userTaken(this.service)
11    ])),
12   password: new FormControl('',
13     Validators.compose([
14       Validators.required
15     ]))
16 });
```

Configuración de cuando disparar las validaciones

Por defecto, las validaciones se disparan en el evento change del elemento que se ha modificado, esto puede llevar a mucha sobrecarga de la aplicación, por lo que a partir de Angular 5, se establece un nuevo parámetro “updateOn” que permite establecer los valores ‘blur’ si se quiere realizar la validación cuando se pierda el foco de cada elemento, o ‘submit’ en caso de que solo se quiera hacer cuando se pulse en el botón de submit. Este sería un ejemplo:

```
1 this.form = new FormGroup({
2   username: new FormControl('', {updateOn: 'blur',
3     Validators.compose([Validators.required,
4       Validators.minLength(2),
5       CommonValidator.startWithNumber])}),
6   password: new FormControl('',
7     Validators.compose([Validators.required]))
8 });
```

Estados de un formulario

Los estados de un formulario vienen determinados por los estados de cada uno de los elementos que componen el formulario, mantienen tres estados que cambian entre estos pares:

- **pristine** ↔ **dirty**: en la carga inicial del formulario el elemento tiene estado “pristine” que quiere decir que no ha sido modificado por el usuario, en cuanto el usuario modifica el valor de algún campo, éste pasa a estado “dirty”, y ya no vuelve a estado “pristine” hasta que el formulario no vuelve a cargarse.
- **untouched** ↔ **touched**: en la carga inicial del formulario todos los elementos están con estado “untouched” que quiere decir que no se ha hecho foco sobre ellos, en el momento en el que pierden el foco (aunque no se haya modificado el valor) pasan a estado “touched” y no pueden volver a estado “untouched” hasta que no se vuelve a cargar el formulario.
- **invalid** ↔ **valid**: este par de estados son los más cambiantes, depende del cumplimiento de las validaciones que se le haya asignado al elemento tendrá un estado u otro que volverá a calcularse cuando el usuario modifique el valor del elemento.

Por tanto el estado general del formulario dependerá del estado particular de cada uno de los elementos; es decir, en la primera carga tendrá estado “untouched” y “pristine” cuando uno solo de ellos haya sido modificado el estado del formulario pasará a “dirty”, cuando en uno de ellos se haya perdido el foco el estado pasará a “touched” y no será “valid” hasta que todos los elementos del formulario tengan estado “valid”.

En base a estos estados Angular varía de forma automática y transparente al desarrollador la clase CSS correspondiente permitiendo definir estilos en función del estado del formulario; así podemos definir los siguientes estilos CSS: `*.ng-valid`, `*.ng-invalid`, `*.ng-touched`, `*.ng-untouched`, `*.ng-pristine`, `*.ng-dirty` y todas sus posibles combinaciones: `.ng-touched.ng-invalid`, `.ng-pristine.ng-untouched`,...

También tenemos que tener en cuenta estos estados cuando queramos mostrar los errores del formulario. En el ejemplo vemos cómo mostrar los errores de un determinado campo sólo cuando el estado del campo sea “dirty” y no sea “valid”.

```

1  <input type="text" formControlName="name" />
2
3  <div *ngIf="form.get('name').dirty && !form.get('name').valid">
4
5      <p *ngIf="form.get('name').hasError('minlength')">
6          El nombre debe tener al menos 4 caracteres
7      </p>
8
9      <p *ngIf="form.get('name').hasError('startsWithNumber')">
10         El nombre ya existe
11     </p>
12
13 </div>

```


De esta forma a modo de depuración podemos ver por pantalla los distintos estados simplemente interpolando las variables que nos resulten útiles y podemos utilizar el valor booleano de ellas para definir la directiva “disabled” de un elemento, por ejemplo, el botón de envío y que solo se habilite cuando el formulario tenga estado “valid”.

```

1  <form [formGroup]="form">
2
3  <p>Username: <input type="text" formControlName="username"></p>
4
5  {{ form.get('username').valid }}
6
7  <p>Password: <input type="password" formControlName="password"></p>
8
9  {{ form.get('password').valid }}
10
11 <button (click)="send()" [disabled]="!form.valid">Send</button>
12
13 {{ form.valid }}
14
15 </form>

```

Componente para mostrar los textos de error

Ya hemos visto una forma muy manual de mostrar los errores por pantalla, como ves es poco reutilizable y bastante repetitiva, así que vamos a crear un componente para mostrar los textos de error.

El código del componente quedaría de la siguiente forma:

```

1  import { FormControl } from '@angular/forms';
2  import { AfterViewInit, Component, Input, OnInit } from '@angular/core';
3
4  @Component({
5    selector: 'app-error-messages',
6    templateUrl: './error-messages.component.html',
7    styleUrls: ['./error-messages.component.css']
8  })
9  export class ErrorMessagesComponent implements OnInit, AfterViewInit {
10
11    @Input('control') control: FormControl;
12    @Input('customErrors') customErrors: Object = {};
13    errorMessages: Object;
14

```

```
15  constructor() { }
16
17  ngOnInit() {
18    this.errorMessages = {
19      'required': 'This field must not be empty',
20      'minlength': 'Sorry, this field is too short',
21      'maxlength': 'Sorry, this field is too long',
22      'pattern': 'Sorry, this is not valid'
23    };
24  }
25
26  ngAfterViewInit() {
27
28    Object.keys(this.customErrors).forEach((errorType) => {
29      this.errorMessages[errorType] = this.customErrors[errorType];
30    });
31
32  }
33
34  get errorMessage() {
35    for (const error in this.control.errors) {
36      if (this.control.errors.hasOwnProperty(error)
37        && (this.control.touched || (this.control.asyncValidator !== null
38          && !this.control.pristine))) {
39        return this.errorMessages[error];
40      }
41    }
42
43    return null;
44  }
45
46 }
```

Como ves definimos dos propiedades de entrada con `@Input`; la primera para recibir el control del formulario asociado, y la segunda por si queremos especificar nuevos mensajes de error, si los de por defecto no cubren con todos los casos.

En el lifecycle “AfterView”, es decir, cuando nos aseguramos que la vista ha sido instanciada, utilizamos el input “customErrors” para establecer los nuevos mensajes y sobrescribir los de por defecto.

En la función `getErrorMessage` establecemos el atributo `errorMessage` con el valor del texto que le corresponda.

Ahora en el template asociado pintamos el atributo `errorMessage` de este modo:

```
1 <div class="async-pending" *ngIf="control.pending">
2   Comprobando...
3 </div>
4 <div *ngIf="errorMessage !== null">
5   <ul><li>{{errorMessage}}</li></ul>
6 </div>
```

Definimos un div con el estado de pending del campo para cuando tenga que evaluar un validador asíncrono; y mostramos el error en formato de lista.

Aquí podéis mostrar el error como queráis y a través del fichero .css asociado y las clases predefinidas darle más o menos estilo.

Ahora simplemente cambiamos el template del formulario para hacer uso del componente.

```
1 <input type="text" formControlName="name" />
2
3 <app-error-messages [control]="form.get('username')" [customErrors]="usernameErr\
4 orsMessages"></app-error-messages>
```

En la lógica del componente tenemos que tener un atributo “usernameErrorMessages” que inicializamos en el método ngOnInit, por ejemplo, de esta forma:

```
1 ngOnInit() {
2   this.form = new FormGroup({
3     username: new FormControl('',
4       Validators.compose([Validators.required, CommonValidator.startWithNumber\
5 ]),
6     Validators.composeAsync([CommonValidator.userTaken])),
7     password: new FormControl('', Validators.compose([Validators.required]))
8   });
9
10  this.usernameErrorsMessages = {
11    'required': 'El username es requerido',
12    'userTaken': 'El username ya está pillado',
13    'startWithNumber': 'El username no puede empezar por número'
14  };
15
16 }
```

Http: comunicación con servidor remoto

Uno de los imprescindibles de toda aplicación frontend es que tiene que comunicarse con un backend para recibir los datos, procesarlos si fuera necesario y mostrarlos por pantalla, lo que obliga a los desarrolladores a implementar mucha lógica que no es propia realmente del negocio.

Angular simplifica esta lógica ofreciéndonos el servicio `HttpClient`, que de una manera muy sencilla nos permite realizar todas las operaciones que necesitamos para conectar con los servicios externos de forma asíncrona para que la aplicación no se quede bloqueada hasta que le llegue la respuesta, sino que se registra un manejador que actúa cuando recibe la respuesta del servidor.

Historia de la asincronía

Como de todos es sabido, JavaScript es monohilo por lo tanto cualquier proceso que lleve mucho tiempo dejará bloqueado el hilo principal a la espera de su finalización para poder continuar con las siguientes instrucciones del programa.

Para resolver este problema se crearon las funciones asíncronas o manejadores que de un modo u otro, no bloquean el hilo principal y se quedan esperando al evento de finalización del proceso pesado para recuperar la información procesada.

Callbacks

Históricamente este tipo de manejadores eran los callbacks que permiten pasar la función manejadora a ejecutar, esto hace que en muchas ocasiones tengamos que enlazar unos callbacks con otros haciendo que nuestro código sea más difícil de entender y por tanto de mantener, ¿quién no ha oído hablar del infierno de los callbacks?

```
1 var function1 = function(callback){
2   function2(function(error, result)){
3     if (error) {callback(error); return;}
4     function3(result, function(error, result){
5       if (error) {callback(error); return;}
6       function4 (result, function(error, result){
7         if (error) {callback(error); return;}
8         function5(result, function(error, result){
9           // ...
10        });
11      });
12    });
13  });
14 };
```

Promesas

La primera solución que se le ha buscado a este problema y la que actualmente más se utiliza son las promesas, que devuelven un objeto con el estado y el resultado de la petición cuando se produce la respuesta.

Luego podemos decidir qué código ejecutar en función de esta respuesta, donde utilizamos el bloque “then” para manejar la respuesta correcta y el bloque “catch” para manejar el error en la respuesta.

Actualmente en ES5 podemos hacer uso de ellas a través de librerías de terceros como “q” o el servicio “\$http” de AngularJS. ES6 ya las incorpora de forma nativa en el lenguaje con la clase Promise.

Esta clase maneja dos funciones “resolve”, para devolver la respuesta correcta y “reject”, para devolver el error si se produce. Aquí vemos, a modo de ejemplo, la definición de una función asíncrona que devuelve una promesa pasado un segundo, donde es correcta cuando n es 1 y en caso contrario devuelve un error.

```
1 function obtenerDatos () {
2   return new Promise((resolve, reject) => {
3     let n = Math.floor(Math.random()*2) + 1;
4     setTimeout(() => {
5       if (n === 1) {
6         resolve('Datos obtenidos');
7       }else{
8         reject('Error')
9       }, 1000);
10    });
11  });
12 }
```

```
10    });  
11 }
```

Este tipo de estructura ya la avanzamos cuando hablábamos de las validaciones asíncronas en los formularios.

El método que haga uso de esta función necesita definir los bloques “then” y “catch” de esta manera:

```
1  obtenerDatos()  
2  .then(data => {  
3    console.log(data);  
4  })  
5  .catch(err => {  
6    console.log(err);  
7  });
```

De esta forma cuando se hace la llamada `obtenerDatos()`, el hilo principal continua su ejecución y pasado un segundo, cuando se resuelve la promesa, ejecuta bien el bloque de “then” o bien el bloque de “catch” en función de la respuesta obtenida.

Observables

Los observables son el concepto más actual para el manejo de la asincronía. Se basa en el patrón Observer, donde los clientes se suscriben a un determinado canal para ser informados de los cambios que se produzcan.

Este concepto pertenece a un concepto más amplio que es la programación reactiva y que Angular utiliza de serie gracias a la librería RxJS. Presenta dos ventajas fundamentales con respecto a las promesas: por un lado tienen ejecución lazy, es decir, solo se ejecutan si hay algún cliente suscrito al canal y la segunda es que el canal puede ser cancelado en cualquier momento.

Además podemos tener n suscriptores a un mismo canal que recibirán la misma información al mismo tiempo, pudiéndose utilizar como mecanismo de comunicación de información.

Para la creación de un canal utilizamos la clase Observable proporcionada por RxJS, este sería el equivalente al caso anterior, para crearlo utilizamos la función “create”, con la función “next” indicamos un nuevo dato en el canal y con “error” indicamos que ha habido un error.

```
1 function obtenerDatos () {
2   return Observable.create((obs) => {
3     let n = Math.floor(Math.random()*2) + 1;
4     setTimeout(() => {
5       if (n === 1) {
6         obs.next('Datos obtenidos');
7       }else{
8         obs.error('Error')
9       }, 1000);
10    });
11  }
```

El cliente se suscribe al canal mediante la función “subscribe” e internamente maneja tres estados definidos por posición: el primero de ellos es el correcto dónde llegarán los datos, el segundo es cuando ocurre algún error en la petición y el tercero cuando se finaliza la petición.

```
1 const sub = obtenerDatos().subscribe(
2   data => {
3     console.log(data);
4   },
5   err => {
6     console.log(err);
7   },
8   () => {
9     console.log('Proceso completado');
10  }
11 );
12
13 ngOnDestroy() {
14   sub.unsubscribe();
15 }
```

Para evitar problema de memory leak es importante almacenar la suscripción en una variable de tipo Subscription para poder ejecutar el método unsubscribe en el momento que sea necesario, por ejemplo, cuando se destruya la instancia del componente.

Servicio HttpClient

A partir de la versión 4.3 se mejora el servicio Http pasándose a llamar HttpClient incluyéndolo dentro del common de Angular. Las mejoras que proporciona son el

parametrizado de la respuesta que ahora por defecto es JSON y, sobre todo, la inclusión del concepto de interceptor.

Para poder hacer uso del servicio HttpClient de Angular tenemos que importar el módulo HttpClientModule que se encuentra dentro de @angular/common/http y contiene todos los providers y elementos necesarios para conectarnos con servidores remotos.

De este modo vamos a la configuración del módulo que se vaya a conectar con un servidor remoto y añadimos el módulo de HttpClientModule, además podemos añadir un objeto de configuración general de la aplicación o hacer uso de los environments de angular-cli.

```
1 import { NgModule } from '@angular/core'
2 import { HttpClientModule } from '@angular/common/http'
3
4 const config = {api: 'http://servicio.api'};
5
6 @NgModule({
7   imports: [HttpClientModule],
8   providers: [{provide: 'config', useValue: config}]}
9 })
```

Ahora creamos un servicio que va a inyectar una instancia del servicio HttpClient y el fichero de configuración. Este servicio será el encargado de hacer de proxy con el servidor remoto a través de los métodos del servicio HttpClient.

Recuperación de datos

Para hacer la recuperación de datos vamos a implementar el método getUsers dentro de esta clase, el cual gracias a la nueva característica de Type Checking del servicio HttpClient va a devolver directamente la información de los usuarios.

Para ello creamos una interfaz con los datos del usuario:

```
1 export interface User {
2   login: string;
3   name: string;
4 }
```

Y lo utilizamos en nuestra clase:


```
1 import { HttpClient, Response } from '@angular/common/http';
2 import { Inject } from '@angular/core';
3
4 export class ApiProxyService {
5   constructor(private http: HttpClient,
6               @Inject('config') private config){}
7
8   getUsers(): Observable<User[]> {
9     return this.http.get<User[]>(this.config.api);
10  }
11
12 }
```

Ahora en el componente inyectamos este servicio (no inyectamos nunca el servicio HttpClient directamente dentro de un componente) y creamos dos atributos: users de tipo de User[] y sub de tipo Suscription, los cuales vamos a utilizar en la implementación del método searchUsers. Es importante implementar la interfaz OnDestroy para hacer la desuscripción del Observable y evitar problemas de “memory leak”, a no ser que se haga uso del pipe async.

```
1 import { User } from '../model/user';
2 import { ApiProxyService } from '../api.proxy.service';
3 import { OnDestroy } from '@angular/core';
4 import { Suscription } from 'rxjs/Suscription';
5
6 export class UsersComponent implements OnDestroy {
7   users: User[];
8   sub: Suscription;
9
10  constructor(private service: ApiProxyService) {}
11
12  searchUsers() {
13    this.sub = this.service.getUsers().subscribe(
14      response => this.users = response.body,
15      error => console.log(error)
16    )
17  }
18
19  ngOnDestroy() {
20    this.sub.unsubscribe();
21  }
22
23 }
```

En el template del componente podemos recorrer la colección con un ngFor para mostrar la información de cada usuario:

```
1 template: `  
2   <ul>  
3     <li *ngFor="let user of users">  
4       {{user.login}}  
5       {{user.name}}  
6     </li>  
7   </ul>`
```

Otra opción que tenemos para visualizar los datos es utilizar el pipe async que trabaja directamente con el Observable asumiendo el control por nosotros, es decir, se encarga de desuscribirlo en el momento oportuno.

Para hacer uso de él tenemos que modificar la implementación del método searchUsers y el tipo de dato de users que pasa a ser Observable de tipo array de users.

```
1 import { User } from './model/user';  
2 import { ApiProxyService } from './api.proxy.service';  
3 import { OnDestroy } from '@angular/core';  
4 import { Subscription } from 'rxjs/Subscription';  
5  
6 export class UsersComponent implements OnDestroy {  
7  
8   users: Observable<User[]>;  
9  
10  constructor(private service: ApiProxyService) {}  
11  
12  searchUsers() {  
13    this.users = this.service.getUsers();  
14  }  
15 }
```

Ahora en el template solo tenemos que hacer uso del pipe async de esta manera:

```
1 template: `  
2   <ul>  
3     <li *ngFor="let user of (users | async)">  
4       {{user.login}}  
5       {{user.name}}  
6     </li>  
7   </ul>`
```

Como hemos dicho esta versión asume que la respuesta siempre es JSON, en caso de que la respuesta no fuera en formato JSON tenemos que especificar el formato con la siguiente sintaxis:

```
1 http.get('/textfile.txt', {responseType: 'text'});
```

En caso de necesitar de parámetros esto se hace con la clase `HttpParams()` de esta forma:

```
1 http.post('url/api', body, {  
2   params: new HttpParams().set('clave', 'valor')  
3 })
```

En caso de necesitar modificar las cabeceras se hace con la clase `HttpHeaders()` de esta forma:

```
1 addUser(user: User): Observable<Response> {  
2   let headers: HttpHeaders = new HttpHeaders();  
3   headers = headers.set('X-API-TOKEN', 'token');  
4   headers = headers.set('Content-Type', 'application/json');  
5  
6   return this.http.post(this.config.api + '/user',  
7     {login: user.login, name: user.name}, {  
8       headers: headers  
9     });  
10  
11 }
```

Ten en cuenta que el método `.set` es inmutable por tanto tienes que asignarlo a la variable `headers` para que tenga efecto.

En caso de necesitar inspeccionar toda la respuesta como se hacía en versiones anteriores tenemos que ejecutar la llamada de esta forma:

```
1 http.get<any>('/data.json', {observe: 'response'})
```

De forma que podremos verificar el “status” de la petición y tendremos que recuperar la información en la propiedad “body” de la respuesta.

Interceptores

El concepto de interceptor nos permite interceptar la petición y realizarle una serie de modificaciones. Para hacer un interceptor tenemos que crear una clase que implemente la interfaz `HttpInterceptor`, este sería un ejemplo, de un interceptor que no hace nada.

```
1 import {Injectable} from '@angular/core';
2 import {HttpEvent, HttpInterceptor, HttpHandler, HttpRequest} from '@angular/com\
3 mon/http';
4
5 @Injectable()
6 export class NoopInterceptor implements HttpInterceptor {
7   intercept(req: HttpRequest<any>,
8     next: HttpHandler): Observable<HttpEvent<any>> {
9     return next.handle(req);
10  }
11 }
```

Como ves recibe la petición y el siguiente interceptor en la cadena, no hace nada y llama al siguiente interceptor. Tenemos que tener en cuenta que **el orden en la cadena de interceptores lo va a determinar el orden de definición de providers**, el cual lo podemos hacer de la siguiente forma:

```
1 import { NgModule } from '@angular/core';
2 import { HTTP_INTERCEPTORS } from '@angular/common/http';
3
4 @NgModule({
5   providers: [{
6     provide: HTTP_INTERCEPTORS,
7     useClass: NoopInterceptor,
8     multi: true,
9   }],
10 })
11 export class AppModule {}
```

A continuación resolvemos los casos de uso más comunes con interceptores:

Modificar cabeceras

El caso de uso más común es modificar la cabecera de autenticación para añadir el token de acceso. Esto lo podemos hacer de la siguiente forma:

```
1  import {Injectable} from '@angular/core';
2  import {HttpEvent, HttpInterceptor, HttpHandler, HttpRequest} from '@angular/com\
3  mon/http';
4
5  @Injectable()
6  export class AuthInterceptor implements HttpInterceptor {
7
8      constructor(private auth: AuthService) {}
9
10     intercept(req: HttpRequest<any>,
11         next: HttpHandler): Observable<HttpEvent<any>> {
12
13         // Obtenemos el token
14         const token = this.auth.getToken();
15         // Importante: modificamos de forma inmutable,
16         //haciendo el clonado de la petición
17         const authReq = req.clone(
18             {headers: req.headers.set('Authorization', token)}
19         );
20         // Pasamos al siguiente interceptor de
21         //la cadena la petición modificada
22         return next.handle(authReq);
23     }
24 }
```

Logging

Otro caso de uso común es el de querer saber cuánto tiempo tarda en resolverse una petición. Para ello podemos implementar un nuevo interceptor de este tipo:

```

1  import { do } from 'rxjs/operators/do';
2
3  export class TimingInterceptor implements HttpInterceptor {
4
5      constructor() {}
6
7      intercept(req: HttpRequest<any>,
8          next: HttpHandler): Observable<HttpEvent<any>> {
9
10         const started = Date.now();
11         return next
12             .handle(req).pipe(
13             do(event => {
14                 if (event instanceof HttpResponse) {
15                     const elapsed = Date.now() - started;
16                     console.log(`Request for ${req.urlWithParams} took ${elapsed} ms.`);
17                 }
18             })
19         );
20     }
21 }

```

Manejar errores

También podemos definir un interceptor para manejar los errores que se produzcan en la comunicación con el servidor.

```

1  import { Injectable } from '@angular/core';
2  import { Events } from 'ionic-angular';
3  import { HttpEvent, HttpHandler,
4  HttpInterceptor, HttpRequest,
5  HttpResponse } from '@angular/common/http';
6  import { Observable } from 'rxjs/Observable';
7  import { catchError } from 'rxjs/operators';
8  import { _throw } from 'rxjs/observable/throw';
9
10 @Injectable()
11 export class ErrorInterceptor implements HttpInterceptor {
12
13     constructor(){}
14
15     intercept(req: HttpRequest<any>,

```

```

16     next: HttpHandler): Observable<HttpEvent<any>> {
17
18     return next
19         .handle(req).pipe(
20             catchError((response: any) => {
21                 if (response instanceof HttpErrorResponse) {
22                     console.log('response in the catch: ', response);
23                 }
24                 return _throw(response);
25             })
26         );
27     }
28
29 }

```

Controlar peticiones a través de timeout

También podemos crear un interceptor para controlar el timeout de todas las peticiones y disparar un error si la petición se demora más de un umbral fijado. Este sería un ejemplo de implementación gracias como no a la programación reactiva con el operador “timeoutWith” que pasado el tiempo estipulado devuelve el observable definido en el segundo parámetro, en este caso, una llamada a un bus de eventos para mostrar por pantalla el error:

```

1  import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from '@angular/c\
2  ommon/http';
3  import { Injectable } from '@angular/core';
4  import { Events } from 'ionic-angular';
5  import { Observable } from 'rxjs/Observable';
6  import { defer } from 'rxjs/observable/defer';
7  import { _throw } from 'rxjs/observable/throw';
8  import { timeoutWith } from 'rxjs/operators/timeoutWith';
9
10 @Injectable()
11 export class TimeoutInterceptor implements HttpInterceptor {
12
13     constructor(private events: Events){}
14
15     intercept(req: HttpRequest<any>,
16         next: HttpHandler): Observable<HttpEvent<any>> {
17
18         return next

```

```
19         .handle(req).pipe(  
20             timeoutWith(10000, defer(() => _throw(this.events.publish('show:erro\  
21 r', {dataerror: 'Error de timeout'}))))  
22         );  
23     }  
24 }
```

El uso del operador `defer` es para evitar que la aplicación de un error de runtime y el mensaje por pantalla al usuario se muestre de forma correcta.

Escucha de eventos de progreso

Otra de las nuevas características que incluye `HttpClient` es la de poder interactuar con los eventos que se producen en el progreso de una subida de ficheros. Para ello tenemos que hacer la petición de esta forma:

```
1 const req = new HttpRequest('POST', '/upload/file', file, {  
2     reportProgress: true  
3 });
```

Entonces podemos subscribirnos al método `request()` que nos devolverá los eventos con lo que podemos interactuar:

```
1 http.request(req).subscribe(event => {  
2     if (event.type === HttpEventType.UploadProgress) {  
3         // Este evento nos permite calcular el % del progreso  
4         const percentDone = Math.round(100 * event.loaded / event.total);  
5         console.log(`File is ${percentDone}% uploaded.`);  
6     } else if (event instanceof HttpResponse) {  
7         console.log('File is completely uploaded!');  
8     }  
9 });
```

Uso de async pipe con objetos

A partir de Angular 4 podemos hacer uso del pipe `async` con objetos siempre y cuando estos sean Observables. De esta forma podemos definir en un componente un Observable de un tipo complejo de dato.


```
1 import { OnInit } from '@angular/core';
2 import { Observable } from 'rxjs/Observable';
3 import { of } from 'rxjs/observable/of';
4
5 export class ExampleComponent implements OnInit {*
6   person$: Observable<{nombre: string, apellido: string}>;
7
8   ngOnInit() {
9     this.person$ = of({nombre: 'Rubén', apellido: 'Aguilera'});
10  }
11 }
```

Y en el template podemos utilizar el pipe async de esta forma:

```
1 <div *ngIf="person$ | async as me">
2   <p>{{me.nombre}}</p>
3   <p>{{me.apellido}}</p>
4 </div>
```

Como regla de estilo todavía no escrita los observables se identifican con un \$ al final del identificador.

El equipo de Angular recomienda el uso de async para trabajar con Observables mejor que hacer subscribe en el componente, ya que este pipe se encarga de gestionar el ciclo de vida del Observable y de manejar la desuscripción de forma transparente al desarrollador.

Testing

Introducción

Cuando hablas con desarrolladores te das cuenta de que muy pocos saben y hacen tests de sus implementaciones. Entonces, les pregunto “¿cómo me demuestras que lo que has hecho está bien?” Siempre me suelen contestar “pues mira abro la aplicación, pincho en el botón y sale lo que tiene que salir”, me dicen orgullosos. Entonces es cuando les digo, “y si te pido que esto me lo demuestres cada vez que hagas un cambio en el código junto con las otras historias de usuario que has implementado para saber que esto se puede poner en producción...” entonces es cuando algunos cambian el rictus y otros, los más atrevidos, dicen “bueno pero esto ya lo validará el equipo de QA que para eso está”.

Y ese, compañeros, es uno de los mayores problemas de las grandes compañías que tienen un equipo de QA, que encima no automatiza las pruebas con lo cual el tiempo desde que una persona de negocio tiene una superidea, que va a reportar millones a su empresa, es directamente proporcional al tiempo que el equipo de QA, con sus pruebas en Excel supercurradas, va a tardar en validar ese desarrollo para su puesta en producción; perdiendo de esta forma la ventana de oportunidad y por tanto la ganancia de la idea.

¿Cómo podemos los desarrolladores minimizar este periodo al máximo? La respuesta es sencilla... estando seguros en todo momento que lo que se va a subir a producción es funcional y técnicamente correcto desde la fase de desarrollo; y esto solo se puede conseguir con tests y procesos automáticos que podamos repetir una y otra vez. De esta forma podríamos hacer subidas a producción con total confianza varias veces al día si fuera necesario.

Si buscas un framework JavaScript que te ayude con el testing de tus aplicaciones, sin duda tienes que decantarte por Angular, ya que te proporciona de serie un mecanismo de inyección de dependencias que se integra perfectamente con Jasmine y que te permite la utilización de fakes para tests unitarios y de integración; y la integración con Protractor para tests de aceptación.

Tests unitarios y de integración con Jasmine

No se puede empezar a hablar de testing en JavaScript sin mencionar a Jasmine. Hay otros frameworks de testing pero Jasmine es el más utilizado por encima de Mocha o Jest que son actualmente las otras alternativas.

Sus características principales son que no depende de ninguna otra librería JavaScript, tiene una sintaxis obvia y limpia que facilita la escritura de los tests, pudiendo añadir aseveraciones que nos permitan determinar si la ejecución ha sido correcta o no.

Además se puede utilizar para implementar tests unitarios y de integración, así como implementar tests de aceptación gracias a su integración con Protractor.

Test Suite

Para quien no esté familiarizado con el concepto de Test Suite simplemente es una forma de organizar casos de tests que estén relacionados.

La sintaxis comienza con la palabra reservada “describe” seguida de un texto descriptivo y una función anónima que contiene los casos de test que se van a ejecutar.

```
1 describe ('texto descriptivo', () => {  
2     //Aquí van los casos de test relacionados  
3 })
```

Test Case

Dentro de un test suite puede haber uno o más casos de test, donde va a residir la lógica de verificación de nuestros tests. Se utiliza la palabra reservada “it”.

```
1 describe ('texto descriptivo', () => {  
2     it ('texto descriptivo', () => {  
3         //Caso de test  
4     })  
5  
6     xit ('texto descriptivo', () => {  
7         //Caso de test  
8     })  
9  
10    fit ('texto descriptivo', () => {  
11        //Caso de test  
12    })  
13 })
```

Fíjate que podemos anteponer a it las letras x y f: la primera indica que queremos ignorar ese caso de test para que no se tenga en cuenta a la hora de ejecutarlos, y la segunda hace que solo se ejecute ese test. También podemos anteponerlas a la palabra reservada “describe” para que afecte a todos los casos de test que tenga relacionados.

Matchers

Son una serie de funciones definidas en Jasmine que nos permiten añadir lógica de verificación a los casos de tests gracias a la función `expect`.

Tienes que tener en cuenta que JavaScript es un poco especial con el valor booleano de las variables de forma que cualquier variable que tenga valor: `false`, o `0`, o cadena vacía, o “`undefined`”, o `null`, o `Nan` será `false`, y el resto de casos que se te ocurran serán `true`.

- **toBe:** devuelve `true` haciendo la comparación de los elementos con el triple igual.
- **toEqual:** es similar al anterior pero se utiliza más para literales y objetos.
- **toMatch:** compara expresiones regulares
- **toBeDefined:** comprueba si el elemento está definido
- **toBeUndefined:** comprueba si el elemento devuelve `undefined`.
- **toBeNull:** comprueba si el elemento es nulo
- **toBeTruthy:** hace cast y devuelve `true` si el valor está fuera de los posibles valores antes mencionados.
- **toBeFalsy:** hace cast y devuelve `true` si el valor está dentro de los posibles valores antes mencionados.
- **toContain:** para comprobar si un elemento está presente en un array.
- **toBeLessThan:** compara dos elementos con el menor que.
- **toBeGreaterThan:** compara dos elementos con el mayor que.
- **toBeCloseTo:** comprueba la precisión de una operación matemática.
- **toThrow:** comprueba si se ha lanzado una excepción.
- **not.:** anteponiendo `not` a cualquiera de los anteriores matchers cambiamos el valor booleano esperado.

beforeEach() y afterEach()

A fin de poder hacer reutilización de código en nuestros tests Jasmine nos ofrece las funciones `beforeEach()` y `afterEach()`

El contenido de la función `beforeEach()` se va ejecutar siempre antes de cada uno de los casos de test y el contenido de la función `afterEach()` se ejecutará justo después de la ejecución de cada uno de los casos de tests.

En el ejemplo podemos ver como la variable “`count`” toma el valor 10 antes de la ejecución de añadir uno a la variable, de forma que la aseveración que lo compara con el valor 11 es afirmativa, y justo después el valor de `count` se establece a `0`.

```
1 describe('A spec suite', () => {
2   var count
3   beforeEach(() => {
4     count = 10
5   })
6   afterEach(() => {
7     count = 0
8   })
9   it('should add one to count', () => {
10    count += 1
11    expect(count).toEqual(11)
12  })
13 })
```

Fíjate cómo hacemos uso de `expect` junto con el matcher “`toEqual`” para verificar que el resultado es el esperado.

En el siguiente ejemplo podemos ver la unión de todos los conceptos para verificar el funcionamiento de la clase `Calculadora`.

```
1 import {Calculadora} from './calculadora'
2
3 describe('Calculadora', () => {
4   let calc;
5   beforeEach(() => {
6     calc = new Calculadora()
7   })
8   it('should add two numbers', () => {
9     let a:number = 3
10    let b:number = 2
11    expect(calc.add(a, b)).toEqual(5)
12  })
13 })
```

Karma

Vale, ya tenemos los test implementados pero ahora, ¿cómo los ejecuto?

Para eso tenemos Karma, que es un test runner que se integra con Jasmine para ayudarnos a lanzar los tests unitarios y de integración una y otra vez contra los navegadores que consideremos más relevantes.

En la configuración podemos indicarle qué frameworks estamos utilizando y qué tipo de informes queremos obtener como resultado de la ejecución de los tests, los más importantes son los de cobertura y éxito.

Si haces uso de angular-cli, ya tienes el fichero de configuración karma.conf.js configurado con lo básico para empezar a lanzar los tests.

Así que para lanzarlos en un proyecto con angular-cli, simplemente tenemos que ejecutar:

```
1 $> npm run test
```

Y si queremos que calcule la cobertura:

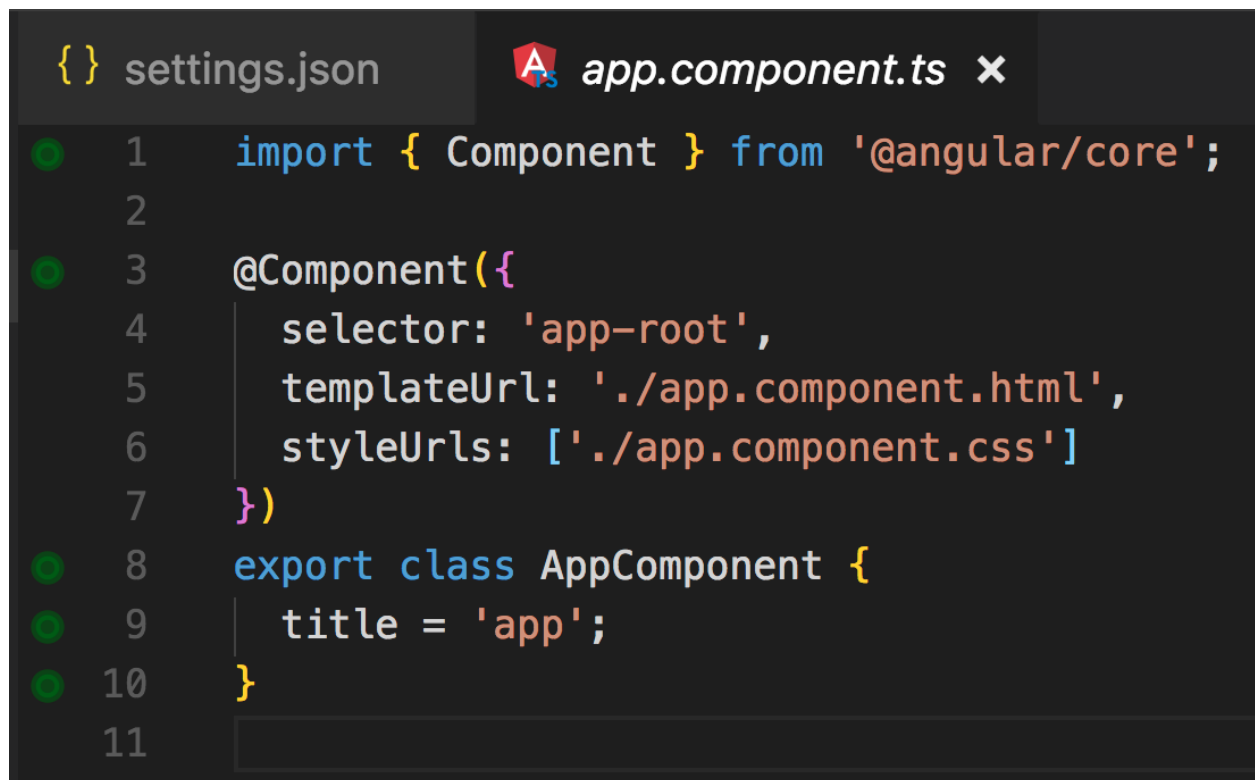
```
1 $> npm run test -- --code-coverage
```

Este último creará una carpeta “coverage” en nuestro proyecto con el informe de cobertura de los tests ejecutados.

Con respecto a la cobertura de test podemos configurar el plugin bma-coverage para que se muestre esta información en el propio código con un punto rojo si no tiene cobertura y un punto verde si la tiene.

Para configurarlo tenemos que ir a “settings” de Visual Studio Code y añadir la configuración de donde se encuentra el fichero .lcov generado al ejecutar los tests con el flag `--code-coverage`.

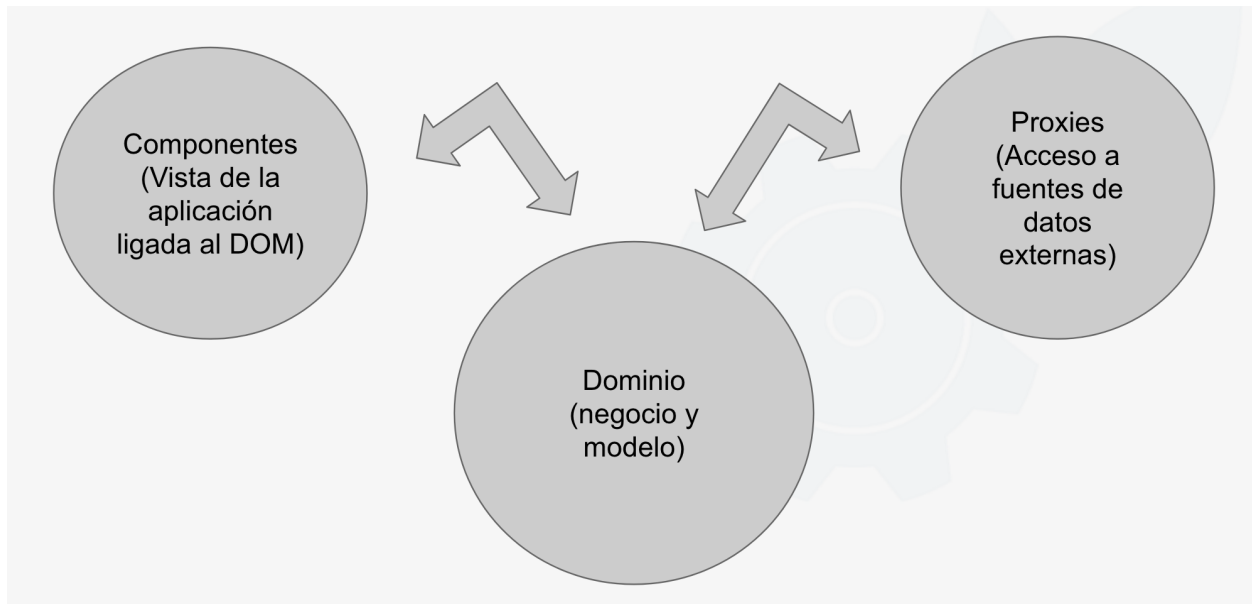
```
1 {
2   "bma-coverage": {
3     "lcovs": [
4       "coverage/lcov.info"
5     ]
6   }
7 }
```



```
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'app-root',
5    templateUrl: './app.component.html',
6    styleUrls: ['./app.component.css']
7  })
8  export class AppComponent {
9    title = 'app';
10 }
11
```

Arquitectura de una aplicación testeable

Para facilitar el testing de una aplicación con Angular tenemos que tener en cuenta que depende mucho de la arquitectura y el diseño de la aplicación.



De esta forma el primer proceso que tenemos que realizar es de análisis de la aplicación para clasificar cada una de las clases en tres grupos distintos:

- **Componentes:** son los más fáciles de identificar, de hecho están marcados con el decorador `@Component` y representan la interfaz de usuario de nuestra aplicación. Tenemos que tener en cuenta que es la parte más cercana al DOM lo que complica el testing. Por eso nos tenemos que preocupar de que los componentes no presenten lógica de negocio más allá de gestionar el estado de la vista e invocar a la lógica de negocio. También aplica a las clases decoradas con `@Directive`.
- **Proxies:** en una aplicación Angular también son fácilmente identificables, son aquellos que inyectan el servicio `HttpClient` para trabajar con la información del servidor. También son más difíciles de testear al ser la parte más cercana al protocolo HTTP, por lo que tenemos que limitar su utilización a comunicar con el servidor y devolver la información a clases que se encarguen de adaptar esa información a las necesidades de negocio.
- **Dominio:** los dos grupos anteriores realmente no representan nada específico de nuestra aplicación, cualquier aplicación de Angular tiene una vista y tiene accesos HTTP para manejar la información. Lo que realmente define una aplicación es su dominio. El dominio representa el corazón de nuestra aplicación y se compone de las entidades, es decir, las clases que modelan los objetos de nuestra aplicación (típicamente interfaces) y las clases que aplican las reglas de negocio que manejan los datos de nuestra aplicación. Entre las clases que existen en el dominio destacan: servicios que aplican las reglas de negocio, adaptadores de información, presentadores, pipes, ... Todas estas clases son mucho más fáciles de testear que las de los otros grupos al no tener dependencias externas con el DOM ni con el protocolo HTTP.

Dicho esto las situaciones que tenemos que evitar para facilitar el testing son, entre otras, la inyección del servicio `HttpClient` a través del constructor de un componente o la inclusión de lógica de adaptación dentro de un método que recupera información del servidor.

TestBed

TestBed es la pieza de Angular que nos facilita la implementación de tests unitarios y de integración.

Ya vimos en el módulo de servicios y dependencias como Angular tiene un mecanismo de inyección de dependencias a través de constructor que nos permite desacoplar nuestras clases y hacer que sea más sencillo sustituir unas por otras.

Para ello, vimos como las dependencias había que declararlas como providers, ya fuera en el módulo o en la sección providers de los componentes. Pues bien, en los tests no tenemos acceso al módulo para hacer esta declaración, es por eso que tenemos la clase **TestBed** que nos permite hacer la declaración de estos providers dentro de la función `configureTestingModule`, con exactamente las mismas características que ya hemos visto en la declaración de los providers; así que como en el ejemplo, fíjate lo sencillo que es declarar un fake dentro de un test.

```
1 import {TestBed} from '@angular/core/testing'
2 import {EngineService} from '../engine/engine.service'
3 import {EngineServiceMock} from '../mocks/engine.mock'
4
5 beforeEach(() => {
6   TestBed.configureTestingModule({
7     providers: [CarService,
8       {provide: EngineService, useClass: EngineServiceFake}]
9   })
10 })
```

Como ya sabemos el segundo paso es utilizar ese provider que previamente hemos declarado, esto lo hacemos a través del constructor de las clases, pero en el caso de los tests no tenemos constructor. En este caso lo realizaremos gracias a la función `get()` de **TestBed** que nos devuelve una instancia que previamente hemos declarado como provider, esto lo podemos hacer dentro de `beforeEach()` o en la lógica de un caso de test.

En el ejemplo vemos cómo inyectar un servicio dentro de un bloque `beforeEach()` para que la instancia pueda ser utilizada por todos los casos de tests.

```
1 describe ('should ...', () => {  
2  
3   let carService: CarService;  
4   beforeEach(()=>{  
5     carService = TestBed.get(CarService);  
6   })  
7  
8 })
```

Entonces cuando se ejecute el test realmente se estará invocando al fake de la clase EngineService que es utilizada dentro de CarService.

Situaciones de testing

Cuando utilizamos el comando “generate” de angular-cli, por defecto, nos incluye un fichero asociado con .spec, este fichero es el fichero de testing que ya viene con mucho código “boilerplate” relacionado con el uso de TestBed. Por lo que el trabajo del desarrollador se “limita” a configurarlo como veremos a continuación y añadir la lógica de verificación.

No borres el fichero .spec, es tu fiel amigo ante cualquier modificación futura para saber que no estamos rompiendo nada ;-)

Test de un pipe / servicio sin dependencias

El caso más sencillo de testing que nos podemos encontrar es el de probar una clase del dominio, ya sea pipe o servicio, que no depende de ninguna otra.

En este caso simplemente tenemos que declarar la clase como provider gracias a TestBed y llamar al método get() para recuperar la instancia de la clase.

Dentro del caso de test hacemos uso de la función expect para establecer las aseveraciones que consideremos oportunas, con la llamada a los métodos o propiedades de la clase.

```
1 import { TestBed } from '@angular/core/testing'
2 import { DecoratorPipe } from '../decorator.pipe'
3
4 describe('DecoratorPipe', () => {
5   beforeEach(() => {
6     TestBed.configureTestingModule({
7       providers: [DecoratorPipe]
8     });
9   });
10
11   it ('apply decorator',() => {
12     let pipe: DecoratorPipe = TestBed.get(DecoratorPipe);
13     expect(pipe.transform('Test', '***')).toBe('***Test***');
14   })
15 })
```

Fíjate cómo establecemos el tipo al identificador para que el IDE pueda autocompletarnos los métodos y atributos de la clase.

Test de un pipe/servicio con dependencias

En caso de que la clase del dominio tenga dependencias, es decir, que a través de su constructor se incluyan una o varias clases del dominio, debemos declarar en la función `configureTestingModule` de `TestBed` la clase principal y todas sus dependencias que en caso de querer ser cambiadas se podrá hacer fácilmente con la receta `useClass`.

De este modo nosotros solo tenemos que recuperar en el caso de test, o en un bloque `beforeEach`, la clase principal y Angular se encargará de inyectar las otras dependencias necesarias por nosotros que tendremos que especificar a través de la propiedad `"deps"`.

En el ejemplo, tenemos una clase `CalcService` cuya suma depende de la clase `AddService`, la cual queremos cambiar su implementación porque, por ejemplo, la operación se este realizando en un servidor y queremos quitarnos ese tiempo de espera en nuestros test unitarios.

Luego en el bloque solo recuperamos la clase principal y al llamar al método `add()` que es el que hace uso de la clase `AddService`, Angular lo sustituye por el Fake declarado, haciendo que no se produzca la llamada al servidor y que el test pase correctamente. Al fin y al cabo lo que queremos es probar la lógica de la clase principal no la conexión remota con el servidor de sus dependencias.

```
1 import { TestBed } from '@angular/core/testing'
2 import { CalcService } from '../calc.service'
3 import { AddServiceFake } from '../mocks/add.service.fake'
4
5 beforeEach(() => {
6   TestBed.configureTestingModule({
7     providers: [
8       {
9         provide: CalcService,
10        useClass: CalcService,
11        deps: [
12          {
13            {provide: AddService, useClass: AddServiceFake}
14          }
15        ]
16      }
17    ])
18
19    describe('CalcService Tests', () => {
20      it ('should add', () => {
21        let calcService = TestBed.get(CalcService);
22        expect(calcService.add(3,6)).toBe(9)
23      })
24    })
25  })
26 })
```

Tests asíncronos

Los tests asíncronos son los que prueban métodos que no detienen su ejecución esperando por un resultado, sino que registran una promesa u observable y cuando el resultado se produce lo devuelven. Lo más común es que estos métodos se encuentren en clases proxies de acceso a servidores remotos o clases de negocio que manejen promesas u observables.

Para facilitar estos tests Angular proporciona el método `async` que permite indicar que el caso de test se va a ejecutar de forma asíncrona y que lo tenga en cuenta a la hora de la ejecución.

```
1 import { TestBed, async, inject } from '@angular/core/testing'
2
3 beforeEach(() => {
4   TestBed.configureTestingModule({providers: [SumService]})
5 })
6
7 it('should sum two numbers async',
8   async(() => {
9     let sumService = TestBed.get(SumService)
10    sumService.executeAsync(4,6)
11      .then((value => {expect(value).toBe(10)}))
12    }))
13  })
```

Ten cuidado con los falsos positivos ya que si no utilizas la función `async` el test va a seguir pasando pero realmente no estará evaluando el contenido de la sección `then`. Además fíjate que el test tardará en ejecutarse el tiempo que tardé en ejecutar la acción por lo que la ejecución puede llegar a ser muy lenta y afectar a otros tests unitarios. En estos casos te aconsejo tratar estos test como de integración y separarlos de la ejecución de los tests unitarios para no penalizar el rendimiento de los tests unitarios.

Test de componente sin dependencias

Lo más importante en un componente es verificar que los atributos se establecen correctamente en base a las interacciones del usuario, más que si se renderiza en el DOM un determinado atributo.

Para ello lo que se utiliza es un fixture del componente que nos devuelve la instancia de la clase que representa el componente. De esta forma en la variable `componente` tenemos los métodos y atributos que podemos verificar, como vemos en el ejemplo:

```
1 import { TestBed, ComponentFixture } from '@angular/core/testing';
2
3 beforeEach(() => {
4   TestBed.configureTestingModule({declarations: [MyComponent],
5     schemas: [NO_ERRORS_SCHEMA],
6     imports: [...]}))
7 })
8 it('should...', () => {
9   let fixture: ComponentFixture<MyComponent> =
10     TestBed.createComponent(MyComponent);
```

```
11 let component = fixture.debugElement.componentInstance;
12 component.method();
13 expect(component.param).not.toBeNull();
14 })
```

La propiedad `NO_ERRORS_SCHEMA` se utiliza para indicar que no queremos que salte error si Angular detecta que no está declarado algún componente que estemos usando en el template como pipes o web components de terceros.

Test de componente con dependencias

En los componentes conviene hacer fake de las dependencias, que generalmente serán servicios con asincronía, de forma que eliminamos esa asincronía, dado que lo que queremos probar es el comportamiento del componente, no nos importa la forma de obtener los datos.

```
1 import { TestBed, ComponentFixture } from '@angular/core/testing';
2
3 beforeEach(() => {
4   TestBed.configureTestingModule({declarations: [MyComponent],
5     providers: [{provide: MyService, useClass: MyServiceFake}],
6     imports: [...]});
7 })
8 it('should...', () => {
9   let fixture: ComponentFixture<MyComponent> =
10     TestBed.createComponent(MyComponent)
11   let component = fixture.debugElement.componentInstance
12   component.method(param); //Llama a MyService.method() del fake
13 })
```

Es importante que el fake no tenga asincronía, que simplemente devuelva los datos en frío cumpliendo la misma signatura que el servicio original.

Esto se facilita mucho cuando trabajamos con Observables, ya que podemos utilizar cualquier método de creación de observable incluyendo datos fake.

Si, por ejemplo, tenemos el siguiente método en una clase proxy:

```
1 getUsers(): Observable<User[]> {  
2     return this.http.get<User[]>('/api/users');  
3 }
```

Entonces podemos crear una clase fake con la misma signatura pero que devuelva un Observable de forma síncrona:

```
1 getUsers(): Observable<User[]> {  
2     return of(  
3         [  
4             {nombre: 'Lucia', apellidos: 'Díez'},  
5             {nombre: 'Mateo', apellidos: 'Aguilera'}  
6         ]  
7     );  
8 }
```

Tests de aceptación con Protractor

En esta sección vamos a ver cómo configurar y utilizar Protractor para la implementación de los test de aceptación.

La principal característica de estos tests es que necesitan que el código a probar este 100% desarrollado y desplegado en un servidor, por lo que son tests que se realizan en fase de QA generalmente por equipos específicos de pruebas.

Estos tests simulan de forma automática la navegación e interacciones del usuario con la aplicación y chequean si el comportamiento de la aplicación es el adecuado.

Son tests muy frágiles ya que cualquier pequeño cambio en la aplicación obliga a su reimplementación, es por eso que no es una buena práctica abusar de ellos y se recomienda su uso para testear dos o tres procesos críticos de la aplicación a fin de saber de forma automática si la aplicación recién desplegada no tiene ningún problema de regresión. A estos tests también se les conoce como smoke test, o test de humo.

Pueden ejecutarse una y otra vez en distintos entornos de aplicación para asegurar el correcto funcionamiento de flujos críticos de negocio, reduciendo de forma drástica el tiempo de verificación y haciendo que la idea de negocio esté cuanto antes en producción con las máximas garantías.

Protractor

Es la tecnología actualmente más utilizada para la implementación de tests de aceptación o e2e en Angular, es decir aquellos que prueban un flujo completo de negocio, donde es posible que se realice una navegación todo lo compleja que se requiera.

Se puede decir que complementa la sintaxis de Jasmine para ofrecernos una serie de funciones que nos permiten evaluar el estado del DOM de los navegadores en base a acciones que podemos ir provocando en el código.

Se integra con Selenium para levantar los navegadores que configuremos y ejecutar los tests implementados.

Se distribuye en forma de paquete npm que podemos instalar con el comando `install` el cual nos va a instalar por un lado la herramienta `protractor` y por otro la herramienta `webdriver-manager` que le permite interactuar con los distintos navegadores gracias a Selenium, y que podemos dejar configurado con el comando `update`.

Esto descarga todos los binarios necesarios para su correcta ejecución, para levantar una instancia y dejar corriendo la herramienta utilizamos el comando `start`.

Si hacemos uso de `angular-cli` no nos tenemos que preocupar de nada de esto y basta con ejecutar:

```
1 $> npm run e2e
```

Si se quiere hacer algún cambio de configuración como: cambiar el base URL contra el que ejecutar los tests, añadir otro navegador, ... se tiene que modificar el fichero `e2e.conf.js`

Protractor nos proporciona en su [documentación oficial](#)¹⁷ un API muy completo para poder inspeccionar e interactuar con el DOM de la página.

Aquí vemos los elementos del API más comunes.

- **browser:** nos permite trabajar con el navegador y poder indicar la URL que tiene que cargar el navegador, consultar el título de la página o preguntar si un elemento está presente en la página, entre muchas otras cosas.
- **locator:** es el elemento que nos permite apuntar a un determinado elemento del DOM. Podemos hacerlo utilizando el id con `by.Id`, el nombre con `by.name`, el nombre de la clase con `by.className`, y así con otros muchos
- **element:** nos permite pasarle un elemento locator para poder quedarnos con la referencia del elemento del DOM, y por ejemplo, si se trata de un input poder rellenar el campo con el método `sendKeys()`.
- **element.all:** donde si el locator que le pasamos devuelve más de un elemento del DOM, por ejemplo, cuando consultamos por css, podemos recuperarlos a través de la función `then` de la promesa que devuelven.

Este podría ser un ejemplo de caso de test con Protractor:

¹⁷<http://www.protractortest.org/#/api>


```
1  it ('should...', () => {
2
3    browser.get('http://myurl.com')
4
5    element(by.id('oneInput')).sendKeys('test')
6
7    element.all(by.css('.items li')).then(function(items){
8      expect(items.length).toBe(3)
9    })
10
11 })
```

Aconsejo ver la documentación oficial para ver todo el potencial de este API

Ya hemos visto un ejemplo de cómo utilizar el API de Protractor dentro de un caso de test. Como has podido ver el código por si mismo no es muy descriptivo y puede ser que muchos elementos se repitan.

Es por eso que como buena práctica de la guía de estilo se recomienda utilizar el patrón **Page Object**, que simplemente es una clase asociada al test que contiene la forma de acceder a cada uno de los elementos a través de funciones que pretenden ser mucho más descriptivas aportando semántica a nuestros tests.

```
1  export class ExamplePO {
2
3    goToURL(): void {
4      browser.get('http://myurl.com')
5    }
6
7    setInputValue(text:string): void {
8      element(by.id('oneInput')).sendKeys(text)
9    }
10
11 }
```

Ahora en nuestro caso de test solo tenemos que instanciar la clase Page Object correspondiente y hacer uso de los métodos, de esta forma el test queda mucho más legible y mantenible.

```
1 import {ExamplePO} from 'ExamplePO'
2
3 it ('text', () => {
4
5     let examplePO = new ExamplePO();
6     examplePO.goToURL();
7     examplePO.setInputValue('test');
8
9 })
```

Haciendo uso de angular-cli los tests de aceptación se almacenan en el directorio e2e.

Caso práctico

La versión con HttpModule (4.2-) la podéis encontrar como [tutorial en adictosaltrabajo.com](https://www.adictosaltrabajo.com/tutoriales/tests-unitarios-de-integracion-y-de-aceptacion-en-angular-con-jasmine-karma-y-protractor/)¹⁸

Vamos a desarrollar una aplicación que recupere los primeros usuarios de GitHub atacando al API (<https://api.github.com/users>) y por cada uno muestre por pantalla los campos: login, avatar, url y admin. Para ello lo primero que vamos a hacer es crear el componente que se encargará de mostrarlos por pantalla, gracias al angular-cli esto es tan sencillo como ejecutar:

```
1 $> npm run ng -- generate component list-users
```

Esto nos va a crear una carpeta con los ficheros del componente, entre ellos un .spec que como la mayoría no sabe lo que es, tiende a borrarse de forma inmediata, pero a partir de ahora no, verdad ;-)

Ahora pensamos en la solución y, por favor, que a nadie se le ocurra utilizar el servicio HttpClient directamente en el componente. Personalmente, este tipo de aplicaciones las estructuro en tres capas: un servicio de proxy que solo tiene como misión conectar con el API y devolver la respuesta, un servicio de adaptación de la respuesta que viene del servidor al modelo de mi aplicación y el componente que se encarga de visualizar esta información por pantalla.

Por tanto creamos el servicio de proxy que inicialmente, como no va a ser utilizado por nadie más, lo vamos a incluir dentro de la carpeta list-users. Para ello ejecutamos:

¹⁸<https://www.adictosaltrabajo.com/tutoriales/tests-unitarios-de-integracion-y-de-aceptacion-en-angular-con-jasmine-karma-y-protractor/>

```
1 $> npm run ng -- generate service list-users/list-users-proxy
```

Para la implementación vamos a hacer uso del servicio HttpClient y vamos a devolver toda la respuesta en vez de solo el json con los datos, ya que no tiene porque coincidir los datos recuperados con el modelo de la aplicación.

```
1 import { HttpClient, HttpResponse } from '@angular/common/http';
2 import { Injectable } from '@angular/core';
3 import { Observable } from 'rxjs/Observable';
4
5 @Injectable()
6 export class ListUsersProxyService {
7
8   constructor(private httpClient: HttpClient) { }
9
10  getUsers(): Observable<HttpResponse<any>> {
11    return this.httpClient.get<any>('https://api.github.com/users', {observe: 'r\
12 esponse'}));
13  }
14
15 }
```

Fíjate que el método devuelve un Observable con el HttpResponse del servicio y no hace ni debe hacer más lógica que ésta. Ahora lo único que queremos verificar es que la llamada física se está haciendo correctamente, por lo tanto, tenemos que implementar un **test de integración** que lo verifique y no tiene sentido que hagamos un test unitario de esta parte. Así que en el fichero .spec asociado a este servicio de proxy vamos a realizar y verificar la llamada de esta forma:

```
1 import { HttpClientModule } from '@angular/common/http';
2 import { TestBed, inject, async } from '@angular/core/testing';
3 import { ListUsersProxyService } from './list-users-proxy.service';
4
5 describe('ListUsersProxyServiceIT', () => {
6   beforeEach(() => {
7     TestBed.configureTestingModule({
8       imports: [HttpClientModule],
9       providers: [ListUsersProxyService]
10    });
11  });
12
13  it('should be created', inject([ListUsersProxyService], (service: ListUsersPro\
```

```
14 xyService) => {
15     expect(service).toBeTruthy();
16 });
17
18 fit ('should get users from server', async(() => {
19     const proxy: ListUsersProxyService = TestBed.get(ListUsersProxyService);
20     proxy.getUsers().subscribe(
21         response => {
22             expect(response.body).not.toBeNull();
23         }
24     );
25 });
26
27 });
```

Te habrás dado cuenta de que el 80% de este código ya nos lo había proporcionado Angular y que nuestra labor como desarrolladores “solo” se limita a configurar la clase TestBed con todas las dependencias necesarias, en este caso, la importación de HttpClientModule, porque estamos usando el servicio Http, y subscribirnos a la llamada para verificar el resultado. En este tipo de tests no hay que ser muy específico en los expects ya que los datos de la llamada pueden variar con frecuencia.

Podemos ejecutar el comando de test para verificar que efectivamente el test pasa;

```
1 $> npm run test -- --code-coverage
```

pero cuidado porque si olvidas el “async” que envuelve la función puedes estar incurriendo en un falso positivo dado que la parte asíncrona del test no se estará ejecutando. Para comprobar esto te aconsejo que pongas un console.log y veas que realmente se muestra en la consola.

Tienes que tener en cuenta que una de las cosas que más complica este tipo de tests es la asincronía así que el truco está en eliminar esta asincronía en el resto de tests para lo cual vamos a crear un fake del servicio de proxy. Para crear el fake tenemos que tener en cuenta que cumpla con la misma signatura de la función que estamos utilizando, esto es, que devuelva una Observable de tipo HttpResponse, pero lo que no hacemos es inyectar el servicio HttpClient sino que creamos un Observable síncrono con los datos de la respuesta real, la cual establecemos como constante en un fichero llamado “list-users.fake.spec.ts” (dejamos la extensión .spec para que no se incluya en el código de producción), con el siguiente contenido:

```
1 export const LIST_USERS_FAKE = [  
2   {  
3     'login': 'mojombo',  
4     'id': 1,  
5     'avatar_url': 'https://avatars3.githubusercontent.com/u/1?v=3',  
6     'gravatar_id': '',  
7     'url': 'https://api.github.com/users/mojombo',  
8     'html_url': 'https://github.com/mojombo',  
9     'followers_url': 'https://api.github.com/users/mojombo/followers',  
10    'following_url': 'https://api.github.com/users/mojombo/following{/other_user}',  
11  },  
12    'gists_url': 'https://api.github.com/users/mojombo/gists{/gist_id}',  
13    'starred_url': 'https://api.github.com/users/mojombo/starred{/owner}/{/repo}',  
14    'subscriptions_url': 'https://api.github.com/users/mojombo/subscriptions',  
15    'organizations_url': 'https://api.github.com/users/mojombo/orgs',  
16    'repos_url': 'https://api.github.com/users/mojombo/repos',  
17    'events_url': 'https://api.github.com/users/mojombo/events{/privacy}',  
18    'received_events_url': 'https://api.github.com/users/mojombo/received_events',  
19  },  
20    'type': 'User',  
21    'site_admin': false  
22  },  
23  {  
24    'login': 'defunkt',  
25    'id': 2,  
26    'avatar_url': 'https://avatars3.githubusercontent.com/u/2?v=3',  
27    'gravatar_id': '',  
28    'url': 'https://api.github.com/users/defunkt',  
29    'html_url': 'https://github.com/defunkt',  
30    'followers_url': 'https://api.github.com/users/defunkt/followers',  
31    'following_url': 'https://api.github.com/users/defunkt/following{/other_user}',  
32  },  
33    'gists_url': 'https://api.github.com/users/defunkt/gists{/gist_id}',  
34    'starred_url': 'https://api.github.com/users/defunkt/starred{/owner}/{/repo}',  
35    'subscriptions_url': 'https://api.github.com/users/defunkt/subscriptions',  
36    'organizations_url': 'https://api.github.com/users/defunkt/orgs',  
37    'repos_url': 'https://api.github.com/users/defunkt/repos',  
38    'events_url': 'https://api.github.com/users/defunkt/events{/privacy}',  
39    'received_events_url': 'https://api.github.com/users/defunkt/received_events',  
40  },  
41    'type': 'User',  
42    'site_admin': true  
43  ]
```

```
43   }  
44 ]`
```

Y ahora lo usamos en el fake “list-users-proxy.service.fake.spec.ts” con el siguiente contenido:

```
1  import { LIST_USERS_FAKE } from './list-users.fake.spec';  
2  import { HttpResponse } from '@angular/common/http';  
3  import { Observable } from 'rxjs/Observable';  
4  import { of } from 'rxjs/observable/of';  
5  
6  export class ListUsersProxyServiceFake {  
7  
8    constructor() { }  
9  
10   getUsers(): Observable<HttpResponse<any>> {  
11     const httpResponse = new HttpResponse<any>(  
12       {body: LIST_USERS_FAKE}  
13     );  
14     return of(httpResponse);  
15   }  
16  
17 }
```

Es el momento de crear nuestro modelo. En este caso nos podemos plantear si construir el modelo con una interfaz o con una clase. La diferencia reside en que con la interfaz no añades más código a la aplicación, es simplemente para que el IDE te pueda autocompletar este tipo de datos; mientras que con la clase sí estás añadiendo código real y no es tan flexible a la hora de inicializar los datos a través del constructor.

A mí últimamente me gusta más hacerlo con interfaces así que la podemos crear con el siguiente comando:

```
1 $> npm run ng -- generate interface list-users/user
```

Con el siguiente contenido:

```
1 export interface User {  
2     login: string;  
3     avatar: string;  
4     url: string;  
5     admin: boolean;  
6 }
```

El siguiente paso es crear el servicio que adapta los datos de la respuesta al modelo. Para ello ejecutamos:

```
1 $> npm run ng -- generate service list-users/list-users
```

Este servicio va a inyectar el proxy y gracias a la programación reactiva con la función `.map` que ofrece la librería `rxjs` podemos fácilmente hacer el mapeo entre los campos de la respuesta y el modelo de nuestro negocio que, como es nuestro caso, no tienen por qué coincidir y nos permite desacoplarnos de la respuesta y dar un sentido semántico al desarrollo que facilita la legibilidad y el mantenimiento de la aplicación. El contenido de este servicio es el siguiente:

```
1 import { map } from 'rxjs/operators/map';  
2 import { ListUsersProxyService } from './list-users-proxy.service';  
3 import { User } from './user';  
4 import { Injectable } from '@angular/core';  
5 import { Observable } from 'rxjs/Observable';  
6  
7  
8 @Injectable()  
9 export class ListUsersService {  
10  
11     constructor(private proxy: ListUsersProxyService) { }  
12  
13     getUsers(): Observable<User[]> {  
14         return this.proxy.getUsers().pipe(  
15             map(  
16                 (response) => {  
17                     const listUsers: User[] = [];  
18                     const data = response.body;  
19                     data.forEach(d => {  
20                         const user: User = {  
21                             login: d.login,  
22                             avatar: d.avatar_url,  
23                             url: d.url,
```

```
24         admin: d.site_admin
25     };
26     //listUsers.push(user);
27     listUsers = [...listUsers, user];
28 });
29     return listUsers;
30 }
31 );
32 );
33 }
34
35 }
```

Fíjate que donde antes lo normal sería utilizar el método push para añadir el elemento al array; ahora hacemos uso de spread parameter (...) consiguiendo que el resultado sea inmutable. Esto es muy importante a la hora de poder simplificar el change detection y ganar en rendimiento.

Ahora vamos a configurar e implementar el test asociado que como vamos a utilizar el fake será un test unitario no haciendo falta la implementación de un test de integración. Este es el contenido del test:

```
1  import { ListUsersProxyService } from './list-users-proxy.service';
2  import { ListUsersProxyServiceFake } from './list-users-proxy.service.fake.spec';
3  import { ListUsersService } from './list-users.service';
4  import { inject, TestBed } from '@angular/core/testing';
5
6
7  describe('ListUsersService', () => {
8      beforeEach(() => {
9          TestBed.configureTestingModule({
10             providers: [
11                 {
12                     provide: ListUsersService,
13                     useClass: ListUsersService,
14                     deps: [
15                         {provide: ListUsersProxyService, useClass: ListUsersProxyServiceFake}
16                     ]
17                 }
18             ]
19         });
20     });
```



```
21
22   it('should be created', inject([ListUsersService], (service: ListUsersService)\
23   => {
24       expect(service).toBeTruthy();
25   }));
26
27   it('should get users', () => {
28       const service: ListUsersService = TestBed.get(ListUsersService);
29       service.getUsers().subscribe(
30           (users) => {
31               expect(users[0].login).toEqual('mojombo');
32               expect(users[0].avatar).toBeDefined();
33           }
34       );
35   });
36 });
```

Fíjate que la clave está en la definición del provider donde establecemos que la implementación la proporcione el fake creado, de esta forma no necesitamos la función `async` y podemos ser más específicos en los expects dado que esta respuesta sí la estamos controlando, y solo queremos verificar que el mapeo de campos se está haciendo de forma adecuada. Ejecutamos el comando de test y vemos que todos los tests van pasando y que tenemos un buen grado de cobertura.

Teniendo ya los servicios implementados y probados, es el momento de implementar nuestro componente. El cuál dentro del método `ngOnInit` va a establecer el valor del atributo “users” que será un observable ya que vamos a hacer uso del pipe “`async`” para aprovechar la desuscripción automática.

```
1  import { Observable } from 'rxjs/Observable';
2  import { User } from './user';
3  import { ListUsersService } from './list-users.service';
4  import { Component, OnInit } from '@angular/core';
5
6  @Component({
7      selector: 'app-list-users',
8      templateUrl: './list-users.component.html',
9      styleUrls: ['./list-users.component.css']
10 })
11 export class ListUsersComponent implements OnInit {
12
13     users$: Observable<User[]>;
14 }
```

```

15   constructor(private service: ListUsersService) { }
16
17   ngOnInit() {
18     this.users$ = this.service.getUsers();
19   }
20
21 }

```

Y en el template podemos establecer el siguiente contenido atendiendo a poner los ids adecuados que faciliten los tests de aceptación. Un posible contenido (sin mucho estilo, aquí es donde digo que entrarían los diseñadores con sus componentes de Polymer o StencilJS supercurrados donde el desarrollador solo tiene que pasarle una estructura de datos definida para que los datos se pinten de forma corporativa y mucho más bonita) podría ser éste:

```

1 <div id="user-{{i}}"
2   *ngFor="let user of (users$ | async); let i=index">
3   <p>{{user.login}}</p>
4   <p>{{user.url}}</p>
5   <p>{{user.admin}}</p>
6   <img [src]="user.avatar" alt="avatar" width="200">
7 </div>

```

Ahora implementamos el test asociado donde es muy importante que lo limitemos a verificar que los atributos del componente se establecen adecuadamente y no empecemos a liarnos a verificar elementos del DOM que van a hacer que nuestro test sea mucho más frágil. El contenido del test unitario sería éste:

```

1 import { async, ComponentFixture, TestBed } from '@angular/core/testing';
2 import { ListUsersComponent } from './list-users.component';
3 import { ListUsersProxyService } from './list-users-proxy.service';
4 import { ListUsersProxyServiceFake } from './list-users-proxy.service.fake.spec';
5 import { ListUsersService } from './list-users.service';
6 import { NO_ERRORS_SCHEMA } from '@angular/core';
7
8
9 describe('ListUsersComponent', () => {
10   let component: ListUsersComponent;
11   let fixture: ComponentFixture<ListUsersComponent>;
12
13   beforeEach(async(() => {
14     TestBed.configureTestingModule({

```

```
15     declarations: [ ListUsersComponent ],
16     providers: [
17         ListUsersService,
18         {provide: ListUsersProxyService, useClass: ListUsersProxyServiceFake}
19     ],
20     schemas: [NO_ERRORS_SCHEMA]
21 })
22 .compileComponents();
23 }));
24
25 beforeEach(() => {
26     fixture = TestBed.createComponent(ListUsersComponent);
27     component = fixture.componentInstance;
28     fixture.detectChanges();
29 });
30
31 it('should be created', () => {
32     expect(component).toBeTruthy();
33 });
34
35 it('should set users', () => {
36     component.ngOnInit();
37     component.users$.subscribe(
38         users => {
39             expect(users[0].login).toEqual('mojombo');
40         }
41     );
42 });
43 });
```

Fíjate como nos subscribimos al observable para verificar que efectivamente hemos recuperado bien en el componente la lista de usuarios que vienen del fake.

La propia implementación por defecto ya nos ofrece las instancias de fixture (para comprobar el DOM) y component que no es más que la instancia del componente que nos permite llamar a los métodos y verificar los atributos.

En este momento nuestra aplicación está implementada y probada. Ahora cualquier cambio no nos dará pánico porque habrá un test que nos dirá si estamos rompiendo funcionalidad y estaremos mucho más confiados a la hora de poner nuestros desarrollos en producción. Recuerda más vale 10 subidas pequeñas al día controladas que una cada

3 meses con un montón de funcionalidad que no da tiempo a verificar en el momento de pasar a producción.

Ahora podemos configurar apropiadamente nuestro fichero “app.module.ts” con el siguiente contenido:

```
1  import { ListUsersProxyService } from './list-users/list-users-proxy.service';
2  import { ListUsersService } from './list-users/list-users.service';
3  import { HttpClientModule } from '@angular/common/http';
4  import { BrowserModule } from '@angular/platform-browser';
5  import { NgModule } from '@angular/core';
6
7  import { AppComponent } from './app.component';
8  import { ListUsersComponent } from './list-users/list-users.component';
9
10 @NgModule({
11   declarations: [
12     AppComponent,
13     ListUsersComponent
14   ],
15   imports: [
16     BrowserModule,
17     HttpClientModule
18   ],
19   providers: [ListUsersService, ListUsersProxyService],
20   bootstrap: [AppComponent]
21 })
22 export class AppModule { }
```

Utilizamos nuestro componente dentro de app.component.html que es el componente principal de nuestra aplicación, sustituyendo todo el contenido que viene por defecto, por éste:

```
1  <app-list-users></app-list-users>
```

Y ahora solo tenemos que arrancar el proyecto:

```
1  $> npm run start
```

Y verificar la salida en el puerto `http://localhost:4200`

Tendrá que salir algo parecido a esto:

mojombo

<https://api.github.com/users/mojombo>

false



defunkt

<https://api.github.com/users/defunkt>

true



pjhyett

<https://api.github.com/users/pjhyett>

A partir de tener la aplicación desarrollada y en funcionamiento, nos podemos plantear realizar los tests de aceptación.

Angular almacena los tests de aceptación en la carpeta e2e y maneja el patrón Page Object donde tenemos un fichero .po. que almacena las funciones de acceso al DOM y otros .spec que implementan los tests haciendo uso de los .po.

De este modo lo primero es crear el fichero “list-users.po.ts” dentro de la carpeta e2e con el siguiente contenido, donde a través de los elementos de protractor nos quedamos con la instancia del DOM del primer usuario a través del id que le hemos puesto.

```
1 import { browser, by, element } from 'protractor';
2
3 export class ListUsersPage {
4   navigateTo() {
5     return browser.get('/');
6   }
7
8   getFirstUser() {
9     return element(by.id('user-0'));
10  }
11
12 }
```

Ahora creamos el fichero “list-users.spec.ts” donde hacemos el flujo de cargar la aplicación y verificar que el primer usuario es ‘mojombo’.

```
1 import { browser } from 'protractor';
2 import { ListUsersPage } from '../pos/list-users.po';
3
4 describe('nglabs List Users', () => {
5   let page: ListUsersPage;
6
7   beforeEach(() => {
8     page = new ListUsersPage();
9   });
10
11   it('should check first user is mojombo', () => {
12     page.navigateTo();
13     page.getFirstUser().getText().then(
14       text => {
15         expect(text).toContain('mojombo');
16       }
17     );
18   });
19 });
```

```
17     );
18     expect(page.getFirstUser().getText()).toContain('mojombo');
19   });
20
21   afterEach(() => {
22     browser.driver.sleep(3000); //Esto es solo para que nuestro ojo humano pueda\
23     ver el resultado en el navegador
24   });
25
26 });
```

Ahora podemos ejecutar estos tests con el comando:

```
1 $> npm run e2e
```

Obviamente este tipo de tests son los más frágiles pero sí que nos valen para registrar los flujos más críticos de nuestra aplicación y lanzarlos como “smoke tests” en cualquier entorno para verificar que una subida a producción se ha hecho de forma satisfactoria, por ejemplo. Esto es mucho más rápido y efectivo que 10 equipos quedando a las 4.00 am para subir a producción y de 4.05 am a varias horas después están verificando manualmente que no han roto nada (esto lo he vivido en cliente); cuando con este tipo de tests en cuestión de minutos y de forma automática está verificado y si fallan se puede configurar para hacer un rollback a la versión anterior.

Es posible que tengáis que adaptar los ficheros de tests que vienen con la configuración inicial, simplemente borrad todos aquellos casos de tests que ya no tengan validez.

Conclusión

Como ves no es tan complicado hacer las cosas bien y vivir mucho más tranquilos dejando el temor de pasar a producción y ayudando a que el mantenimiento sea mucho menos costoso. Pudiendo hacer realidad las fantasías de cualquier persona de negocio de ver su idea en producción, realmente, lo antes posible.

Integraciones

Integración con PrimeNG

PrimeNG es una librería que a día de hoy proporciona más de 70 componentes empresariales (grids, botones, gráficos, ...) listo para usar dentro de nuestras aplicaciones de Angular.

Es la integración perfecta cuando lo que queremos es crear un backoffice del que no nos queremos preocupar mucho por la apariencia, aunque está se puede modificar en base a colores base definidos en los temas, y permite personalizar tu propio tema.

Instalación de PrimeNG en angular-cli

Si tenemos un proyecto creado con angular-cli la forma más sencilla de instalar PrimeNG para poder hacer uso de todos los componentes es a través de NPM:

```
1 $> npm install --save primeng
```

Esta librería hace uso de font-awesome así que tenemos que instalarla de esta forma:

```
1 $> npm install --save font-awesome
```

Luego editamos el fichero .angular-cli.json para añadir estas líneas dentro de la sección “styles” respetando las existentes.

```
1 "styles": [  
2   "styles.css",  
3   "../node_modules/primeng/resources/themes/afterwork/theme.css",  
4   "../node_modules/primeng/resources/primeng.min.css",  
5   "../node_modules/font-awesome/css/font-awesome.min.css"  
6 ]
```

Si te fijas en la estructura de carpetas dentro de la carpeta themes del módulo de primeng tienes todos los temas disponibles y listos para utilizar, simplemente tienes que modificar la ruta y en vez de “afterwork” establecer “omega”, por ejemplo.

Utilización de la librería

Hecho esto podemos consultar la [documentación](http://www.primefaces.org/primeng/)¹⁹ de PrimeNG para ver cómo podemos hacer uso de sus componentes.

En general todos ellos están estructurados en módulos de forma que habrá que declarar el import correspondiente en el módulo en el que queramos hacer uso del componente. Este sería el ejemplo de declaración del uso del componente Button dentro del fichero `app.module.ts`

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { FormsModule } from '@angular/forms';
4  import { HttpClientModule } from '@angular/http';
5  import { AppComponent } from './app.component';
6  import { ButtonModule } from 'primeng/primeng';
7
8  @NgModule({
9    declarations: [
10      AppComponent
11    ],
12    imports: [
13      BrowserModule,
14      FormsModule,
15      HttpClientModule,
16      ButtonModule
17    ],
18    providers: [],
19    bootstrap: [AppComponent]
20  })
21
22  export class AppModule { }
```

De esta forma en cualquier template de un componente ligado con este módulo podemos hacer uso del componente Button, por ejemplo, de esta forma, en el fichero `app.component.html`:

¹⁹<http://www.primefaces.org/primeng/#/>

```
1 <div>
2   {{title}}
3   <button pButton type="button" label="Click"></button>
4 </div>
```

Integración con Polymer 2.0

Esta es la otra gran alternativa de Google que permite trabajar con Web Components, de hecho parece que hay una guerra fratricida entre ellas por ver quién se queda con el pastel completo del desarrollo de aplicaciones del lado del cliente.

En mi opinión, es una guerra que las dos tienen completamente pérdida porque cada una es la mejor en su campo: Angular como framework que facilita la creación de aplicaciones y Polymer como librería web que facilita la creación de componentes web reutilizables.

Juegan en ligas diferentes, me explico:

Polymer está conceptualmente por encima de Angular, y es la tecnología que dentro de una empresa deberían utilizar los arquitectos y diseñadores para crear librerías de componentes reutilizables que poder ofrecer a sus desarrolladores, para construir cualquier tipo de aplicación web.

Estos roles deberían centrarse en aportar riqueza visual y reutilización a los componentes por lo que una de las premisas fundamentales es que deben abstraerse del negocio, trabajando solo con estructuras de datos y aplicando patrones de diseño para favorecer la reutilización de los componentes, creando etiquetas HTML que los desarrolladores puedan utilizar fácilmente para aportar riqueza visual a las aplicaciones manteniendo un estilo corporativo único y homogéneo.

Por contra, conceptualmente es mucho más costoso (no digo que no se pueda) crear una aplicación completa donde hay que intercomunicar todos los componentes y manejar conceptos como el routing, las validaciones de formularios, la conexión con servidores remotos, ... de una manera mucho menos clara y mantenible que con Angular.

Angular es un framework mucho más orientado a desarrolladores que tienen que dar una respuesta casi inmediata a los requisitos de negocio.

Con Angular es muy rápido crear el prototipo de una aplicación pero no tiene ninguna característica que favorezca la reutilización de componentes y la riqueza visual de la aplicación.

Por otro lado proporciona inyección de dependencias que facilita el testing general de la aplicación y maneja el routing, el data binding, las validaciones de los formularios y las conexiones con servidores a través de HTTP de una manera mucho más sencilla que Polymer.

Entonces, parece obvio, que el caballo ganador es dejar que los desarrolladores creen las aplicaciones sin perder tiempo en el aspecto visual y centrándose en el aspecto funcional dando solución a los requisitos de negocio; mientras los arquitectos y diseñadores facilitan componentes reutilizables que permitan a los desarrolladores, simplemente con la utilización de ciertas etiquetas, dar el estilo visual que Negocio requiera.

Nos ponemos la gorra de arquitectos y diseñadores

Como arquitectos y diseñadores solo tenemos que preocuparnos por la reutilización y el diseño visual, que no es poco. Así que tenemos que aprender a fondo el funcionamiento de Polymer.

Si recurrimos a la documentación oficial nos damos cuenta de que está completamente orientada a la creación de aplicaciones, incluso te facilitan una herramienta que te crea una aplicación de forma rápida, pero a poco que entras en ver cómo está configurada te das cuenta de que no entiendes ni la mitad de las cosas que hace.

En mi opinión esto es un error ya que lo que deberían facilitar es una herramienta orientada a la creación de librerías de componentes reutilizables; es por eso que me decidí a crear un generador de Yeoman con lo básico para la creación de una librería de componentes reutilizables con Polymer, que podéis instalar de esta forma:

```
1 $> npm install -g yo
2 $> npm install -g generator-polymer-lib
3 $> mkdir nombre-empresa-lib && cd nombre-empresa-lib
4 $> yo polymer-lib
```

La estructura del proyecto es muy sencilla, todos los componentes tanto propios como de Polymer que queramos compartir con nuestros desarrolladores están alojados en el fichero lib/lib.html y en el fichero index.html tendremos el banco de pruebas con la documentación de cada uno de los componentes. El proyecto está gestionado con Bower que es el gestor de dependencias que utiliza Polymer actualmente.

Por lo que cualquier dependencia la podemos instalar:

```
1 $> bower install --save PolymerElements/app-layout
```

Nos va crear toda la configuración necesaria y un par de componentes de ejemplo, que podréis visualizar ejecutando:

```
1 $> npm run live
```

En este punto podemos modificar el proyecto para añadir todos los componentes y dependencias que queramos a nuestro proyecto utilizando Polymer. La anatomía básica de un elemento propio con Polymer es esta:

```
1 <dom-module id="tnt-hello">
2   <template>
3     <style>
4       :host {
5         display: block;
6       }
7     </style>
8
9     <h1 on-click="select">Hello [[name]]</h1>
10
11   </template>
12   <script>
13     class HelloElement extends Polymer.Element {
14       static get is() {return "tnt-hello";}
15
16       static get properties() {
17         return {
18           name: {
19             type: String,
20             value: "by default"
21           }
22         }
23       }
24
25       select() {
26         this.dispatchEvent(new CustomEvent('select', {detail: 'Message'}));
27       }
28
29     }
30     customElements.define(HelloElement.is, HelloElement);
31   </script>
32 </dom-module>
```

Para probar en fase de desarrollo este componente podemos importarlo y hacer uso de la etiqueta en el fichero index.html del proyecto, quedando de esta forma:

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5    <meta charset="utf-8">
6    <title>Polymer Lib</title>
7    <script src="bower_components/webcomponentsjs/webcomponents-loader.js"></scrip\
8  t>
9    <link rel="import" href="lib/lib.html" />
10 </head>
11
12 <body>
13
14   <h1>Showcase components</h1>
15
16   <tnt-hello name="All"></tnt-hello>
17
18   <script>
19     document.querySelector('tnt-hello').addEventListener('select', function ($ev\
20 ent) {
21     alert($event.detail);
22     })
23   </script>
24
25 </body>
26
27 </html>
```

Una vez tengamos la librería a nuestro gusto (o al gusto de negocio) la preparamos para producción con el comando:

```
1  $> npm run pro
```

Este comando pasa un proceso de vulcanizado creando un único fichero bundle.html en la carpeta “dist” con todos los componentes propios y de terceros necesarios.

Ahora para poder compartir la librería con nuestros desarrolladores necesitamos hacerlo a través de un repositorio npm privado, yo aconsejo utilizar la última versión de Nexus OSS que nos permite tener rápidamente un repositorio de npm privado y corporativo; también tiene repositorios para Bower, NuGet e incluso Docker.

En el fichero package.json del proyecto tenemos un script npm llamado “publish” donde pondremos la URL apuntando a nuestro repositorio npm privado.

Hecho esto, y con las credenciales necesarias, basta con ejecutar:

```
1 $> npm run publish
```

Si todo es correcto tendremos en el repositorio privado de npm la librería disponible para el resto de proyectos.

Nos ponemos la gorra de desarrolladores

Como desarrolladores nos gustaría poder centrarnos en la lógica de la aplicación encomendada y no tener que pasar horas moviendo arriba y abajo el CSS porque salvo honrosas excepciones un desarrollador no tiene mucho gusto estético.

Para integrar la librería anterior en nuestra aplicación solo tenemos que ejecutar:

```
1 $> npm install --save nombre-empresa-lib
```

Esto instalará la librería en la carpeta “node_modules” donde dentro del directorio “dist” encontraremos dos ficheros: webcomponents-loader.js (polyfill para el uso de Web Components en navegadores que no lo soporten) y el bundle.html con nuestra librería de componentes reutilizables.

Si estamos usando angular-cli, lo mejor es que editemos el fichero .angular-cli.json y en la sección “scripts” añadamos la ruta relativa al fichero webcomponents-loader.js.

En el caso de bundle.html, lo tenemos que copiar en la carpeta src/assets y refenciarlo en el index.html de esta forma:

```
1 <link rel="import" href="assets/bundle.html">
```

Ahora podemos hacer uso de los componentes definidos en la librería tanto en el propio index.html como en el template de cualquier elemento de Angular, pudiendo pasar el valor de los atributos a los distintos inputs de los componentes de Polymer.

Integración con StencilJS

StencilJS es un compilador que genera Web Components nativa 100% compatibles con el estándar y que por tanto lo hacen 100% compatibles con cualquier librería y framework web sin necesidad de ejecutarse en un contexto determinado.

Esta tecnología juega en la liga de Polymer y es muy posible que llegue a sustituirla dada su facilidad gracias a que soporta TypeScript de serie y su gran rendimiento haciendo uso del virtual DOM de React.

De hecho verás que la sintaxis es una mezcla entre Angular y React, aunque también es compatible con Vue, Backbone, o cualquier tecnología web en la que te quieras apoyar.

Por ponerle una pega actualmente no tiene un catálogo de componentes ya desarrollados aunque en poco tiempo seguro que esto dejará de ser un problema.

Como Polymer, StencilJS es la tecnología ideal para crear librerías de componentes reutilizables y para los desarrolladores de Angular tiene la ventaja de que la sintaxis y el ecosistema son más parecidos que con Polymer.

Detrás de esta tecnología están los chicos de Ionic y ya han anunciado que para Ionic 4 todos los componentes de su core estarán implementados con StencilJS lo que los hará agnósticos al framework.

Empezando con StencilJS

Un buen punto de partida para empezar con esta tecnología es la propia documentación oficial que podemos encontrar [aquí](https://stenciljs.com/docs/getting-started)²⁰

Esta documentación nos ofrece un proyecto starter para tener ya todo configurado que podemos utilizar ejecutando:

```
1 $> git clone https://github.com/ionic-team/stencil-starter.git ng-stencil
```

A continuación entramos dentro de la carpeta generada:

```
1 $> cd ng-stencil
```

Eliminamos el remote origin original:

```
1 $> git remote rm origin
```

Instalamos todas las dependencias necesarias:

```
1 $> yarn o npm install
```

Y directamente podemos ejecutar el proyecto de prueba con:

²⁰<https://stenciljs.com/docs/getting-started>

```
1 $> npm run start
```

En la URL `http://localhost:3333` podremos ver algo parecido a esto:



Creación del primer componente

Para nuestro primer componente con StencilJS vamos a replicar el componente `tnt-hello` que implementamos con Polymer y de paso vemos las diferencias de código.

El componente `tnt-hello` recibe un parámetro de entrada “name” y envía un evento con la palabra ‘Message’ cuando se pulsa sobre él, que puede ser recogido por cualquier framework o directamente con VanillaJS en el `index.html`.

Entonces creamos la carpeta `tnt-hello` y dentro dos ficheros: el primero `tnt-hello.scss` para darle “estilo” al componente, con el siguiente contenido:

```
1 tnt-hello {  
2   color: blue;  
3 }
```

y el segundo `tnt-hello.tsx` que contendrá la lógica y el contenido visual, con el siguiente contenido:


```
1  import { Component, Prop, Event, EventEmitter } from '@stencil/core';
2
3
4  @Component({
5    tag: 'tnt-hello',
6    styleUrls: 'tnt-hello.scss'
7  })
8  export class TntHello {
9
10     @Prop() name: string;
11     @Event() select: EventEmitter;
12
13     onSelect(event) {
14       this.select.emit(this.name);
15     }
16
17     render() {
18       return (
19         <h1 onClick={() => this.onSelect()}>Hello {this.name}</h1>
20       );
21     }
22  }
```

Como ves el componente se define con el decorador `@Component` que requiere dos propiedades obligatorias:

- **tag**: exactamente igual al selector en el componente de Angular; define el nombre de la etiqueta.
- **styleUrl**: donde indicamos la localización del fichero que alberga el estilo del componente, que puede ser scss o css.

Como ves en el componente definimos la propiedad de entrada “name” con el decorador `@Prop` de tipo string y el evento de salida “select” con el decorador `@Event` de tipo `EventEmitter`.

Esto es muy parecido al `@Input` y `@Output` que vimos en el tema de Data Binding.

Ahora echarás en falta el `templateUrl` de la definición del componente, esto es porque StencilJS al igual que React hace uso del Virtual DOM por lo que hay que definir un método llamado “render” donde vamos a devolver el contenido html del componente.

Fíjate que el HTML se devuelve directamente entre los paréntesis, sin template string ni nada parecido.

En el contenido HTML estamos mostrando el texto “Hello” seguido del valor de la propiedad “name” entre etiquetas h1. Fíjate que la interpolación de la variable se hace con un solo {}, en contra del {}{} y que tenemos que hacer uso de this para acceder al valor de la propiedad.

Para manejar el evento click sobre el h1, lo definimos con la palabra “onClick” seguida de la llamada al manejador, en este caso el método “onSelect” que simplemente hace uso del EventEmitter “select” para a través del método “emit” lanzar fuera del componente el valor de la propiedad “name” sin preocuparse por quién o quiénes estén escuchando ese evento.

Es el momento de hacer uso de nuestro componente en el proyecto y probar su funcionamiento. Para ello editamos el fichero index.html y justo debajo de la etiqueta “my-name” colocamos nuestro componente pasándole valor en el atributo “name”, de esta forma:

```
1 <my-name first="Stencil" last="JS"></my-name>
2 <tnt-hello name="Rubén Aguilera"></tnt-hello>
```

Además para capturar el evento lanzado cuando pulsemos sobre el componente vamos a hacer uso del método addEventListener() que nos proporciona JavaScript, de esta forma:

```
1 <script>
2   document.querySelector('tnt-hello').addEventListener('select', function (event\
3 nt) {
4     alert(event.detail);
5   })
6 </script>
```

Este sería el resultado con los dos componentes cuando pinchamos en el nuestro:



Como ves esta parte funciona perfectamente y ya tenemos dos componentes listos para usar en producción :-)

Ahora vamos a ver cómo de bien se integran nuestros componentes de StencilJS con mi framework favorito Angular. Para ello editamos el fichero `stencil.config.js` donde añadimos nuestro componente al array de “components” y eliminamos la referencia al router, quedando de este modo:

```
1 exports.config = {
2   bundles: [
3     { components: ['my-name', 'tnt-hello'] }
4   ]
5 };
6
7 exports.devServer = {
8   root: 'www',
9   watchGlob: '**/**'
10 }
```

Con esta configuración estamos indicando que queremos generar un único bundle con los dos componentes. Estos bundles se van a cargar de forma lazy sin hacer nada; la documentación oficial recomienda que cada componente vaya en un bundle independiente, así que lo definimos del siguiente modo:

```
1 exports.config = {
2   bundles: [
3     { components: ['my-name'] },
4     { components: ['tnt-hello'] }
5   ]
6 };
7
8 exports.devServer = {
9   root: 'www',
10  watchGlob: '**/**'
11 }
```

Ahora tenemos que hacer la distribución de nuestra librería. Para ello tenemos que editar el fichero “`stencil.config.js`” y añadir las propiedades: `namespace`, `generateDistribution` y `generateWWW`, quedando el fichero de este modo:

```
1  exports.config = {
2    namespace: '<nombre del namespace que le quieras dar>',
3    generateDistribution: true,
4    generateWWW: true,
5    bundles: [
6      { components: ['tnt-hello'] }
7    ]
8  };
9
10 exports.devServer = {
11   root: 'www',
12   watchGlob: '**/**'
13 }
```

Además en el fichero package.json tenemos que añadir las siguientes referencias (entre “description” y “scripts”):

```
1  ...
2    "main": "dist/collection/index.js",
3    "types": "dist/collection/index.d.ts",
4    "collection": "dist/collection/collection-manifest.json",
5    "files": [
6      "dist/"
7    ],
8    "browser": "dist/<nombre del namespace que le hayas puesto antes>.js",
9  ...
```

Para generar los bundles deseados de producción solo tenemos que ejecutar:

```
1  $> npm run build
```

Esto nos va a crear la carpeta “dist” que va a contener todos los ficheros necesarios para la distribución en producción de nuestros componentes. Por tanto este contenido es el que tenemos que hacer llegar a los proyectos que quieran hacer uso de ellos. La forma correcta es publicando la librería en algún repositorio npm ya sea público o privado, para ello tenemos que logarnos en ese repositorio con “npm login” y ejecutar “npm publish”.

Una vez compartida en el repositorio NPM vamos a hacer uso de ella en un proyecto Angular, así que vamos a crear un nuevo proyecto:

```
1 $> ng new ng-app
```

Para instalar la librería de web components con StencilJS simplemente ejecutamos:

```
1 $> npm install --save nombre-libreria
```

Para que tenga en cuenta los distintos cambios de versión de la librería vamos a registrarla como assets en el fichero `.angular-cli.json` de esta forma:

```
1 ...
2 "assets": [
3   "assets",
4   "favicon.ico",
5   { "glob": "**/*", "input": "../node_modules/nombre-libreria", "output": "../nom\
6 bre-libreria" }
7 ],
8 ...
```

Ahora editamos el fichero `src/index.html` del proyecto de Angular, añadimos la referencia al fichero que contiene la implementación de nuestros web components y hacemos uso del componente `tnt-hello` fuera del ámbito de Angular, de esta forma:

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>Autentia Angular University</title>
6   <base href="/">
7   <meta name="viewport" content="width=device-width, initial-scale=1">
8   <link rel="icon" type="image/x-icon" href="favicon.ico">
9   <script src="nombre-libreria/dist/nombre-namespace-que-le-hayas-dado.js"></scr\
10 ipt>
11 </head>
12 <body>
13   <tnt-hello name="Funciona fuera de Angular"></tnt-hello>
14   <app-root>Loading...</app-root>
15 </body>
16 </html>
```

Para poder hacer uso del componente de Angular, al igual que ocurre cuando hacemos uso de componentes de Polymer, que Angular no entiende porque no tiene registrados, tenemos que editar el fichero `app.module` y añadir la propiedad `"schemas"` con el valor `CUSTOM_ELEMENTS_SCHEMA`:

```
1 @NgModule({
2   declarations: [
3     AppComponent
4   ],
5   imports: [
6     BrowserModule
7   ],
8   schemas: [CUSTOM_ELEMENTS_SCHEMA]
9 })
```

Vamos a incluir el componente dentro del template del componente principal pasándole un texto y recibiendo el texto de vuelta a través del evento. Para ello vamos a editar primero el fichero app.component.ts, dentro del método ngOnInit establemos valor al atributo name y creamos el método manejador del evento select que saltará cuando pinchemos sobre el h1 del componente. El resultado sería este:

```
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent implements OnInit {
9   name: string;
10
11   constructor() { }
12
13   ngOnInit() {
14     this.name = 'Funciona dentro de Angular';
15   }
16
17   onSelect(event) {
18     alert(event.detail);
19   }
20
21 }
```

Ahora solo nos queda editar el fichero app.component.html para hacer uso del componente en el template quedando de esta forma:

```
1 <tnt-hello [name]="name" (select)="onSelect($event)"></tnt-hello>
```

Nota: Fíjate como hacemos uso del “data binding” de Angular, para las propiedades utilizamos el corchete para que se evalúe la variable entre dobles comillas y para los eventos hacemos uso de los paréntesis para declarar el manejador del evento “select” que simplemente mostrará la alerta. Es importante que como cualquier evento con Angular lo denotemos exactamente con \$event sino la información no será transmitida.

Si ejecutas este ejemplo tendrás que ver que en pantalla se muestra el texto de las dos instancias del componente, una dentro del ámbito de Angular y la otra fuera y que solo al pinchar en la de dentro se muestra la alerta con el texto.



Hello Funciona dentro de Angular

Como ves todo funciona a la perfección y nos permite poder dividir los proyectos y que los desarrolladores hagan aplicaciones con Angular sin preocuparse del estilo, simplemente añadiendo las etiquetas que los arquitectos, diseñadores y UX les han facilitado para hacerla vistosa, usable y funcional en un tiempo record, donde cada rol se dedica a lo suyo.

Puesta en producción

Llegados a este punto ya tenemos una versión en desarrollo que queremos poner en producción. Pero antes tenemos que tener algunas cosas en cuenta.

Análisis del tamaño de la aplicación

Un punto crítico en el rendimiento de este tipo de aplicaciones es el tamaño del bundle resultante, ya que el usuario no podrá comenzar a trabajar con la aplicación hasta que no esté descargado en su navegador.

Verifica que solo se incluye lo obligatorio

Una herramienta muy interesante que refleja visualmente el tamaño de los distintos elementos que componen una aplicación es `source-map-explorer` la cual podemos instalar con `npm`:

```
1 $> npm install source-map-explorer --save-dev
```

Ahora haremos la build del proyecto pero indicando que queremos generar los `source-maps` de los ficheros asociados:

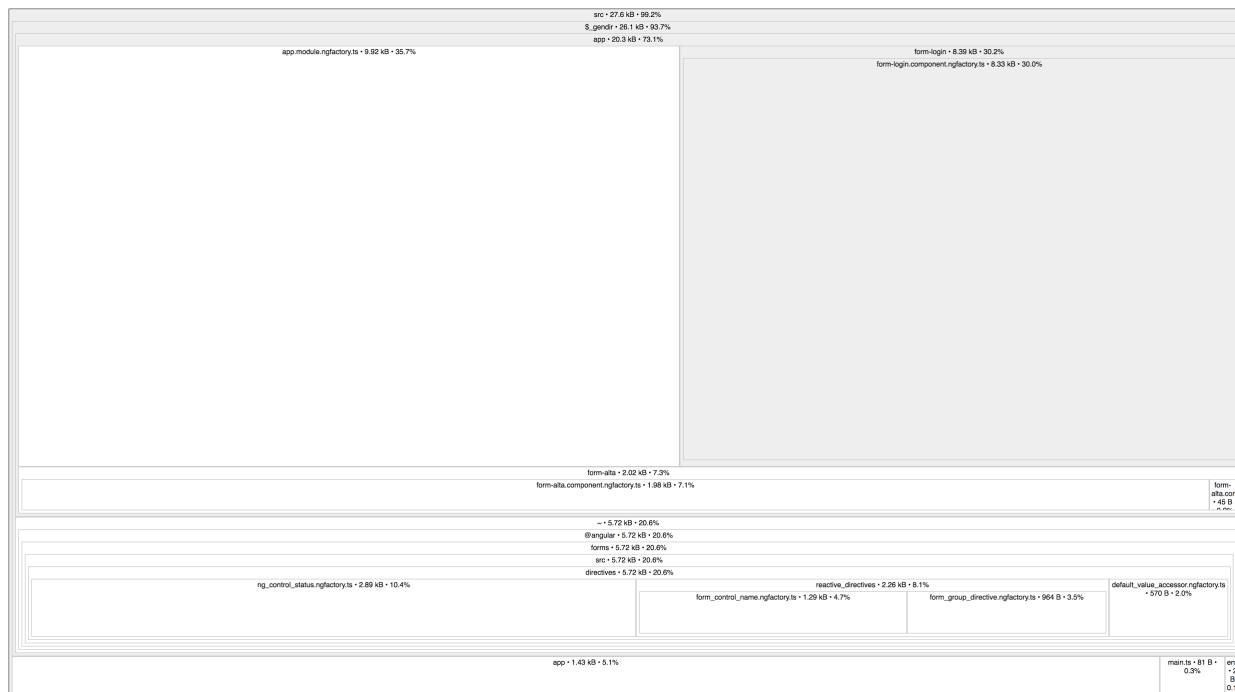
```
1 $> ng build --prod -sm
```

El flag `--prod` lleva implícito el uso de AOT y la carga del fichero de propiedades de producción. Con lo que nos aseguramos de que no incluimos el peso del compilador y que se hace `tree-shaking` para eliminar el peso de referencias muertas.

De esta forma en la carpeta “`dist`” se crean los ficheros de distribución con sus correspondientes `.map`. Ahora podemos ejecutar la herramienta indicando uno de estos bundles para ver el detalle del tamaño de cada elemento importado en este fichero.


```
1 $> source-map-explorer dist/main.js
```

De esta forma se abrirá el navegador y obtendremos un gráfico similar a este:



El gráfico es interactivo de forma que podemos pinchar en cada una de las secciones para verla más en detalle.

Verifica el tamaño del bundle principal

Si el tamaño del bundle principal después de AOT y de la verificación de usos sigue siendo excesivo, tendremos que plantearnos la modularización de la aplicación y el uso de lazy loading para la carga.

Este proceso no se puede hacer de forma automática, tendremos que ver por dónde podemos cortar el módulo principal para crear módulos secundarios por funcionalidad que puedan ser cargados de forma perezosa. De esta manera la carga inicial de la aplicación será mucho más rápida y la experiencia de usuario mejorará drásticamente.

No subas los .map

Ten en cuenta antes de subir a producción no subir los archivos .map que haya dentro de la carpeta “dist” para que no sea fácil ver las tripas de la aplicación desde las herramientas del navegador.

Para evitar esto, bien haces un script que los elimine o ejecutas la build sin el modificador `-sm`, de esta forma:

```
1 $> npm run build -- --prod
```

Genera la documentación

Podemos documentar nuestro proyecto de una manera más o menos estándar con la herramienta `compodoc`, la cual instalamos como paquete de npm:

```
1 $> npm install compodoc --save-dev
```

Su funcionamiento es muy sencillo solo tenemos que pasarle el `tsconfig.json` de la aplicación y la herramienta te genera la carpeta “documentation” con todos los ficheros necesarios para desplegar un site de documentación.

```
1 $> ./node_modules/.bin/compodoc -p tsconfig.json -s -o
```

Con `-p` indicamos el path del fichero `tsconfig.json`, con `-s` que levante el site de documentación en el puerto 8080 y con `-o` que abra la página principal en el navegador por defecto. El site de documentación será parecido a este:

blank-angular-cli documentation

Getting started
Modules
AppComponent
SearchUserModule
Components
Classes
Routes
Injectables

Modules / AppModule

Legend
Declarations Module Bootstrap Providers Exports

Zoom in Reset Zoom out

File
src/app/app.module.ts

Declarations ⓘ
AppComponent

Imports ⓘ
BrowserModule
SearchUserModule
RouterModule.forRoot(args)

Bootstrap ⓘ
AppComponent

Documentation generated using
compoDoc

Aplicar técnicas de optimización

Para poder aplicar correctamente técnicas de optimización antes tenemos que saber como funciona el “Browser Rendering”.

Entendiendo el “Browser rendering”

Toda aplicación web que se ejecuta en un navegador pasa por estas fases cada vez que quiere mostrar algo por pantalla:

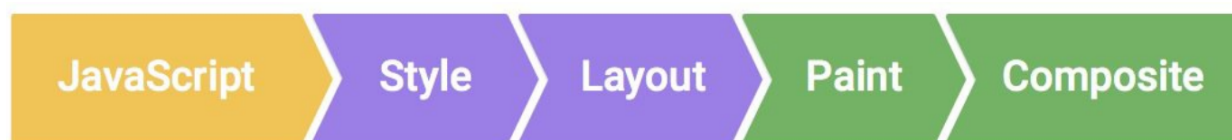


Image from developers.google.com

La sensación de buen rendimiento pasa porque todo el proceso anterior no tarde más de 17 ms, que es el límite de tiempo que el hilo principal puede estar bloqueado sin que nuestro ojo perciba la congelación de pantalla.

Pasamos a describir qué es lo que se hace en cada una de las fases:

- **JavaScript/CSS:** el proceso lo dispara algún evento que realiza un cambio visual por pantalla, por ejemplo, eliminar un elemento de una lista, navegar a otra página, ... Seguro que algo de tiempo necesita emplear en realizar dichas acciones.
- **Style:** en este paso se calcula el CSS de todos los elementos de la página, calculando cada uno de ellos y combinándolo con el resto de reglas CSS que afecten a cada elemento.
- **Layout:** una vez el navegador conoce los estilos de cada elemento tiene que posicionarlos en el lugar adecuado de la pantalla. Esto podría ser un proceso muy pesado sobre todo cuando insertamos un elemento al comienzo de una lista, ya que el navegador tiene que recalcular la posición de cada uno de los elementos de la lista.
- **Paint:** después de tener todo calculado, en esta fase se pintan los píxeles correspondientes por pantalla y en su capa adecuada.
- **Composite:** en esta fase se ordenan de forma adecuada las múltiples capas para que la aplicación se visualice correctamente.

Cómo mejorar el “Browser rendering”

Para poder mejorar el proceso de renderizado de la página en el navegador tenemos que perseguir dos objetivos:

- Evitar la ejecución de todos los pasos que se pueda.
- Realizar el trabajo dentro de una fase de la forma más eficiente posible.

En la mayoría de casos la fase que lleva más tiempo es la de “Layout” dado que tiene que recalcular la posición de todos los elementos de la página, por tanto tenemos que conseguir que se ejecute el menor número de veces posible.

Un truco para conseguir esto, es añadir la siguiente propiedad CSS a los elementos que no queremos que sean recalculados.

```
1 contain: layout;
```

Un ejemplo claro es si tenemos un modal por encima de la página y añadimos un elemento a ese modal, no queremos recalcular toda la página, solo la sección que ocupa el modal, por lo tanto ponemos esa propiedad al elemento del modal para que se renderice en 0.05 ms por los 56.90 ms que lo haría sin la propiedad. Esto supone una ganancia del 1425x

Otro caso de uso donde podemos evitar el uso de layout es a la hora de una animación. Si la animación la hacemos a base de reglas de CSS “clásicas” como:

```
1 margin-left: -20px;
```

Es seguro que el navegador va a necesitar la fase de “Layout” para mostrar los cambios. En cambio, si hacemos uso de `translate3d` esto hace que el navegador no tenga que hacer uso de la fase de “Layout”:

```
1 transform: translate3d(-20px, 0px, 0px);
```

Técnicas de optimización en Angular

En Angular tenemos las siguientes técnicas de optimización.

Eliminación de espacios en blanco

A partir de Angular 4.4 podemos hacer uso de la propiedad “`preserveWhitespaces`” en la definición de un componente para determinar si queremos o no mantener los espacios en blanco que se produzcan en el momento del transpilado.

Hay que tener cuidado pues esta propiedad podría afectar a nuestro layout. Lo que conseguimos eliminando todos los espacios en blanco posibles es una reducción al máximo del bundle resultante con AOT lo que implica una mejora en los tiempos de descarga.

Uso de web workers

Los Web Workers son la solución del estándar para ejecutar procesos pesados en otro hilo y no bloquear el hilo principal que es el encargado del pintar la interfaz.

Esto lo podemos ver de una forma muy gráfica, colocando un gif animado y ejecutando un proceso cuya ejecución tarde más de 17 ms. ¿Por qué 17 ms?

La explicación la encontramos en que los monitores suelen tener un refresco de 60 frames por segundo, lo que hace que el límite de tiempo que puedan estar congelados sin que lo percibamos sea de $1000 / 60 = 16.7$ ms, si el hilo principal se bloquea más tiempo nuestro ojo percibirá que la interfaz se ha congelado, en este caso, que el gif no se mueve o no lo hace de forma fluida.

Para poder aplicarlos en Angular, actualmente, tenemos dos opciones: configurar todo el proyecto para que se ejecute en web workers o no modificar el proyecto y hacer uso de ellos en los puntos en los que haga falta.

Para la primera opción os dejo un enlace donde Enrique Oriol nos explica cómo hacerlo paso a paso. [Blog²¹](#)

²¹<http://blog.enriqueoriol.com/2017/04/angular-webworkers.html>

La pega que le veo a esta forma de hacerlo es que, actualmente es requisito indispensable hacer un eject de angular-cli y modificar muchas cosas de la configuración de webpack del proyecto a mano, lo cual lo hace demasiado intrusivo para mi gusto.

La otra opción es hacer uso de la librería “angular2-web-workers” que podemos instalar ejecutando en la raíz de nuestro proyecto:

```
1 $> npm install --save angular2-web-worker
```

Esta librería nos proporciona un servicio de Angular que se encarga de encapsular el comportamiento de los Web Workers, que como ya hemos dicho pertenecen al estándar web.

Vemos en un ejemplo cuál es el problema y cómo hacer uso de este servicio para resolverlo.

Imagina que tienes un componente que renderiza un gif animado, es decir, una imagen donde se aprecia movimiento y que vamos a ejecutar el siguiente proceso que su ejecución lleva mucho más de 17 ms.

```
1 export class PocService {
2
3   constructor() { }
4
5   metodoPesado(): number {
6     let total = 0;
7     for (let i = 0; i < 10000000000; i++) {
8       total = total + i;
9     }
10
11     return total;
12   }
13
14 }
```

Al ejecutar este método del servicio apreciaríamos como el gif deja de moverse hasta que el proceso finaliza, porque bloquea el hilo principal de ejecución de la aplicación.

Para resolver esto, inyectamos a través del constructor el servicio WebWorkerService que nos proporciona la librería “angular2-web-worker” y creamos un nuevo método que va a devolver una promesa cuando el proceso que le indiquemos finalice, quedando de esta forma:

```
1  import { WebWorkerService } from 'angular2-web-worker';
2  import { Injectable } from '@angular/core';
3
4  @Injectable()
5  export class PocService {
6
7      constructor(private webWorkerService: WebWorkerService) { }
8
9      metodoConWebWorker(): Promise<number> {
10         return this.webWorkerService.run(this.metodoPesado);
11     }
12
13     metodoPesado(): number {
14         let total = 0;
15         for (let i = 0; i < 10000000000; i++) {
16             total = total + i;
17         }
18
19         return total;
20     }
21
22 }
```

De esta forma si llamamos al método con web worker desde el componente, recibiremos la promesa, la cual podemos manejar con el método then, como vemos a continuación:

```
1  import { PocService } from './poc.service';
2  import { Component } from '@angular/core';
3  import { WebWorkerService } from 'angular2-web-worker';
4
5  @Component({
6      selector: 'app-root',
7      templateUrl: './app.component.html',
8      styleUrls: ['./app.component.css'],
9      providers: [PocService, WebWorkerService]
10 })
11 export class AppComponent {
12
13     constructor(
14         private pocService: PocService
15     ) {}
16 }
```

```
17  ejecucionConWebWorker() {
18      const promise = this.pocService.metodoConWebWorker();
19      promise.then(total => console.log(total));
20  }
21
22  ejecucionSinWebWorker() {
23      const total = this.pocService.metodoPesado();
24      console.log(total);
25  }
26
27 }
```

La mayor ventaja que le veo a esta forma de hacerlo es que no tenemos que cambiar nada de configuración, recordad que un “ng eject” es irreversible.

Extracción de la configuración de la aplicación

Una de las cosas más importantes que tenemos que tener en cuenta a la hora de desplegar aplicaciones con Angular, es la de extraer la configuración de la aplicación, de forma que la aplicación se pueda ejecutar en cualquier entorno: desarrollo, pre-producción y producción sin tener que volver a construirse para ese entorno específico; sino sobreescribiendo un fichero de configuración externo.

La forma de gestionar la configuración en angular-cli es a través de los ficheros de environments que permiten construir la aplicación con los parámetros de configuración para desarrollo o para producción; pero esto es insuficiente para el objetivo que perseguimos de extraer la configuración por lo que vamos a editarlos para añadir una propiedad “configFile” indicando la URL donde vamos a alojar el fichero de configuración. Así quedaría el fichero src/environments/environment.ts:

```
1  declare var require: any;
2
3  export const environment = {
4      production: false,
5      version: require('../../package.json').version,
6      configFile: 'assets/config/config.json'
7  };
```

También se muestra como podríamos extraer la versión del fichero package.json para mostrarla en la aplicación.

El fichero src/environments/environment.prod.ts sería igual pero con la propiedad “production” a true:


```
1 declare var require: any;
2
3 export const environment = {
4   production: true,
5   version: require('../../package.json').version,
6   configFile: 'assets/config/config.json'
7 };
```

Es importante que el fichero de configuración se situe dentro de la carpeta assets para poder ser modificado en tiempo de despliegue. El contenido típico del fichero config.json podría ser algo así, donde apuntamos a un API del back que corresponda con el entorno:

```
1 {
2   "api": "http://api-back:8070/"
3 }
```

Ahora necesitamos un servicio que lea este fichero de configuración y nos permita utilizarlo en la aplicación. Esta funcionalidad la vamos a encapsular en un nuevo módulo (src/config/config.module.ts) con el siguiente contenido:

```
1 import { CommonModule } from '@angular/common';
2 import { HttpClientModule } from '@angular/common/http';
3 import { NgModule } from '@angular/core';
4
5 import { ConfigProxyService } from './config-proxy.service';
6 import { ConfigService } from './config.service';
7
8 @NgModule({
9   imports: [
10     CommonModule,
11     HttpClientModule
12   ],
13   declarations: [],
14   providers: [ConfigService, ConfigProxyService]
15 })
16 export class ConfigModule { }
```

Ahora vamos a implementar la interfaz de nuestro dominio que va a almacenar la estructura del fichero de configuración (src/config/config.ts):

```
1 export interface Config {  
2     api: string;  
3 }
```

Ahora vamos a implementar el servicio ConfigProxyService (src/config/config-proxy.service.ts) donde a través de la definición de la propiedad “configFile” y el servicio HttpClientService de Angular vamos a cargar el JSON asociado con la configuración.

```
1 import { HttpClient } from '@angular/common/http';  
2 import { Injectable } from '@angular/core';  
3 import { Observable } from 'rxjs/Observable';  
4  
5 import { environment } from './../../environments/environment';  
6 import { Config } from './config';  
7  
8 @Injectable()  
9 export class ConfigProxyService {  
10  
11     constructor(private httpClient: HttpClient) { }  
12  
13     getConfig(): Observable<Config> {  
14         return this.httpClient.get<Config>(`${environment.configFile}`);  
15     }  
16  
17 }
```

Para comprobar que todo es correcto implementamos el siguiente test de integración (src/config/config-proxy.service.spec.ts):

```
1 import { HttpClientModule } from '@angular/common/http';  
2 import { async, inject, TestBed } from '@angular/core/testing';  
3  
4 import { ConfigProxyService } from './config-proxy.service';  
5  
6 describe('ConfigProxyService', () => {  
7     beforeEach(() => {  
8         TestBed.configureTestingModule({  
9             imports: [HttpClientModule],  
10            providers: [ConfigProxyService]  
11        });  
12    });  
13
```

```

14   it('should be created', inject([ConfigProxyService], (service: ConfigProxyServ\
15   ice) => {
16       expect(service).toBeTruthy();
17   }));
18
19   it('should get configuration', async(() => {
20       const service: ConfigProxyService = TestBed.get(ConfigProxyService);
21       service.getConfig().subscribe(
22           config => expect(config.api).not.toBeNull()
23       );
24   }));
25
26   });

```

Como se explico en el tema de testing, ahora la clave es crear un fake de este servicio para poder ser utilizado en los test unitarios. De forma que creamos el fake con el siguiente contenido (src/config/config-proxy.service.fake.ts) , donde a través del método of de Observable podemos devolver información síncrona:

```

1  import { Injectable } from '@angular/core';
2  import { Observable } from 'rxjs/Observable';
3  import { of } from 'rxjs/observable/of';
4
5  import { Config } from './config';
6
7  @Injectable()
8  export class ConfigProxyServiceFake {
9
10     constructor() { }
11
12     getConfig(): Observable<Config> {
13         return of(CONFIG_FAKE);
14     }
15
16 }
17
18 const CONFIG_FAKE: Config = {
19     'api': 'http://api-back:8070/'
20 };

```

Ahora vamos a implementar el servicio que se va a encargar de mantener la configuración en memoria y será el que tengamos que inyectar para recuperarla en los elementos que la

requieran (src/config/config.service.ts), fíjate que la información se guarda en el objeto config y que es importante que este servicio se cargue en el inyector principal para que sea singleton y se pueda inyectar en cualquier parte del árbol que forma los elementos de la aplicación.

```
1  import { Injectable } from '@angular/core';
2
3  import { Config } from './config';
4  import { ConfigProxyService } from './config-proxy.service';
5
6  @Injectable()
7  export class ConfigService {
8
9      config: Config;
10
11      constructor(private proxy: ConfigProxyService) { }
12
13      load() {
14          return new Promise((resolve) => {
15              this.proxy.getConfig().subscribe(
16                  config => {
17                      this.config = config;
18                      resolve();
19                  }
20              );
21          });
22      }
23
24  }
```

Ahora creamos el test unitario asociado para comprobar la validez de la lógica en la carga de la información de configuración. (src/config/config.service.spec.ts)

```
1  import { inject, TestBed } from '@angular/core/testing';
2
3  import { ConfigProxyService } from './config-proxy.service';
4  import { ConfigProxyServiceFake } from './config-proxy.service.fake';
5  import { ConfigService } from './config.service';
6
7  describe('ConfigService', () => {
8      beforeEach(() => {
9          TestBed.configureTestingModule({
```

```
10     providers: [  
11         ConfigService,  
12         {provide: ConfigProxyService, useClass: ConfigProxyServiceFake}  
13     ]  
14     });  
15 });  
16  
17 it('should be created', inject([ConfigService], (service: ConfigService) => {  
18     expect(service).toBeTruthy();  
19 }));  
20  
21 it('should load configuration', () => {  
22     const service: ConfigService = TestBed.get(ConfigService);  
23     service.load();  
24     expect(service.config.api).not.toBeNull();  
25 });  
26 });
```

Ahora necesitamos que esta configuración se cargue de forma automática en el arranque de la aplicación. Para ello tenemos que editar el fichero `src/app.module.ts` para añadir nuestra lógica al provider de Angular `APP_INITIALIZER`, haciendo uso de la receta `useFactory` y la propiedad `multi` a `true`, dado que es un array:

```
1  import { APP_INITIALIZER, NgModule } from '@angular/core';  
2  import { BrowserModule } from '@angular/platform-browser';  
3  
4  import { AppComponent } from './app.component';  
5  import { ConfigModule } from './config/config.module';  
6  import { ConfigService } from './config/config.service';  
7  
8  export function ConfigLoader(configService: ConfigService) {  
9      return () => configService.load();  
10 }  
11  
12 @NgModule({  
13     declarations: [  
14         AppComponent  
15     ],  
16     imports: [  
17         BrowserModule,  
18         ConfigModule  
19     ],
```

```
20   providers: [  
21     {provide: APP_INITIALIZER, useFactory: ConfigLoader, deps: [ConfigService], \  
22     multi: true}  
23   ],  
24   bootstrap: [AppComponent]  
25 })  
26 export class AppModule { }
```

De esta forma al arrancar la aplicación Angular y antes de mostrar el componente principal, Angular ejecutará la lógica de carga de la configuración inicial. Para cambiar esta configuración, simplemente tenemos que sobrescribir el fichero `src/assets/config/config.json` con cualquiera que tengamos dentro de la carpeta `src/assets/config`, por ejemplo, `config.pre.json`, `config.pro.json`, etc... lo importante es que todas tengas las mismas propiedades con el valor que corresponda en su entorno de ejecución.

Despliegue de la solución en nginx

Voy a mostrar como hacer el despliegue en un nginx recién instalado en un servidor con Ubuntu.

Simplemente tenemos que copiar el contenido de la carpeta “dist” de nuestro proyecto dentro de la ruta `/var/www/html` y navegar a la URL del servidor para ver nuestra aplicación.

Ahora prueba a hacer un refresco de la página y verás que te muestra un error indicando que no encuentra la aplicación.

Para resolver esto tenemos que editar el fichero `/etc/nginx/sites-available/default` y cambiar la línea:

```
1 location / {  
2     try_files $uri $uri/ =404;  
3 }
```

Por:

```
1 location / {  
2     try_files $uri $uri/ /index.html;  
3 }
```

Esto es porque al ser una SPA siempre tiene que tener una referencia al `index.html`, a no ser que hayamos cambiado la estrategia del router a Hash (#).

Para cambiar la configuración simplemente tenemos que sobrescribir el fichero `/var/www/html/assets/config/config.json` con el fichero que corresponda con su entorno de ejecución.

Empaquetado con Docker

Docker es la solución estándar para el empaquetado de aplicaciones en contenedores donde se define todo lo necesario para la ejecución de la aplicación de forma aislada a la máquina donde se ejecuta.

En la práctica permite ejecutar aplicaciones Angular sin necesidad de que la máquina de destino tenga instalado un runtime de NPM, solo tiene que tener instalado un runtime de Docker, lo que permite que las aplicaciones puedan ejecutarse con distintas versiones de NPM sin colisiones.

Para ello en la raíz del proyecto vamos a crear el fichero `Dockerfile` donde vamos a especificar el entorno y como se tiene que ejecutar nuestra aplicación:

```
1 FROM nginx:1.13.3-alpine
2
3 ## Copy our default nginx config
4 COPY nginx/default.conf /etc/nginx/conf.d/
5
6 ## Remove default nginx website
7 RUN rm -rf /usr/share/nginx/html/*
8
9 ## From 'builder' stage copy over the artifacts in dist folder to default nginx \
10 public folder
11 COPY dist /usr/share/nginx/html
12
13 ENV PHASE int
14
15 COPY ./entrypoint.sh /entrypoint.sh
16 RUN chmod +x /entrypoint.sh
17 ENTRYPOINT ["/entrypoint.sh"]
```

Partimos de una imagen base de nginx con alpine, que es una distribución ligera de Linux de tan solo 8 Mb. Después copiamos el fichero `default.conf` de nuestro proyecto que contiene la configuración de nginx para hacer el redirect en aplicaciones SPA, luego borramos todo el contenido que viene por defecto con nginx y hacemos la copia de la carpeta `dist` con los ficheros preparados para producción.

Definimos la variable de entorno PHASE que representa la fase de ejecución del contenedor, por defecto integración, y copiamos el fichero entryptoint.sh donde realizamos la sobreescritura del fichero config.json en base a la variable de entorno PHASE con la que se ejecute el contenedor, dándole permisos de ejecución. Por último, definimos el entryptoint que ejecuta el contenido del fichero, que vemos a continuación:

```
1  #!/bin/sh
2
3  cp -fv /usr/share/nginx/html/assets/config/config.${PHASE}.json /usr/share/nginx\
4  /html/assets/config/config.json
5  nginx -g 'daemon off;'
```

El contenido de la configuración de nginx lo almacenamos en el fichero nginx/default.conf

```
1  server {
2
3      listen 80;
4
5      sendfile on;
6
7      default_type application/octet-stream;
8
9
10     gzip on;
11     gzip_http_version 1.1;
12     gzip_disable      "MSIE [1-6]\.";
13     gzip_min_length   256;
14     gzip_vary         on;
15     gzip_proxied      expired no-cache no-store private auth;
16     gzip_types        text/plain text/css application/json application/javascript \
17 application/x-javascript text/xml application/xml application/xml+rss text/javas\
18 cript;
19     gzip_comp_level   9;
20
21
22     root /usr/share/nginx/html;
23
24
25     location / {
26         try_files $uri $uri/ /index.html =404;
27     }
```



```
28  
29 }
```

Una vez creado el fichero Dockerfile vamos a construir la imagen de esta forma, ejecutando en un terminal en la raíz del proyecto:

```
1 $> docker build -t guia/app-web:[version del package.json] .
```

De esta forma ya sea con Docker o con Docker Compose podemos ejecutar el contenedor estableciendo la variable de entorno y el puerto en el que se va a quedar escuchando.

Con Docker:

```
1 $> docker run -d -e ENV=pre -p 8993:80 guia/app-web:${APP_VERSION}
```

O su equivalente con Docker Compose:

```
1 version: '3'  
2  
3 services:  
4  
5   devops-web-develop:  
6     image: guia/app-web:${APP_VERSION}  
7     environment:  
8       - "PHASE=int"  
9     ports:  
10      - "8993:80"  
11     networks:  
12      - devops-web-develop  
13  
14 networks:  
15   devops-web-develop:
```

Programación reactiva (RxJS)

Es un nuevo paradigma de programación que extiende el [patrón de diseño Observer](#)²² definido por el GoF.

Me gusta mucho este ejemplo de [Yakov Fain](#)²³ para diferenciarlo de la programación imperativa.

Tenemos un ejemplo muy simple de suma de dos números:

```
1 let a = 2;
2 let b = 4;
3 let c = a + b; //c = 6
```

Pero que pasa ahora si cambiamos el valor de a y b:

```
1 let a = 2;
2 let b = 4;
3 let c = a + b; // c = 6
4
5 a = 55;        // c = 6 debería ser 59
6 b = 20;        // c = 6 debería ser 75
```

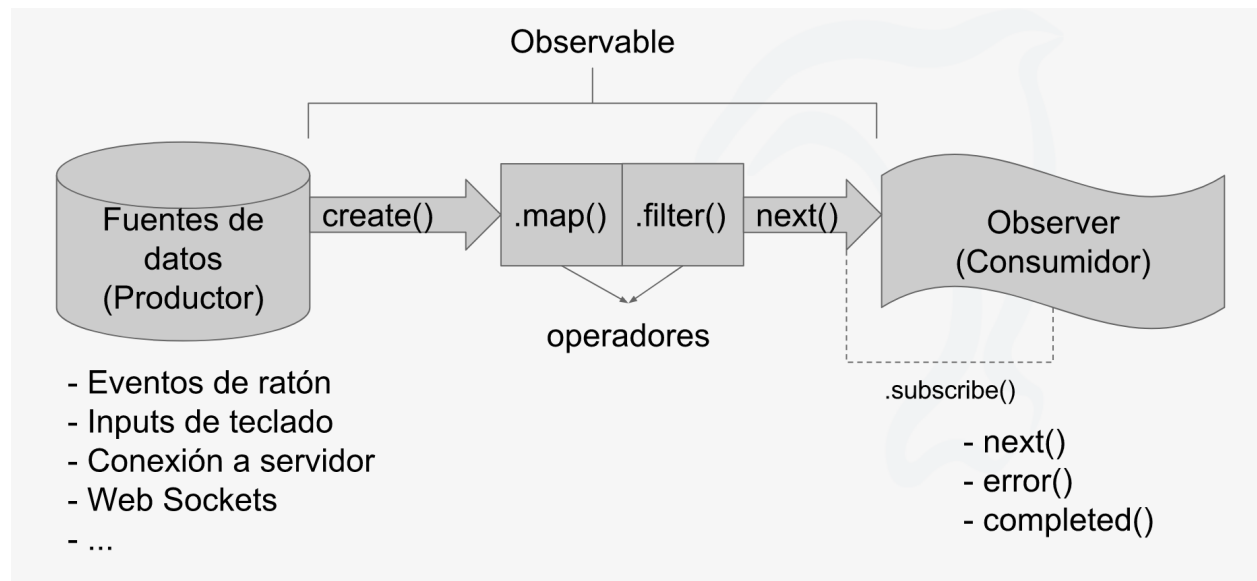
En programación imperativa está claro que c sigue siendo 6, pero lo que demandan ahora la mayoría de aplicaciones es que este tipo de datos se actualice automáticamente. Imaginad que hacéis uso de Excel, en una celda ponemos un dato, en otra otro dato y en otra el resultado (=sum(a,b)). Lo que se espera es que si cambiamos un dato se actualice automáticamente el resultado. Pues eso sería el concepto de programación reactiva.

Su uso no se limita a la programación web sino que muchos lenguajes como Java ya tienen una librería para implementar este paradigma del lado del servidor.

²²https://en.wikipedia.org/wiki/Observer_pattern

²³<https://yakovfain.com>

Conceptos básicos



DataSource

La fuente de datos, puede tener distintas naturalezas: datos que vienen de un servidor externo, un array de datos, eventos de ratón, inputs de teclado, etc...

Observable

Es la estructura que permite convertir la fuente de datos en un stream o flujo de información que permita la suscripción de distintos observers.

Son dos características fundamentales las que los distinguen de las promesas:

- **Son lazy:** es decir, no se ejecutan hasta que alguien ejecuta un subscribe.
- **Son cancelables:** es decir, en cualquier momento podemos cancelar la suscripción para dejar de recibir datos.

Formas de crear un Observable

- **Observable.create():** dentro tenemos que establecer la lógica de cuando se emite un dato con next() y cuando el stream se ha completado con un complete(); además podemos establecer una lógica para error(). Es decir recibe un objeto que cumple con la interfaz de Observer.
- **Observable.of():** crea un observable de un valor estático como puede ser un JSON, personalmente lo utilizo mucho para hacer fake de los servicios de proxy.

- **Observable.from(anArray):** convierte el array pasado como argumento en un Observable.
- **Observable.fromEvent(myInput, 'keyup'):** convierte el evento de algún elemento de HTML en un Observable.
- **Observable.fromPromise(myPromise):** convierte una promesa en un Observable.
- **Observable.range(10, 100):** crea un Observable que emite una secuencia de enteros entre 10 y 100.
- **Observable.interval(x):** crea un Observable que emite una secuencia de enteros cada x milisegundos.

Subscriber

Es el elemento que realiza la suscripción al Observable indicando que el consumidor quiere ser informado de ese flujo de información.

Observer

Es la lógica del subscriber, es el que determina que hacer en los tres casos específicos que se pueden dar:

- `next()` \Rightarrow si todo es correcto
- `error()` \Rightarrow si existe un error
- `completed()` \Rightarrow si el stream ha finalizado

Subscription

Es el resultado de la suscripción, nos permite almacenar esta referencia para poder desuscribirnos en cualquier momento, indicando que ya no estamos interesados en seguir recibiendo el flujo de información.

Operadores

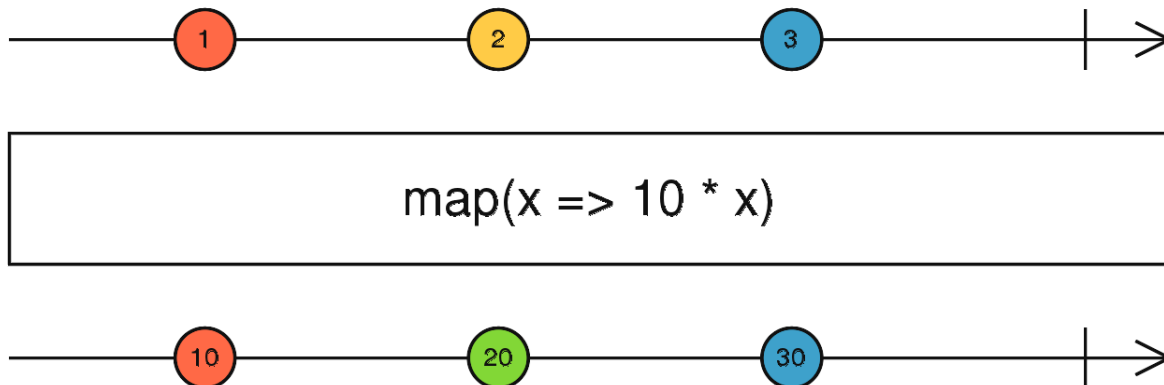
Un “operator” es una función de transformación sobre un Observable. Es decir, recibe un Observable, le aplica la transformación deseada y devuelve el Observable transformado. Son operaciones que se hacen antes de realizar la suscripción.

En la documentación oficial se explican los diagramas de canicas o RxMarbles (todos los diagramas están obtenidos de la documentación oficial)

Estos son los principales operadores que podemos utilizar:

.map(dato => function(dato))

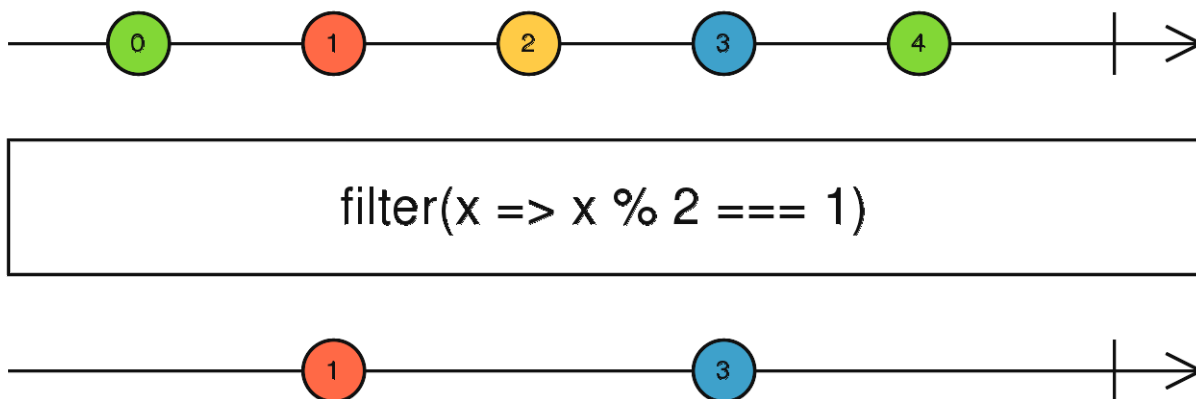
Permite aplicar una función de transformación sobre cada elemento del stream.



Como se ve a cada una de las canicas se le aplica la función de multiplicar por 10 su valor en el orden en el que son emitidas.

.filter(dato => condicion booleana)

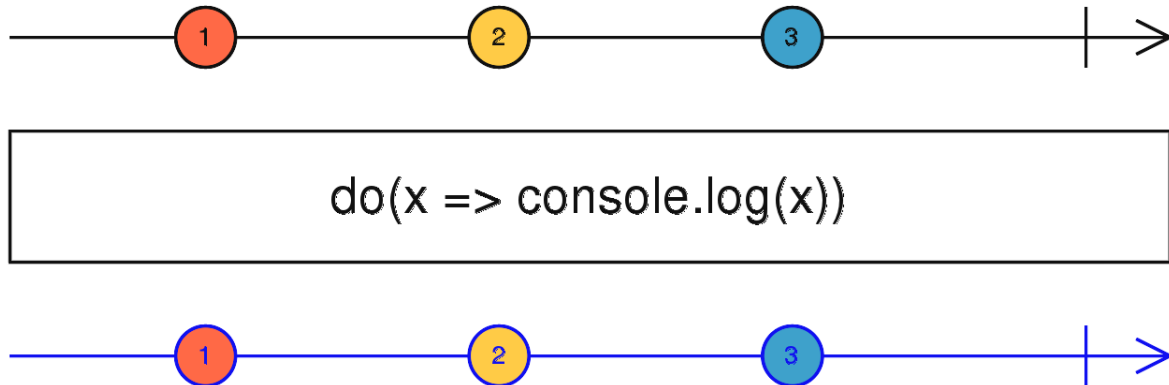
Permite definir una condición para filtrar cada elemento del stream. Si se resuelve a true el dato se incluye en el siguiente Observable, si se devuelve a false se “elimina”, ya que la fuente de datos permanece intacta.



En este caso solo nos quedamos con los valores emitidos que sean impares.

.do(x => function(x))

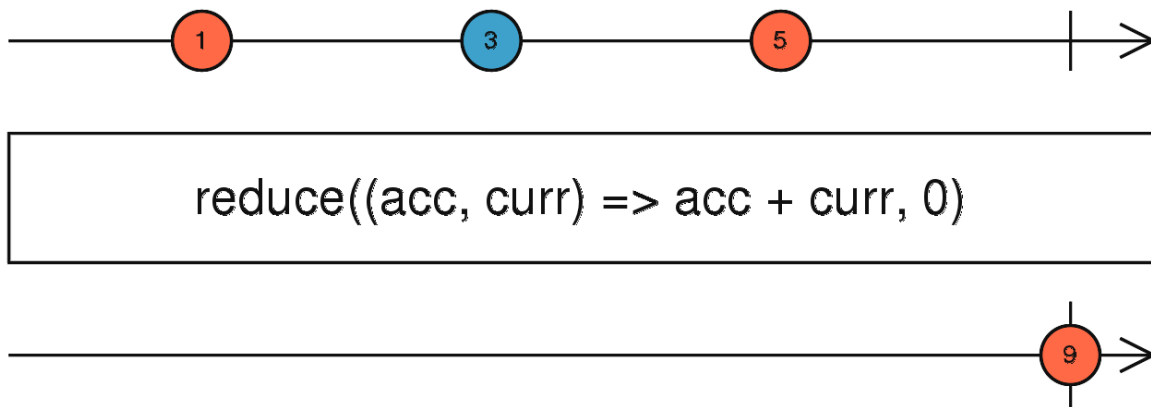
Este operador intercepta cada elemento emitido y le aplica una función pero siempre devuelve el mismo valor.



Generalmente se utiliza para depurar y ver que el valor de los elementos en un determinado punto del stream con un `console.log`

.reduce()

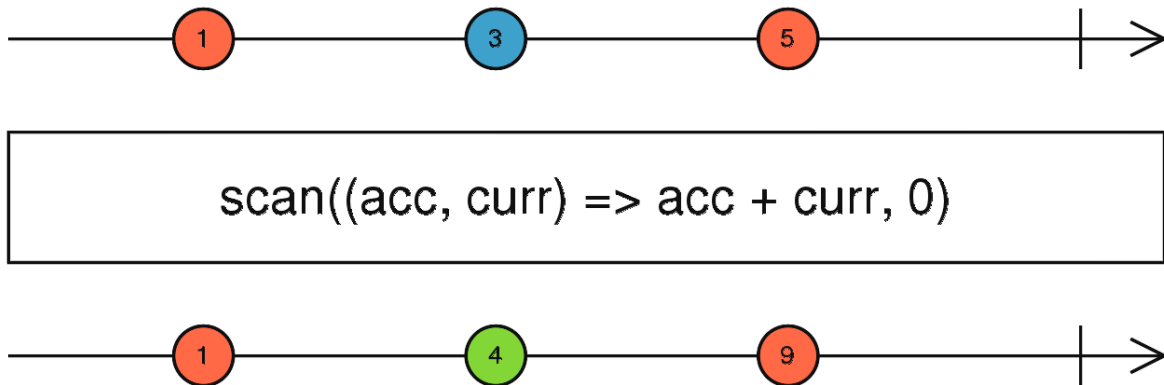
Este operador permite agregar los valores emitidos y aplicarles una función. Siempre recibe un segundo parámetro que es el valor inicial del acumulador.



Aquí vemos un caso muy común donde se utiliza el operador `reduce` para sumar el valor total de los valores emitidos hasta ese momento.

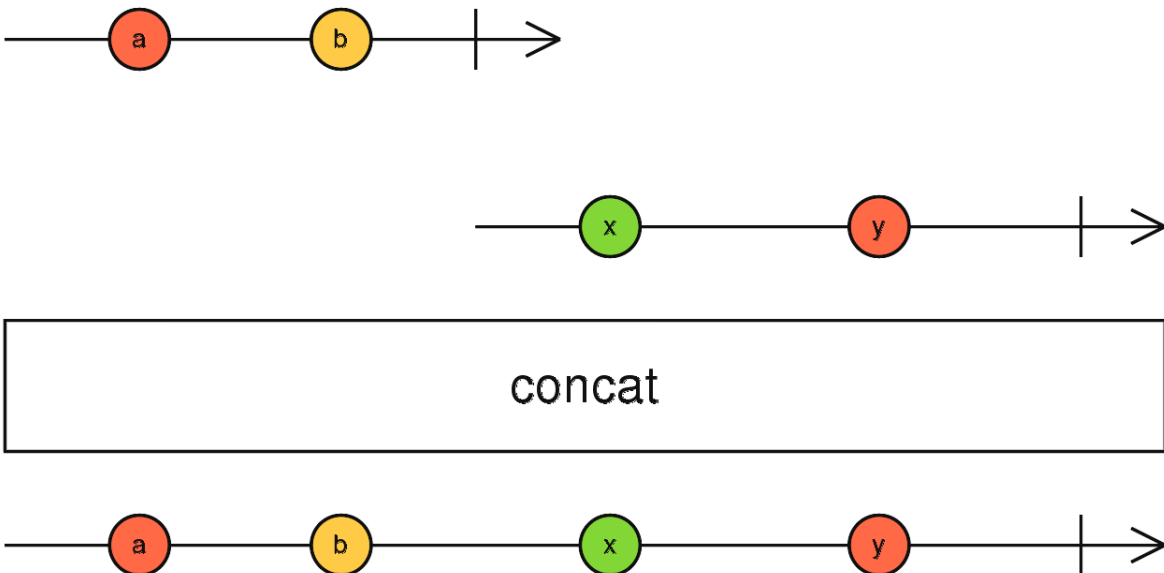
.scan()

Este operador es muy similar a `reduce()` con la diferencia de que emite el valor calculado por cada elemento, en vez de esperar a tener un conjunto de ellos.



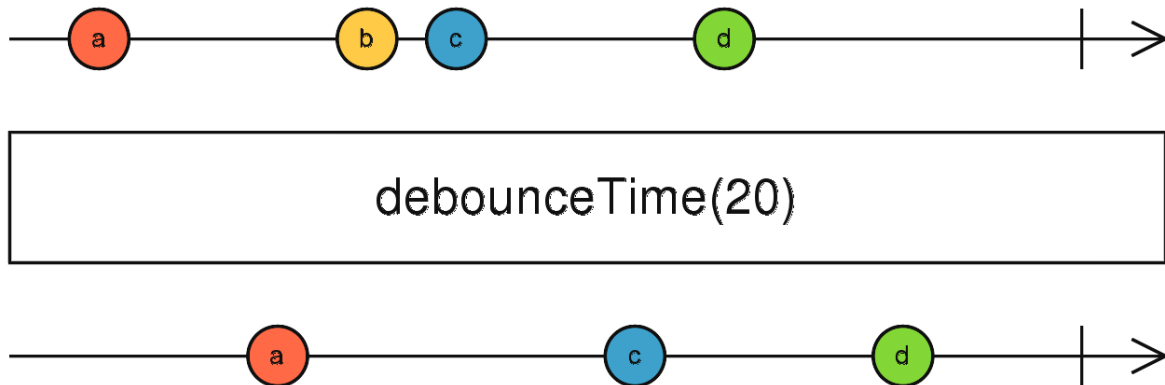
.concat(otherObservable)

Une en un único stream dos o más Observables.



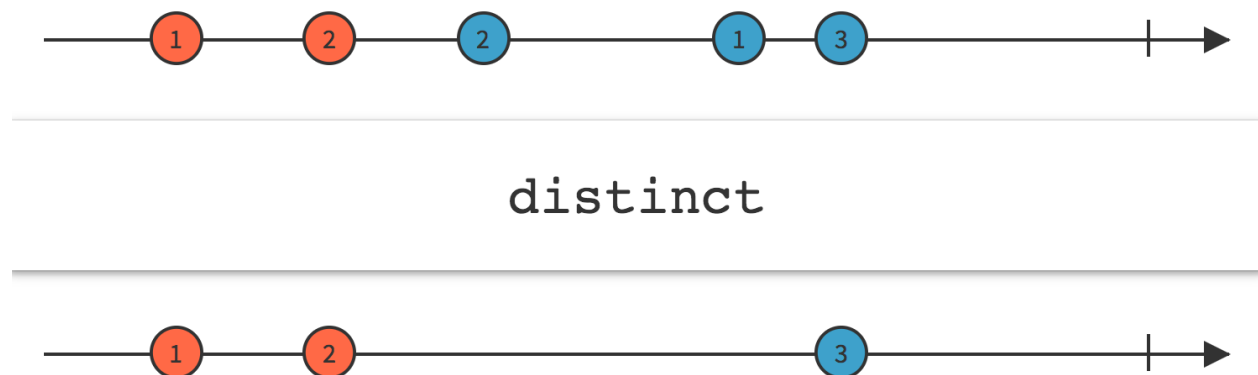
.debounceTime(x ms)

Indica que el valor del observable se emite cuando pase el valor en milisegundos indicado. Nos permite acumular valores emitidos por otro observable. Un ejemplo muy común es cuando queremos emitir lo que un usuario va escribiendo en un input pasado x milisegundos después de dejar de escribir/emitir.



.distinct()

No emite valores que haya emitido previamente. Nos sirve para eliminar duplicados en un array, por ejemplo.



.distinctUntilChanged()

Igual que el anterior pero no teniendo en cuenta solo el último valor emitido. Es decir, no emite valor si el último valor emitido es igual.

.switchMap()

Este operador detecta si hay un nuevo valor emitido y mata el anterior quedándose con el nuevo.

Un ejemplo práctico, puede ser, si lanzamos una consulta con lo que vamos escribiendo en un input, a medida que escribimos se va lanzando un nuevo valor del input con una

nueva consulta. Con este operador vemos como las consultas se cancelan hasta que no dejamos de escribir y se lanza la consulta con el texto completo, en caso de que no sigamos escribiendo.

.catch()

Permite gestionar los errores que ocurran en la cadena de operadores de un observable, de otra forma si ocurre un error no esperado el observable directamente morirá.

.buffer(obs)

Permite almacenar en memoria el último valor emitido hasta que se sustituye por la emisión de un valor nuevo. Puede funcionar como caché.

.bufferTime(x)

Permite cachear el último valor emitido durante x milisegundos. Es decir, pasado ese tiempo haya o no un nuevo valor emitido se invalida la caché.

.bufferCount(n)

Permite cachear el valor un máximo de n veces. Pasado este valor se invalida.

Subject

Representa un objeto que es tanto una secuencia observable como un observer. Permite notificar a todos los suscriptores que ya estén suscritos, a los futuros solo les notifica del último valor. Es como si fuera una pila donde me mete un valor y si no se lee se machaca con el siguiente. Existen distintos tipos:

BehaviorSubject

Representa un valor que cambia a lo largo del tiempo. Los observadores se suscriben para recibir el valor inicial y luego todos los valores siguientes.

ReplaySubject

Representa un objeto que es una secuencia observable como un observer. Permite notificar a todos los suscriptores y futuros suscriptores de todos los pasos de la secuencia. Es como si tuviera memoria y fuera una cola persistente.

AsyncSubject

Solo almacena el último valor y lo publica cuando la secuencia está completa. Se utiliza para situaciones “hot” que tiene que completarse antes de que ningún observador puede suscribirse. Los futuros observadores recibirán el valor publicado.

Lo encontramos en el funcionamiento del servicio HttpClient de Angular.

Casos de uso

Gracias a la programación reactiva podemos implementar casos de uso muy comunes de la manera más sencilla, reutilizable y mantenible posible.

Evitar problema al llamar dos veces a un Observable

Imagina que implementamos un componente que permite seleccionar una película y hacer una consulta al backend para obtener los datos relacionados, el código podría ser algo así:

```
1  @Component({
2    selector: 'movie-showings-component',
3    templateUrl: './movie-showings.component.html'
4  })
5  export class MovieShowingsComponent {
6    public movieTitle: string;
7    public showings: string[];
8
9    constructor(private backend: Backend) {}
10
11   selectMovie(movieTitle: string) {
12     this.movieTitle = movieTitle;
13
14     this.backend.getShowings(movieTitle).subscribe(showings => {
15       this.showings = showings;
16     });
17   }
18 }
```

Podemos tener el problema de que el usuario seleccione en poco tiempo más de una película y se de el caso en que por pantalla se visualiza un título pero la información relacionada es de otra película.

La fuente del problema es que estamos creando Observables independientes cada vez que el usuario selecciona el título de una película.

Para evitar esto, tenemos que crear un único Observable de Observables, y gracias al operador `switchMap` quedarnos solo con el último seleccionado, cancelando los que hubiera en proceso.

Para hacer esto, creamos un atributo de tipo `Subject<string>` y en el método `selectMovie` además de establecer el valor de la película seleccionada, enviamos el título de la película en el `next()` del Observable.

Ahora en el método `ngOnInit` utilizamos el operador `switchMap()` con el `Subject` de forma que solo dejamos que se suscriban a la última petición, cancelando el resto.

En la lógica del `switchMap` hacemos la llamada al backend y nos suscribimos para recuperar los resultados y mostrarlos por pantalla.

El código resultante sería el siguiente:

```
1  @Component({
2    selector: 'movie-showings-cmp',
3    templateUrl: './movie-showings.component.html'
4  })
5  export class MovieShowingsComponent implements OnInit {
6    public movieTitle: string;
7    public showings: string[];
8
9    private getShowings = new Subject<string>();
10
11    constructor(private backend: Backend) { }
12
13    ngOnInit(){
14      this.getShowings.switchMap(movieTitle =>
15        this.backend.getShowings(movieTitle))
16        .subscribe(showings => {
17          this.showings = showings;
18        });
19    }
20
21    showShowings(movieTitle: string) {
22      this.movieTitle = movieTitle;
23      this.getShowings.next(movieTitle);
24    }
25  }
```

Encadenar dos peticiones donde la segunda depende de los datos de la primera

Este es un caso típico donde necesitamos los datos de otra llamada para poder completar otra petición.

De forma fácil esto podría resolverse encadenando la segunda llamada en el subscribe de la primera, pero aunque esto funciona, no es aconsejable por los problemas que pueda dar tener dos suscripciones abiertas. Como vemos en el siguiente ejemplo:

```
1  this.http.get('./src/data.json')
2    .map(data => data.json())
3    .subscribe(res => {
4      this.http
5        .get(`./src/data-${res.id}.json`)
6        .map(data => data.json())
7        .subscribe(res => {
8          console.log(res);
9        });
10 });
```

La buena práctica para hacer esto determina que se haga uso del operador mergeMap, como se ve en el siguiente ejemplo:

```
1  this.http.get('./src/data.json')
2    .map(data => data.json())
3    .mergeMap(data =>
4      this.http.get(`./src/data-${data.id}.json`)
5      .map(data => data.json())
6    )
7    .subscribe(res => {
8      console.log(res);
9    });
```

De esta forma lo resolvemos con una única suscripción, y podemos manejar los errores de ambas en el mismo manejador.

En caso de que las llamadas fueran independientes y devolvieran la misma estructura de datos, haríamos uso del operador merge()

El anterior código es “blocking” así que es mejor hacerlo de esta forma que es “progressive”.

```
1  this.dataGet$ = this.http.get('./src/data.json');
2  this.detailGet$ = this.dataGet$.mergeMap(data =>
3      this.http.get(`./src/data-${data.id}.json`)
4      .map(data => data.json())
5  )
```

En este caso si que tenemos dos subscripciones pero que pueden simplificarse con el uso de async pipe.

Sistema de notificaciones a través de bus

Una de las maneras más elegantes que existen para mostrar mensajes al usuario es utilizando notificaciones por pantalla, de hecho muchas librerías ya cuentan con componentes de tipo growl para mostrarlos. Pero claro tenemos que utilizar ese componente en todos los puntos de la aplicación donde se vaya a necesitar y es muy tedioso... Entonces es cuando la reactividad viene en nuestra ayuda a través de un bus de comunicación.

Lo primero es crear el servicio con nuestro bus, el cual puede ser de esta forma:

```
1  import { Observable } from 'rxjs/Observable';
2  import { Notificacion } from './notificacion';
3  import { ReplaySubject } from 'rxjs/ReplaySubject';
4  import { Injectable } from '@angular/core';
5
6  @Injectable()
7  export class NotificacionesBusService {
8
9      showNotificacionSource: ReplaySubject<Notificacion>;
10
11      constructor() {
12          this.showNotificacionSource = new ReplaySubject<Notificacion>();
13      }
14
15      getNotificacion(): Observable<Notificacion> {
16          return this.showNotificacionSource.asObservable();
17      }
18
19      showError(msg: string, summary?: string) {
20          this.show('error', summary, msg);
21      }
22
23      showSuccess(msg: string, summary?: string) {
```

```

24     this.show('success', summary, msg);
25 }
26
27 showInfo(msg: string, summary?: string) {
28     this.show('info', summary, msg);
29 }
30
31 showWarn(msg: string, summary?: string) {
32     this.show('warn', summary, msg);
33 }
34
35 private show(severity: string, summary: string, msg: string) {
36     const notificacion: Notificacion = {
37         severity: severity,
38         summary: summary,
39         detail: msg
40     };
41     this.notify(notificacion);
42 }
43
44 private notify(notificacion: Notificacion): void {
45     this.showNotificacionSource.next(notificacion);
46 }
47
48 }

```

Como vemos primero declaramos dos atributos: uno de tipo `ReplaySubject` y el otro de tipo `Observable` del tipo de objeto que queremos transportar en nuestro bus y con el método “next” del `Observable` lo enviamos al bus.

Ahora tenemos que tener un componente que esté escuchando este evento y sepa qué hacer con él. Lo más sencillo es hacer uso del componente `p-growl` de `PrimeNg` y editar el fichero `app.component.html` de esta forma:

```

1 <p-growl [(value)]="msgs" life="2000"></p-growl>
2 <router-outlet></router-outlet>

```

Para poder hacer uso del componente antes tenemos que instalar la librería `PrimeNg` y declarar el módulo `GrowlModule` en la sección “imports” de `app.module`

Ahora en la lógica del componente inicializamos el escuchador del bus, el mejor sitio para hacerlo es el método `ngOnInit`, de esta forma:

```
1 msgs: Notificacion[] = [];  
2 ngOnInit() {  
3   this.notificacionesBus.getNotificacion.subscribe(  
4     (notificacion) => {  
5       this.msgs = [];  
6       this.msgs.push(notificacion);  
7     }  
8   );  
9 }
```

Esto es ya como funciona el componente de PrimeNG que recibe un array de objetos Notificacion (realmente utiliza la interfaz Message pero es para no casarnos). Lo importante es que cuando se envía una notificación por el bus, se ejecuta el método subscribe que establece el mensaje y el componente de PrimeNG ya es capaz de mostrarlo por pantalla.

Por último la interfaz Notificacion podemos implementarla de esta forma:

```
1 export interface Notificacion {  
2   severity: string;  
3   summary: string;  
4   detail: string;  
5 }
```

Ahora desde cualquier parte de la aplicación (dependerá de donde inyecte el provider con el servicio del bus) puede inyectar el bus y llamar a cualquiera de los métodos (showError, showInfo, ...) y automáticamente se mostrará el mensaje al usuario por pantalla.

Repetir una llamada Http cada cierto tiempo con actualización de datos sin parpadeo de pantalla

Otro de los casos de uso que nos podemos encontrar es el de tener que repetir una llamada Http al servidor cada cierto tiempo, lo que se conoce como pulling. (Si es cada muy poco tiempo es mejor que sea el servidor el que se encargue de actualizar la aplicación a través de Web Sockets).

El caso es que podrías hacer esto en en el ngOnInit de algún componente:

```
1 setInterval(() => {  
2   this.users$ = this.proxy.getUsers()  
3 }, 2000)
```

Si haces la prueba verás que tus datos se refrescan cada 2 segundos pero con un ligero parpadeo en la pantalla que puede llegar a ser molesto.

Para evitar el uso de `setInterval` podemos hacer uso del método `interval(x ms)` que crea un `Observable` cada `x` milisegundos y con `switchMap` nos aseguramos de emitir solo una petición cada vez.

El resultado sería:

```
1 this.users$ = Observable.interval(2000).switchMap(() => {  
2     return this.proxy.getUsers()  
3 });
```

De esta forma los valores se actualizan pero no se produce el parpadeo por pantalla.

Combinar datos de dos llamadas distintas

Lo ideal es que los datos que necesitemos nos los proporcione el back con un única llamada ya sea via REST o el cada vez más utilizado GraphQL. La explicación es sencilla los servidores de back están mucho mejor preparados para realizar cálculos complejos y combinar datos que no el navegador del cliente.

Pero claro esto es lo ideal, es muchas ocasiones para pintar la información por pantalla en un mismo componente tenemos que llamar a más de un endpoint para recibir los datos deseados. Vamos a poner un ejemplo donde queremos mostrar información de GitHub de ciertos usuarios que obtenemos con en el endpoint `"/api/public/users"` mientras que la información de GitHub asociada la obtenemos del endpoint `"/api/public/users-github/nombre_usuario"` y tenemos un equipo de back muy testarudo que no quiere hacer esta combinación por nosotros.

En este caso tenemos que recurrir a la programación reactiva para solucionar el problema. La función podría quedar de esta forma:

```
1 getUsersWithMoreInfo(): Observable<any[]> {  
2     return this.http.get('/api/public/users').pipe(  
3         mergeMap((users: Array<any>) => {  
4             if (users.length > 0) {  
5                 return forkJoin(  
6                     users.map((user: any) => {  
7                         return this.http.get(`/api/public/users-github/${user.username}`).  
8                     pipe(  
9                         map((info: any) => {  
10                             user.info = info;  
11                             return user;  
12                         })  
9                     )  
10                 )  
11             }  
12         })  
13     )  
14 }
```



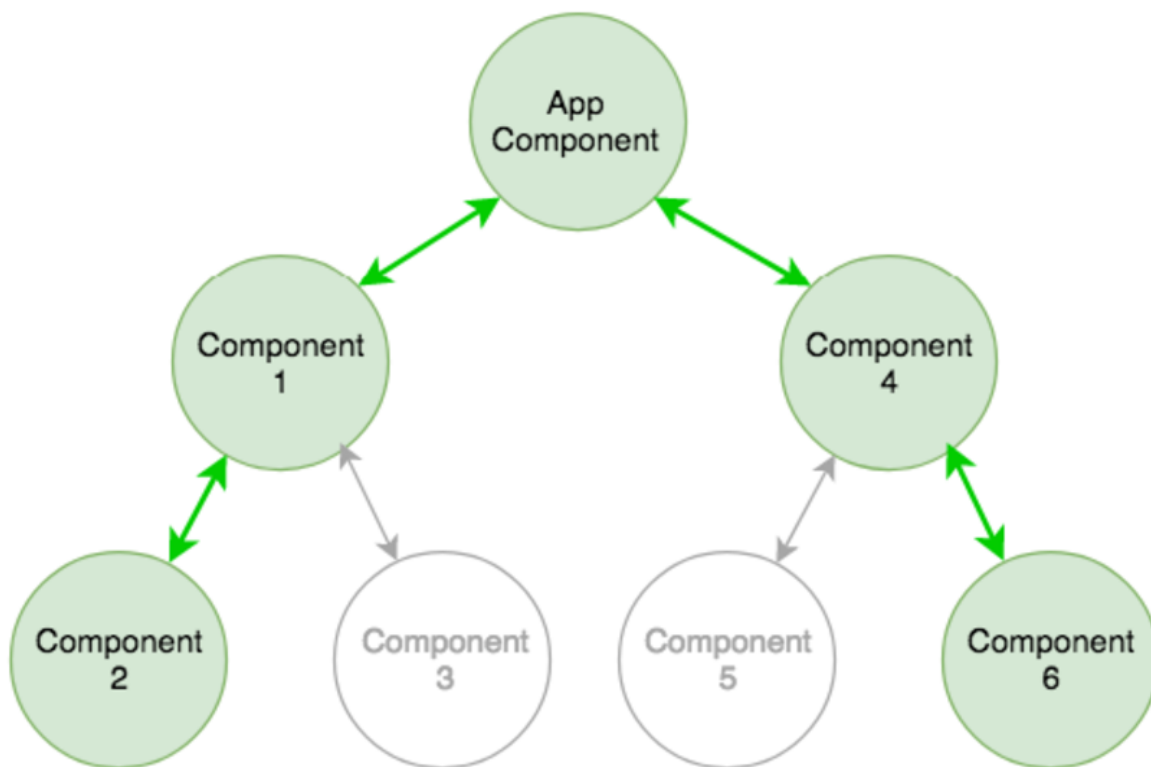
```
12         })
13     )
14 })
15 )
16 }
17     return of([]);
18 },
19 )
20 }
```

Como ves lo primero que hacemos es hacer la llamada para recuperar el listado de los usuarios que queremos mostrar por pantalla; después con el operador `mergeMap` nos quedamos con el array que devuelve la llamada y hacemos un `forkJoin` con cada una de las llamadas que hacemos para recuperar la información de github, donde el resultado lo incorporamos a una nueva propiedad “info” del objeto “user”. En caso de no tener usuarios, simplemente, devolvemos un `Observable` con un array vacío.

Es importante darse cuenta de que `user.map` es la forma de recorrer el array de usuarios que devuelve la primera llamada, no es el operador `map` de la programación reactiva.

Gestión del estado

El estado de una aplicación es la foto de los datos que maneja la aplicación en un determinado momento. Por defecto, cada componente en Angular mantiene su estado definido en las propiedades de la clase y cuando quiere transmitir información de su estado a otros componentes entonces hace uso de `@Input` o `@Output`. El problema es que está limitado al paso entre padres e hijos, por lo que comunicar el estado a un componente que se encuentra bajo el router o en otra jerarquía del árbol resulta tedioso y complicado. Imagina que en esta estructura el componente 2 quiere transmitir el estado al componente 6. Si seguimos esta aproximación esto supone implementar 4 saltos.



Para resolver esta situación podemos implementar otras aproximaciones:

- **LocalStorage / SessionStorage:** podemos almacenar el estado en el navegador para que pueda ser leído y modificado desde cualquier parte del árbol. Esto presenta serios problemas de seguridad ya que cualquier dato en el Local o Session storage puede ser fácilmente cambiado por el usuario. Además de que los datos

se almacenan en plano por lo que hay que tener mucho cuidado de no almacenar información sensible como números de cuenta o contraseñas.

- **Servicio Singleton:** podemos almacenar el estado en las propiedades de una clase que definamos como provider dentro del AppModule. De esta forma estará disponible desde cualquier punto del árbol para modificar o leer el estado, aunque esta aproximación sólo sería válida para aplicaciones pequeñas y hay que tener en cuenta que estamos jugando con la mutabilidad del servicio lo que podría provocar efectos colaterales.
- **BehaviorSubject:** otra alternativa que ya hemos visto es hacer uso de la programación reactiva y definir un BehaviorSubject que almacene el estado. Además esta clase en RxJS 5 ya cuenta con un método `getValue()` o simplemente `.value` que devuelve el último estado almacenado; y cuando cambiamos el estado se emite a todos los suscriptores.

Model Pattern

Se trata de un patrón que nos sirve para generalizar el uso de BehaviorSubject para el almacenamiento del estado con el uso de genéricos proporcionados por TypeScript.

El patrón podemos implementarlo nosotros en una única clase o importar la dependencia `ngx-model` que ya proporciona una implementación con todo lo necesario.

```
1 $> npm install --save ngx-model
```

Todos los detalles de esta implementación los encontramos en la [página del autor](#)²⁴

Consta de un único fichero de 1.07 Kb que haciendo uso de la programación reactiva y más concretamente del BehaviorSubject permite mantener el estado de la aplicación de una forma que a mi parecer es más intuitiva y que se adapta mejor a la forma de implementar las aplicaciones con Angular que otras librerías más complejas como `ngRx`. De una forma sencilla permite la instanciación de clases de modelo ajustadas a las interfaces creadas para la representación del modelo de nuestro dominio.

El código de dicho fichero se muestra a continuación:

²⁴<https://tomastrajan.github.io/angular-model-pattern-example#/about>

```
1  import { BehaviorSubject } from 'rxjs/BehaviorSubject';
2  import { Observable } from 'rxjs/Observable';
3  import { map } from 'rxjs/operators/map';
4
5  export class Model<T> {
6      private _data: BehaviorSubject<T>;
7
8      data$: Observable<T>;
9
10     constructor(initialData: any, immutable: boolean, clone?: (data: T) => T) {
11         this._data = new BehaviorSubject(initialData);
12         this.data$ = this._data
13             .asObservable()
14             .pipe(
15                 map(
16                     data =>
17                         immutable
18                             ? clone ? clone(data) : JSON.parse(JSON.stringify(data))
19                             : data
20                 )
21             );
22     }
23
24     get(): T {
25         return this._data.getValue();
26     }
27
28     set(data: T) {
29         this._data.next(data);
30     }
31 }
32
33 export class ModelFactory<T> {
34     create(initialData: T): Model<T> {
35         return new Model<T>(initialData, true);
36     }
37
38     createMutable(initialData: T): Model<T> {
39         return new Model<T>(initialData, false);
40     }
41
42     createWithCustomClone(initialData: T, clone: (data: T) => T) {
```

```
43     return new Model<T>(initialData, true, clone);
44   }
45 }
46
47 export function useModelFactory() {
48   return new ModelFactory();
49 }
50
51 export const MODEL_PROVIDER = {
52   provide: ModelFactory,
53   useFactory: useModelFactory
54 };
```

Como ves no es nada que no podamos haber implementado nosotros mismos, de hecho el propio autor dice que si no queremos añadir la dependencia, simplemente creamos el fichero con el contenido mostrado arriba. Lo que hace es a través de genéricos generalizar la creación de BehaviorSubject en base al tipo de datos que necesitemos, proporcionando métodos para la consulta y modificación de los datos del modelo y mejoras para hacer el clonado de los objetos y mantener la inmutabilidad.

De esta forma podemos instanciar en cualquier módulo el provider:

```
1 @NgModule({
2   providers: [MODEL_PROVIDER]
3 })
```

o si hacemos uso de la librería

```
1 import { NgxModelModule } from 'ngx-model';
2
3 @NgModule({
4   imports: [NgxModelModule]
5 })
```

Y ya podemos hacer uso de esta clase de instanciación de modelo. Por ejemplo, si queremos almacenar el estado de un usuario, podríamos crear la interfaz con sus datos:

```
1 export interface User {  
2   login: string;  
3   name: string;  
4   avatar: string;  
5   admin: boolean;  
6 }
```

Y crear un servicio de dominio donde utilizar esta interfaz para instanciar el modelo de User e implementar todos los métodos que cambian y consultan los datos del modelo.

```
1 import { Observable } from 'rxjs/Observable';  
2 import { Injectable } from '@angular/core';  
3 import { Model, ModelFactory } from 'ngx-model';  
4  
5 import { User } from '../list-users/user';  
6  
7 @Injectable()  
8 export class CurrentUserModelService {  
9  
10   private model: Model<User>;  
11   user$: Observable<User>;  
12  
13   constructor(private modelFactory: ModelFactory<User>) {  
14     this.model = this.modelFactory.create({});  
15     this.user$ = this.model.data$;  
16   }  
17  
18   setUser(newUser: User) {  
19     this.model.set(newUser);  
20   }  
21  
22   getUser(): User {  
23     return this.model.get();  
24   }  
25  
26 }
```

Ahora en los componentes o servicios que se quiera consultar/modificar el estado del usuario solo se tiene que inyectar el servicio CurrentUserModelService y hacer uso de los métodos creados. A continuación vemos un ejemplo de componente que consulta los datos para mostrarlos por pantalla:

```
1  @Component({
2      selector: 'app-user',
3      template: `
4          <div *ngIf="user$ | async as user">
5              <p>{{user.login}}</p>
6              <p>{{user.name}}</p>
7              <p>{{user.admin}}</p>
8              <img [src]="user.avatar">
9          </div>
10     `,
11 })
12 export class UserComponent implements OnInit {
13
14     user$: Observable<User>;
15
16     constructor(private userService: CurrentUserModelService) {}
17
18     ngOnInit() {
19         this.user$ = this.userService.user$;
20     }
21
22     onClick(user: User) {
23         this.userService.setUser(user);
24     }
25
26 }
```

Fíjate que estamos haciendo uso del observable para suscribirnos dentro del método `ngOnInit` a los cambios que se produzcan en el objeto, de forma que si otro componente hiciera uso del método `setUser` para modificar la información, inmediatamente este componente mostraría esa información de forma reactiva, es decir, sin tener que refrescar la aplicación. Y si hubiera más de un componente suscrito, toda la información cambiaría al mismo tiempo.

Creación de una librería para Angular

Seguro que a estas alturas ya estáis cansados de escucharme decir que una de las grandezas de Angular es que es altamente reutilizable a través de sus módulos y que podemos encapsularlos en librerías para poder ser utilizadas en otros proyectos de Angular.

Mi consejo es que esta librería tenga el mínimo código de HTML y CSS posible y que se centre en la lógica que quiera resolver y una interfaz “fea” de pruebas.

De esta forma podemos implementar una librería con toda la lógica perfectamente testeada y que podremos reutilizar en proyectos de naturaleza Angular como Ionic y con ciertas restricciones también en NativeScript.

Con esto marcamos la separación entre la lógica y la forma de presentar la solución haciendo que cada rol (desarrollador/arquitecto vs diseñador/ux) invierta el tiempo en lo que realmente es productivo.

Ya os digo yo que cuando tengo que tocar CSS y HTML para que quede “bonito” me vuelvo altamente improductivo.

Usando @angular/cli

La forma más cómoda es seguir utilizando la misma herramienta que utilizamos para la implementación de aplicaciones con Angular.

El caso es que a día de hoy no existe un soporte oficial del CLI para la creación de librerías; así que nos tenemos que apoyar en un proyecto llamado “packagr” y seguir los siguientes pasos.

Creación de la librería desde cero

Para la creación de la librería simplemente creamos un nuevo proyecto con el comando ng.

```
1 $> ng new my-lib-poc
```

Una vez creado el proyecto podemos abrirlo con el editor de texto y crear un módulo secundario como ya sabemos con la lógica que queramos compartir, en este caso, lo vamos a simplificar al máximo, porque el objetivo es crear una librería no importa tanto el contenido de la misma.


```
1 $> npm run ng -- g module header
```

Vamos a crear un componente asociado al módulo:

```
1 $> npm run ng -- g component header/header
```

Y un servicio también asociado al módulo:

```
1 $> npm run ng -- g service header/header
```

Como ves no perdemos las ventajas de trabajar con CLI.

Ahora editamos el fichero header.service.ts para añadir un método que devuelva el texto del header.

```
1 export class HeaderService {  
2  
3   constructor() { }  
4  
5   getHeader(): string {  
6     return 'Header service';  
7   }  
8  
9 }
```

Este servicio lo instanciamos en el componente, para ello editamos el fichero header.component.ts y establecemos el valor en una propiedad del componente.

```
1 import { HeaderService } from './header.service';  
2 import { Component, OnInit } from '@angular/core';  
3  
4 @Component({  
5   selector: 'app-header',  
6   templateUrl: './header.component.html',  
7   styleUrls: ['./header.component.css']  
8 })  
9 export class HeaderComponent implements OnInit {  
10  
11   header: string;  
12 }
```

```
13   constructor(private service: HeaderService) { }
14
15   ngOnInit() {
16     this.header = this.service.getHeader();
17   }
18
19 }
```

Ahora editamos el HTML asociado para interpolar esta propiedad:

```
1 <h1>
2   {{header}}
3 </h1>
```

Y editamos el fichero header.module.ts para permitir a los usuarios de nuestra librería poder hacer uso del componente HeaderComponent, declarándolo en el array de “exports” y establecemos en el array de “providers” el servicio para poder hacer uso de él de forma interna. En caso de permitir hacer uso del servicio de forma externa tenemos que declarar la función forRoot(), como vemos en el siguiente código:

```
1 import { HeaderService } from './header.service';
2 import { ModuleWithProviders, NgModule } from '@angular/core';
3 import { CommonModule } from '@angular/common';
4 import { HeaderComponent } from './header.component';
5
6 @NgModule({
7   imports: [
8     CommonModule
9   ],
10  declarations: [HeaderComponent],
11  exports: [HeaderComponent],
12  providers: [HeaderService]
13 })
14 export class HeaderModule {
15   public static forRoot(): ModuleWithProviders {
16     return {
17       ngModule: HeaderModule,
18       providers: [
19         HeaderService
20       ]
21     };
22   }
23 }
```

Nuestra librería tiene implementada toda la funcionalidad que queremos compartir, podemos hacer uso del componente en el fichero `app.component.html` y arrancar la aplicación para verificarlo.

Uso de packagr para la distribución

Este es el momento de empaquetar nuestra librería y para ello vamos a instalar la siguiente dependencia:

```
1 $> npm install --save-dev ng-packagr
```

Esta librería necesita que creamos dos ficheros en la raíz del proyecto: el primero llamado “`ng-package.json`” donde indicamos el esquema que tiene que utilizar la librería y donde se encuentra el segundo de los ficheros:

```
1 {  
2   "$schema": "./node_modules/ng-packagr/ng-package.schema.json",  
3   "lib": {  
4     "entryFile": "public_api.ts"  
5   }  
6 }
```

El segundo fichero “`public_api.ts`” define todos los exports de nuestra librería y se utiliza para generar los `.d.ts` adecuados.

```
1 export * from './src/app/header/header.module';  
2 export * from './src/app/header/header.service';  
3 export * from './src/app/header/header.component';
```

Ahora editamos el fichero `package.json` para poner la propiedad `private` a `false`, ya que se quiere publicar en algún repositorio de npm, y añadimos en la sección de scripts un “task” de npm para ejecutar la herramienta de empaquetado. Además todas las dependencias del proyecto las ponemos como `peerDependencies` para que sean tenidas en cuenta por el proyecto que vaya a utilizar nuestra librería:

```
1  ...
2  "scripts": {
3    "ng": "ng",
4    "start": "ng serve",
5    "build": "ng build",
6    "test": "ng test",
7    "lint": "ng lint",
8    "e2e": "ng e2e",
9    "packagr": "ng-packagr -p ng-package.json"
10 },
11 "private": false,
12 "peerDependencies": {
13   "@angular/animations": "^5.0.0",
14   "@angular/common": "^5.0.0",
15   "@angular/compiler": "^5.0.0",
16   "@angular/core": "^5.0.0",
17   "@angular/forms": "^5.0.0",
18   "@angular/http": "^5.0.0",
19   "@angular/platform-browser": "^5.0.0",
20   "@angular/platform-browser-dynamic": "^5.0.0",
21   "@angular/router": "^5.0.0",
22   "core-js": "^2.4.1",
23   "rxjs": "^5.5.2",
24   "zone.js": "^0.8.14"
25 },
26 ...
```

Ahora simplemente ejecutamos:

```
1 $> npm run packagr
```

Y si el proceso es correcto nos generará una carpeta dist con el contenido de nuestra librería listo para publicar y preparado para soportar AOT; además genera los ficheros d.ts del API, los bundles UMD para su ejecución con SystemJS y los ficheros en es5 necesarios para su ejecución en el navegador.

Ahora solo tenemos que entrar dentro de la carpeta dist y ejecutar “npm publish” con los permisos necesarios en el repositorio corporativo o público que tengamos configurado y la librería ya puede ser consumida por nuestros usuarios, simplemente ejecutando en sus proyectos:

```
1 $> npm install --save my-lib-poc
```

Server rendering

Introducción

Una de las cosas que tenemos que tener muy en cuenta cuando desarrollamos una aplicación SPA con la tecnología que sea es que no es muy “seo friendly”, dado que requiere de una carga inicial de la aplicación (el típico loading...) que hace que las arañas no puedan ver el contenido de la aplicación y no puedan indexar correctamente la página.

Para resolver este problema muchas tecnologías optan por el “server rendering” o renderizado de páginas en el servidor, lo que quiere decir que al cliente que solicita la información se le sirve HTML estático. Hasta ahora en Angular no era algo trivial, y digo hasta ahora porque se ha creado el siguiente repositorio de angular-cli con la solución, tanto de server rendering como prerendering pudiendo hacer uso del lazy loading de módulos:

<https://github.com/angular/universal-starter/tree/master/cli>²⁵

Así que si estás a punto de empezar un proyecto con requerimiento de SEO te aconsejo que te bases directamente en este repositorio y si lo que quieres es adaptar un proyecto ya existente, sigue leyendo que a continuación te explico los pasos a seguir partiendo del repositorio anterior.

Pasos para hacer server rendering

Así que ya tienes una aplicación con Angular en producción y alguien de negocio se ha acordado ahora del SEO porque pensaba que esto de Angular ya lo traía por defecto... bueno no te preocupes y sigue estos pasos.

1. Vamos a instalar las dependencias necesarias:

²⁵<https://github.com/angular/universal-starter/tree/master/cli>

```
1 $> npm install --save @angular/platform-server
2 $> npm install --save @nguniversal/express-engine
3 $> npm install --save @nguniversal/module-map-ngfactory-loader
4 $> npm install --save-dev cpy-cli
```

Las tres primeras son necesarias para la implementación del servidor que va a renderizar las páginas, y la última la vamos a utilizar en la fase de build.

1. Creamos el fichero server.js en la raíz del proyecto con este contenido:

```
1 require('zone.js/dist/zone-node');
2 require('reflect-metadata');
3 const express = require('express');
4 const fs = require('fs');
5
6 const { platformServer, renderModuleFactory } = require('@angular/platform-server');
7 r');
8 const { ngExpressEngine } = require('@nguniversal/express-engine');
9 // Import module map for lazy loading
10 const { provideModuleMap } = require('@nguniversal/module-map-ngfactory-loader');
11
12 // Import the AOT compiled factory for your AppServerModule.
13 // This import will change with the hash of your built server bundle.
14 const { AppServerModuleNgFactory, LAZY_MODULE_MAP } = require(`./dist-server/main.bundle`);
15
16
17 const app = express();
18 const port = 8000;
19 const baseUrl = `http://localhost:${port}`;
20
21 // Set the engine
22 app.engine('html', ngExpressEngine({
23   bootstrap: AppServerModuleNgFactory,
24   providers: [
25     provideModuleMap(LAZY_MODULE_MAP)
26   ]
27 }));
28
29 app.set('view engine', 'html');
30
31 app.set('views', './');
```

```

32 app.use('/', express.static('./', {index: false}));
33
34 app.get('*', (req, res) => {
35   res.render('index', {
36     req,
37     res
38   });
39 });
40
41 app.listen(port, () => {
42   console.log(`Listening at ${baseUrl}`);
43 });

```

Es el fichero que va a levantar nuestro servidor de NodeJS en el puerto que especifiquemos en la constante “port”. A destacar el uso LAZY_MODULE_MAP para soportar la carga lazy de módulos secundarios.

1. Creamos el fichero src/app/app.server.module.ts con el siguiente contenido:

```

1  import {NgModule} from '@angular/core';
2  import {ServerModule} from '@angular/platform-server';
3  import {ModuleMapLoaderModule} from '@nguniversal/module-map-ngfactory-loader';
4
5  import {AppModule} from './app.module';
6  import {AppComponent} from './app.component';
7
8  @NgModule({
9    imports: [
10     // The AppServerModule should import your AppModule followed
11     // by the ServerModule from @angular/platform-server.
12     AppModule,
13     ServerModule,
14     ModuleMapLoaderModule,
15   ],
16   // Since the bootstrapped component is not inherited from your
17   // imported AppModule, it needs to be repeated here.
18   bootstrap: [AppComponent],
19 })
20 export class AppServerModule {}

```

En este fichero estamos importando el módulo principal de nuestra aplicación (AppModule), el módulo para nuestro servidor (ServerModule) y el módulo que permite el “lazy

loading” (ModuleMapLoaderModule). Además establecemos en la propiedad bootstrap cuál es nuestro componente principal (AppComponent)

1. Creamos el fichero src/tsconfig.server.json con el siguiente contenido:

```
1 {
2   "extends": "../tsconfig.json",
3   "compilerOptions": {
4     "outDir": "../out-tsc/app",
5     "baseUrl": "./",
6     // Set the module format to "commonjs":
7     "module": "commonjs",
8     "types": []
9   },
10  "exclude": [
11    "test.ts",
12    "**/*.spec.ts"
13  ],
14  // Add "angularCompilerOptions" with the AppServerModule you wrote
15  // set as the "entryModule".
16  "angularCompilerOptions": {
17    "entryModule": "app/app.server.module#AppServerModule"
18  }
19 }
```

Fíjate como en las opciones de compilación de Angular le especificamos la ruta del módulo creado en el paso anterior.

1. Creamos el fichero main.server.ts con el siguiente contenido:

```
1 import { environment } from '../environments/environment';
2 import { enableProdMode } from '@angular/core';
3
4 if (environment.production) {
5   enableProdMode();
6 }
7
8 export {AppServerModule} from './app/app.server.module';
```

Aquí indicamos el export a la clase AppServerModule que creamos anteriormente.

1. Editamos el fichero `app.module.ts` para hacerlo compatible con Universal añadiendo la función `.withServerTransition()` y especificando un ID de aplicación, que puede ser el que se nos ocurra. En la práctica es dejar el fichero tal cual lo tengas y añadir al módulo `BrowserModule` lo siguiente:

```
1 BrowserModule.withServerTransition({appId: 'angular-app'}),
```

1. Añadimos una nueva app llamada “server” en `.angular-cli` añadiendo lo siguiente a la propiedad “apps”

```
1  ...
2  ,
3    {
4      "name": "server",
5      "platform": "server",
6      "root": "src",
7      "outDir": "dist/dist-server",
8      "assets": [
9        "assets",
10       "favicon.ico"
11     ],
12     "index": "index.html",
13     "main": "main.server.ts",
14     "test": "test.ts",
15     "tsconfig": "tsconfig.server.json",
16     "testTsconfig": "tsconfig.spec.json",
17     "prefix": "app",
18     "styles": [
19       "styles.css"
20     ],
21     "scripts": [],
22     "environmentSource": "environments/environment.ts",
23     "environments": {
24       "dev": "environments/environment.ts",
25       "prod": "environments/environment.prod.ts"
26     }
27   }
28  ...
```

Este paso es muy importante y nos permite tener los dos tipos de aplicaciones en un mismo proyecto, el “normal” para desarrollar del modo común y el “server” para que nuestra aplicación se pueda servir desde el servidor en vez de renderizarse en cliente.

Especificamos un nuevo directorio de salida y le indicamos que haga uso de los ficheros que hemos ido creando: `main.server.ts` y `tsconfig.server.json`

1. Añadimos nuevos script al fichero `package.json`

```
1 {  
2 ...  
3 "build:universal": "ng build --prod && ng build --prod --app 'server' --output-h\  
4 ashing=false && cpy ./server.js ./dist",  
5 "serve:universal": "npm run build:universal && cd dist && node server"  
6 ...  
7 }
```

1. Probamos el resultado

Hechas estas configuraciones es momento de arrancar nuestra aplicación para comprobar el resultado. Para ello ejecutamos:

```
1 $> npm run serve:universal
```

Terminado el proceso nos dirá que la aplicación está corriendo el puerto 8000 o el que hayamos especificado en la constante `port` del fichero `server.js`

Nos conectamos a esta URL y el efecto más inmediato de que todo ha ido bien es que no vemos el típico “Loading...” (o el contenido que tengamos entre las etiquetas del selector principal en el `index.html`) cuando la aplicación carga, ni tan siquiera cuando forzamos un refresco de pantalla. Además si cargamos módulos secundarios con “lazy loading” tenemos que ver que solo se descargan cuando se solicitan.

Añadimos pre-rendering

Otra de las técnicas que podemos implementar para mejorar el SEO de nuestras aplicaciones Angular es el pre-rendering, que se diferencia del server rendering en que el renderizado se hace en tiempo de compilación y por tanto no es necesario un servidor NodeJS que atienda las peticiones, nos vale con un servidor normal como `nginx`, `apache`, `github pages`, etc...

Para poder hacer también pre-rendering de nuestra aplicación tenemos que crear el fichero `prerender.js` en la raíz del proyecto con el siguiente contenido:

```

1  // Load zone.js for the server.
2  require('zone.js/dist/zone-node');
3  require('reflect-metadata')
4  const fs = require('fs');
5
6  // Import renderModuleFactory from @angular/platform-server.
7  const { renderModuleFactory } = require('@angular/platform-server');
8
9  // Import module map for lazy loading
10 const { provideModuleMap } = require('@nguniversal/module-map-ngfactory-loader');
11
12 // Import the AOT compiled factory for your AppServerModule.
13 // This import will change with the hash of your built server bundle.
14 const { AppServerModuleNgFactory, LAZY_MODULE_MAP } = require(`./dist/dist-server\
15 r/main.bundle`);
16
17 // Load the index.html file containing references to your application bundle.
18 const index = fs.readFileSync('./dist/index.html', 'utf8');
19
20 // Writes rendered HTML to ./dist/index.html, replacing the file if it already e\
21 xists.
22 renderModuleFactory(AppServerModuleNgFactory, {
23   document: index,
24   url: '/',
25   extraProviders: [
26     provideModuleMap(LAZY_MODULE_MAP)
27   ]
28 })
29 .then(html => fs.writeFileSync('./dist/index.html', html));

```

y añadir los siguientes scripts al fichero package.json:

```

1  "build:prerender": "ng build --prod && ng build --prod --app 'server' --output-h\
2  ashing=false && node prerender",
3  "serve:prerender": "npm run build:prerender && cd dist && live-server --entry-fi\
4  le=index.html",

```

Antes de ejecutar estos scripts vamos a instalar la dependencia live-server que nos permite levantar la aplicación y establecer el punto de entrada en el index.html para que al refrescar no perdamos la aplicación.

```
1 $> npm install --save-dev live-server
```

Ahora podemos desplegar la aplicación con prerender con el comando:

```
1 $> npm run serve:prerender
```

O crear los ficheros de distribución con el comando:

```
1 $> npm run build:prerender
```

En ambos casos al levantar la aplicación tenemos que ver el mismo efecto de que no se muestre el contenido que tengamos entre las etiquetas del componente principal de la aplicación.

Conclusión

Como ves uno de los puntos menos fuertes de las SPAs el equipo de Angular lo ha solucionado de una forma elegante y casi transparente al desarrollador. Seguro que en futuras versiones de angular-cli pondrán el típico flag para ejecutar todos estos pasos de configuración de forma automática.

Así que ahora cuando te planteen el tema del SEO con una aplicación de Angular ya no te tienes que echar a temblar y ya puedes decir, “por supuesto el framework lo soporta” :-)

Además podemos hacer “deep linking” con las URLs amigables de la aplicación dado que accederán directamente al contenido sin hacer la carga inicial.

Internacionalización

En un mundo cada vez más globalizado es muy frecuente encontrarse con un requisito no funcional relativo al multi-idioma de la aplicación.

En esta sección vamos a explicar como conseguirlo en nuestras aplicaciones Angular/Ionic.

Haciendo uso del core de Angular

Esta es la forma que actualmente recomienda el equipo de Angular para hacerlo lo más compatible posible con Server Rendering. La pega es que se tienen que realizar una compilación por cada idioma que se quiera soportar. A continuación se describen los pasos.

Indicar los textos que se quieren internacionalizar

Para marcar un texto que se quiere internacionalizar utilizamos la palabra reservada “i18n”, de esta forma:

```
1 <h1 i18n>Bienvenido {{nombre}}</h1>
```

No se trata de una directiva es un custom attribute que utiliza el compilador para saber que texto tiene que internacionalizar, no aparece en el código en producción.

Esto es lo único que necesitamos para marcar que un texto tiene que ser internacionalizado, podemos añadir más información que ayude al traductor a saber el contexto de lo que tiene que traducir, añadiendo la descripción de un significado, de esta forma:

```
1 <h1 i18n="saludo|saludo de bienvenida@@saludo">
2   Bienvenido {{nombre}}!
3 </h1>
```

Ninguno de estos parámetros es obligatorio pero sí son muy recomendables para facilitar el mantenimiento de los ficheros de traducción.

También podemos querer internacionalizar ciertos atributos HTML como el título de imagen para favorecer la accesibilidad, para ello hacemos uso de la palabra reservada i18n seguida del atributo HTML, dejando el atributo HTML original con el texto en el idioma que consideremos por defecto.

```

1 <img src="" title="Imagen vacía" i18n-title="imagen|imagen titulo vacia|@@imagen\
2 Titulo" />

```

En el caso de tener que aplicar un texto u otro en función de la cardinalidad de una determinada variable, podemos hacer uso de `i18n` en el formato [ICU Message Format](#)²⁶, esta sería la forma de indicar distintos mensajes internacionalizados en función de la longitud de una variable “mensajes” que tiene que existir en la lógica de componente:

```

1 <span i18n="mensajes|mensajes@@mensajes">{mensajes.length, plural, =0 {'no h\
2 ay mensajes'} =1 {'hay un mensaje'} =2 {'hay dos mensajes'} other {'hay {{mensaj\
3 es.length}} mensajes'}}</span>

```

Si lo que queremos es poner un mensaje distinto en función de una condición, por ejemplo, el género de una persona, se sigue el estándar [ICU message syntax](#)²⁷, este sería un ejemplo:

```

1 <span i18n="genero persona|genero persona@@generoPersona">
2     La persona es {genero, select, hombre {un hombre} mujer {una mujer}}
3 </span>

```

El género es un atributo del componente de tipo string que puede tener el valor “hombre” o “mujer”.

Por último, podemos tener una situación en la que mezclar los dos formatos, es decir, si dentro de los mensajes de cardinalidad necesitamos ser específicos en base a una condición. Este sería un ejemplo, tenemos una variable “forma” donde 1 es triángulo y 2 es cuadrado, y queremos mostrar el texto en base a la variable “número”:

```

1 <span i18n="contar formas|contar formas@@contarFormas">
2     {numero, plural,
3       =0 {no hay ningún {forma, select, 1 {triángulo} 2 {cuadrado}}}
4       =1 {hay un {forma, select, 1 {triángulo} 2 {cuadrado}}}
5       other {hay {{numero}} {forma, select, 1 {triángulo} 2 {cuadrado}}}
6     }
7 </span>

```

Extracción a fichero de mensajes

Marcar los textos de esta forma nos sirve para especificar que textos queremos extraer. Para extraerlos angular-cli nos proporciona la herramienta “ng-xi18n” donde especificamos el formato de fichero y el nombre del fichero a generar. Podríamos establecerlo como un nuevo script en la sección de “scripts” del fichero package.json:

²⁶<http://userguide.icu-project.org/formatparse/messages>

²⁷<http://userguide.icu-project.org/formatparse/messages>

```

1  scripts: {
2      "i18n": "ng-xi18n --i18nFormat=xliff --outFile=messages.xliff"
3  }

```

Ahora simplemente ejecutamos el comando:

```
1  $> npm run i18n
```

Lo que nos generará la plantilla de mensajes en base a los textos que hayamos especificado, esta sería la plantilla asociada a los ejemplos anteriores:

```

1  <?xml version="1.0" encoding="UTF-8" ?>
2  <xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
3      <file source-language="es" datatype="plaintext" original="ng2.template">
4          <body>
5              <trans-unit id="saludo" datatype="html">
6                  <source>
7                      Bienvenido <x id="INTERPOLATION"/>!
8                  </source>
9                  <context-group purpose="location">
10                     <context context-type="sourcefile">src/app/app.component.ts</context>
11                     <context context-type="linenumber">3</context>
12                 </context-group>
13                 <note priority="1" from="description">saludo de bienvenida</note>
14                 <note priority="1" from="meaning">saludo</note>
15             </trans-unit>
16             <trans-unit id="imagenTitulo" datatype="html">
17                 <source>Imagen vacía</source>
18                 <context-group purpose="location">
19                     <context context-type="sourcefile">src/app/app.component.ts</context>
20                     <context context-type="linenumber">7</context>
21                 </context-group>
22                 <note priority="1" from="description">imagen titulo vacia|</note>
23                 <note priority="1" from="meaning">imagen</note>
24             </trans-unit>
25             <trans-unit id="mensajes" datatype="html">
26                 <source>{VAR_PLURAL, plural, =0 {&apos;no hay mensajes&apos;} =1 {&apos;\
27 hay un mensaje&apos;} =2 {&apos;hay dos mensajes&apos;} other {&apos;hay <x id="\
28 INTERPOLATION"/> mensajes&apos;} }</source>
29                 <context-group purpose="location">
30                     <context context-type="sourcefile">src/app/app.component.ts</context>
31                     <context context-type="linenumber">10</context>

```



```

32     </context-group>
33     <note priority="1" from="description">mensajes</note>
34     <note priority="1" from="meaning">mensajes</note>
35 </trans-unit>
36 <trans-unit id="generoPersona" datatype="html">
37     <source>
38     La persona es <x id="ICU"/>
39 </source>
40     <context-group purpose="location">
41         <context context-type="sourcefile">src/app/app.component.ts</context>
42         <context context-type="linenumber">14</context>
43     </context-group>
44     <note priority="1" from="description">genero persona</note>
45     <note priority="1" from="meaning">genero persona</note>
46 </trans-unit>
47 <trans-unit id="81c74c1ef45d94ffcf51e6e6983ef4dd0a784b05" datatype="html">
48     <source>{VAR_SELECT, select, hombre {un hombre} mujer {una mujer} }</sou\
49 rce>
50     <context-group purpose="location">
51         <context context-type="sourcefile">src/app/app.component.ts</context>
52         <context context-type="linenumber">15</context>
53     </context-group>
54 </trans-unit>
55 <trans-unit id="contarFormas" datatype="html">
56     <source>
57     <x id="ICU"/>
58 </source>
59     <context-group purpose="location">
60         <context context-type="sourcefile">src/app/app.component.ts</context>
61         <context context-type="linenumber">19</context>
62     </context-group>
63     <note priority="1" from="description">contar formas</note>
64     <note priority="1" from="meaning">contar formas</note>
65 </trans-unit>
66 <trans-unit id="40b6b1bf1634bde7e9e2611d6d5b23dab9a2f6af" datatype="html">
67     <source>{VAR_PLURAL, plural, =0 {no hay ningún {VAR_SELECT, select, 1 {t\
68 riángulo} 2 {cuadrado} }} =1 {hay un {VAR_SELECT_1, select, 1 {triángulo} 2 {cua\
69 drado} }} other {hay <x id="INTERPOLATION"/> {VAR_SELECT_2, select, 1 {triángulo\
70 } 2 {cuadrado} }} }</source>
71     <context-group purpose="location">
72         <context context-type="sourcefile">src/app/app.component.ts</context>
73         <context context-type="linenumber">20</context>

```

```
74         </context-group>
75     </trans-unit>
76 </body>
77 </file>
78 </xliff>
```

Creación de los ficheros de mensajes por idioma

Como se ha dicho esto es solo la plantilla en la que apoyarnos para ir creando el resto de ficheros en función de los idiomas que queramos soportar. Ahora creamos un directorio “locale” en el raíz del proyecto donde incluimos una copia del fichero messages.xlf

Lo normal es que este fichero lo dejemos para el idioma por defecto que vayamos a soportar, y que para el resto de idiomas usemos el patrón messages.\$locale.xlf, por ejemplo, para el inglés sería, messages.en.xlf.

Si abrimos el fichero messages.xlf veremos que todas las marcas puestas responden a una determinada estructura donde el texto original está entre las etiquetas “source”, como vemos en este ejemplo:

```
1 <source>
2     Bienvenido <x id="INTERPOLATION"/>!
3 </source>
```

Ahora hacemos una copia llamando a la etiqueta “target” y estableciendo el texto en el idioma que marque el fichero en el que estamos escribiendo, por ejemplo en alemán:

```
1 <target>
2     Willkommen <x id="INTERPOLATION"/>!
3 </target>
```

Puesta en producción

Nosotros podemos ir desarrollando la aplicación con el idioma por defecto sin preocuparnos por más que anotar con “i18n” lo que queramos internacionalizar, e ir creando en paralelo los ficheros de idiomas que queramos soportar como ya hemos visto.

A la hora de poner la aplicación multi-idioma en producción tenemos que tener en cuenta que serán tantas aplicaciones como idiomas vayamos a soportar por lo que tenemos que ejecutar la “build” indicando el idioma y el fichero de mensaje asociado de esta forma:

```
1 $> npm run build -- --aot --i18nFile=./locale/messages.en.xlf --locale=en --i18n\
2 Format=xlf
```

Dentro del servidor en el que vayamos a desplegar la aplicación tenemos que crear tantas carpetas como idiomas soportemos e incluir en cada de ellas su correspondiente resultado de la build (el contenido de la carpeta dist), de forma que podamos navegar de una aplicación a otra cuando se cambie el idioma.

Haciéndolo de esta forma seremos compatibles con Angular Universal y por tanto podremos aprovechar el server rendering y las mejoras de SEO.

Uso de la librería de ngx-translate

En caso de no tener que soportar Server Rendering podemos hacer uso de la librería ngx-translate que nos permite modificar dinámicamente el idioma de la aplicación con una única compilación. A continuación se describen los pasos:

Instalación de las dependencias

Ya sea un proyecto Angular o Ionic lo primero que tenemos que hacer es instalar las siguientes dependencias:

```
1 $> npm install @ngx-translate/core
2 $> npm install @ngx-translate/http-loader
```

Configuración en el módulo principal

En el módulo principal de nuestro proyecto vamos a configurar el uso de esta librería. Para ello, editamos el fichero app.module.ts.

Dentro vamos a crear la siguiente función para inicializar el uso del http loader necesario para la carga de los distintos ficheros de idiomas, donde indicamos la ruta de almacenamiento de dichos ficheros. Lo normal es que se almacenen dentro de la carpeta assets para que estén accesibles vía Http.

```
1 import { TranslateHttpLoader } from '@ngx-translate/http-loader';
2 import { HttpClientModule, HttpClient } from '@angular/common/http';
3 ...
4
5 export function createTranslateLoader(http: HttpClient) {
6   return new TranslateHttpLoader(http, './assets/i18n/', '.json');
7 }
8 ...
```

Dentro de la sección imports del módulo vamos a añadir el nuevo módulo TranslateModule de esta forma, indicando en la receta useFactory la función creada anteriormente e indicando como dependencia el servicio HttpClient de Angular para poder realizar la carga vía Http de los ficheros de idiomas.

```
1 ...
2
3 imports: [
4   BrowserModule,
5   HttpClientModule,
6   TranslateModule.forRoot({
7     loader: {
8       provide: TranslateLoader,
9       useFactory: (createTranslateLoader),
10      deps: [HttpClient]
11    }
12  })),
13 ...
```

Estableciendo el idioma por defecto

Ahora en la carga del componente principal de la aplicación vamos a añadir la lógica para la selección del idioma por defecto y cuál se va a utilizar en función de las propiedades del navegador. En caso del idioma chino añadimos cierta lógica para distinguir el dialecto:

```

1  ...
2
3  constructor(private translate: TranslateService) {
4      this.initTranslate();
5  }
6
7  initTranslate() {
8      this.translate.setDefaultLang('en');
9      const browserLang = this.translate.getBrowserLang();
10
11     if (browserLang) {
12         if (browserLang === 'zh') {
13             const browserCultureLang = this.translate.getBrowserCultureLang();
14
15             if (browserCultureLang.match(/-CN|CHS|Hans/i)) {
16                 this.translate.use('zh-cmn-Hans');
17             } else if (browserCultureLang.match(/-TW|CHT|Hant/i)) {
18                 this.translate.use('zh-cmn-Hant');
19             }
20         } else {
21             this.translate.use(this.translate.getBrowserLang());
22         }
23     } else {
24         this.translate.use('en');
25     }
26
27 }
28
29 ...

```

Creación de los ficheros de idiomas

Ahora dentro de la carpeta `src/assets` creamos la subcarpeta “`i18n`” y dentro vamos a añadir todos los ficheros de idioma que queramos soportar, por ejemplo, el español (es) y el inglés (en) con extensión `.json`.

También podemos hacer uso de parámetros los cuáles podremos establecer con `{{}}`

El contenido de `src/assets/i18n/en.json` podría ser este:

```
1 {  
2   "demo": {  
3     "hello": "hello {{name}}"  
4   }  
5 }
```

Y el de es.json:

```
1 {  
2   "demo": {  
3     "hello": "hola {{name}}"  
4   }  
5 }
```

Es importante que ambos ficheros mantengan las mismas claves.

Uso de las traducciones en los componentes

Para hacer uso de las traducciones en un componente tenemos que hacerlo a través del pipe “translate” que ya nos proporciona la librería de forma que en cualquier template de cualquier componente podemos poner un id existente, el pipe translate y automáticamente se mostrará el valor asociado en función del idioma establecido en el navegador.

```
1 template: `<h4>{{'demo.hello' | translate:{'name': 'Ruben'}}}</h4>`
```

Uso de las traducciones en los servicios/pipes

Para hacer uso de las traducciones fuera de un template lo haremos inyectando por constructor el servicio “TranslateService” que tiene una función “get” para la recuperación del valor, que recibe la key y las interpolaciones asociadas, y devuelve un observable con el valor para el idioma actual. En caso de cambiar el lenguaje, dinámicamente se cambiará este valor gracias a la programación reactiva.

```
1 this.translate.get('demo.hello', {name: 'desde servicio'}).subscribe(  
2   value => this.trans = value  
3 );
```

Integración y despliegue continuo

Los procesos de integración continua nos permiten compilar, pasar los tests y desplegar nuestros proyectos siempre que se produce algún cambio, lo que hace que tengamos un feedback del resultado lo antes posible.

En función de la rama que modifiquemos el resultado puede desplegarse de forma automática en un determinado entorno, incluso en producción.

La idea es que no tengamos miedo a producción y podamos desplegar varias veces al día si fuera necesario.

Integración continua con TravisCI y despliegue en Firebase

Para poder trabajar con TravisCI es imprescindible tener una cuenta en [GitHub](https://github.com/)²⁸

Para poder trabajar con Firebase es imprescindible tener una cuenta de [Google](https://accounts.google.com/SignUp?hl=en)²⁹

Lo siguiente que necesitamos es tener el proyecto que queremos configurar en integración continua.

Para ello utilizando angular-cli simplemente tenemos que ejecutar:

```
1 $> ng new nglabs-travis
```

Podemos probar a arrancarla para ver que efectivamente se ha creado correctamente, ejecutando:

```
1 $> npm run start
```

Verificando en la URL <http://localhost:4200> que la aplicación se muestra.

Y lo que es más importante que nuestro proyecto compila correctamente con los parámetros de producción.

²⁸<https://github.com/>

²⁹<https://accounts.google.com/SignUp?hl=en>

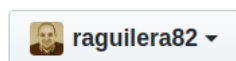
```
1 $> npm run build -- --prod --build-optimizer
```

El siguiente paso es asociar el proyecto con un repositorio de tu cuenta de GitHub. En caso de no tener ninguna, simplemente, nos conectamos con nuestras credenciales y pulsamos en “New Repository” rellenando la información que nos solicita de esta forma:

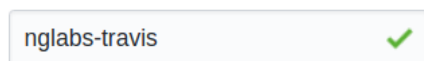
Create a new repository

A repository contains all the files for your project, including the revision history.

Owner



Repository name



Great repository names are short and memorable. Need inspiration? How about **studious-eureka**.

Description (optional)

Repositorio para workshop de integración continua de una aplicación con Angular en TravisCI

☒ **Public**

Anyone can see this repository. You choose who can commit.

☐ **Private**

You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▼

Add a license: **None** ▼



Create repository

Esto nos va a crear el repositorio el cuál vamos a ligar a nuestro proyecto ejecutando en la raíz del proyecto los siguientes comandos:

```
1 $> git init
2 $> git remote add origin git@github.com:raguilera82/nglabs-travis.git
3 $> git commit -m "Initial Commit"
4 $> git push origin master
```

El siguiente paso es configurar TravisCI para ello necesitamos acceder a su [web](https://travis-ci.org/)³⁰ y crear

³⁰<https://travis-ci.org/>

una cuenta a partir de la de GitHub.

Una vez dentro de TravisCI vamos a la opción “New Repository” y veremos todos los repositorios que tenemos públicos en GitHub, habilitamos los que queramos, en nuestro caso “nglabs-travis”



En caso de no ver el repositorio recién creado pulsa en la opción “Sync Account” y vuelve a buscar.


Pinchamos en el engranaje que aparece al lado del nombre y vamos a la pestaña de “Settings” para habilitar la opción “Build only if .travis.yml is present” para que solo se dispare la build cuando se detecte que Travis ya está configurado.

raguilera82 / nglabs-travis  build unknown

Current Branches Build History Pull Requests Settings

General

☒ ON Build only if .travis.yml is present

☐ OFF Limit concurrent jobs 

El siguiente paso es crear el fichero `.travis.yml` en la raíz de nuestro proyecto con el siguiente contenido:

```
1 language: node_js
2
3 node_js:
4   - node # descargará la última versión de node
5
6 before_script:
7   - yarn
8
9 script:
10  - npm run build -- --prod
```

Ahora tenemos que subir este cambio al repositorio, ejecutando:

```
1 $> git add --all
2 $> git commit -m "Añadimos la configuración de TravisCI"
3 $> git push origin master
```

Al hacer esto tenemos que comprobar que TravisCI empieza a trabajar con nuestro proyecto y que da una salida correcta.

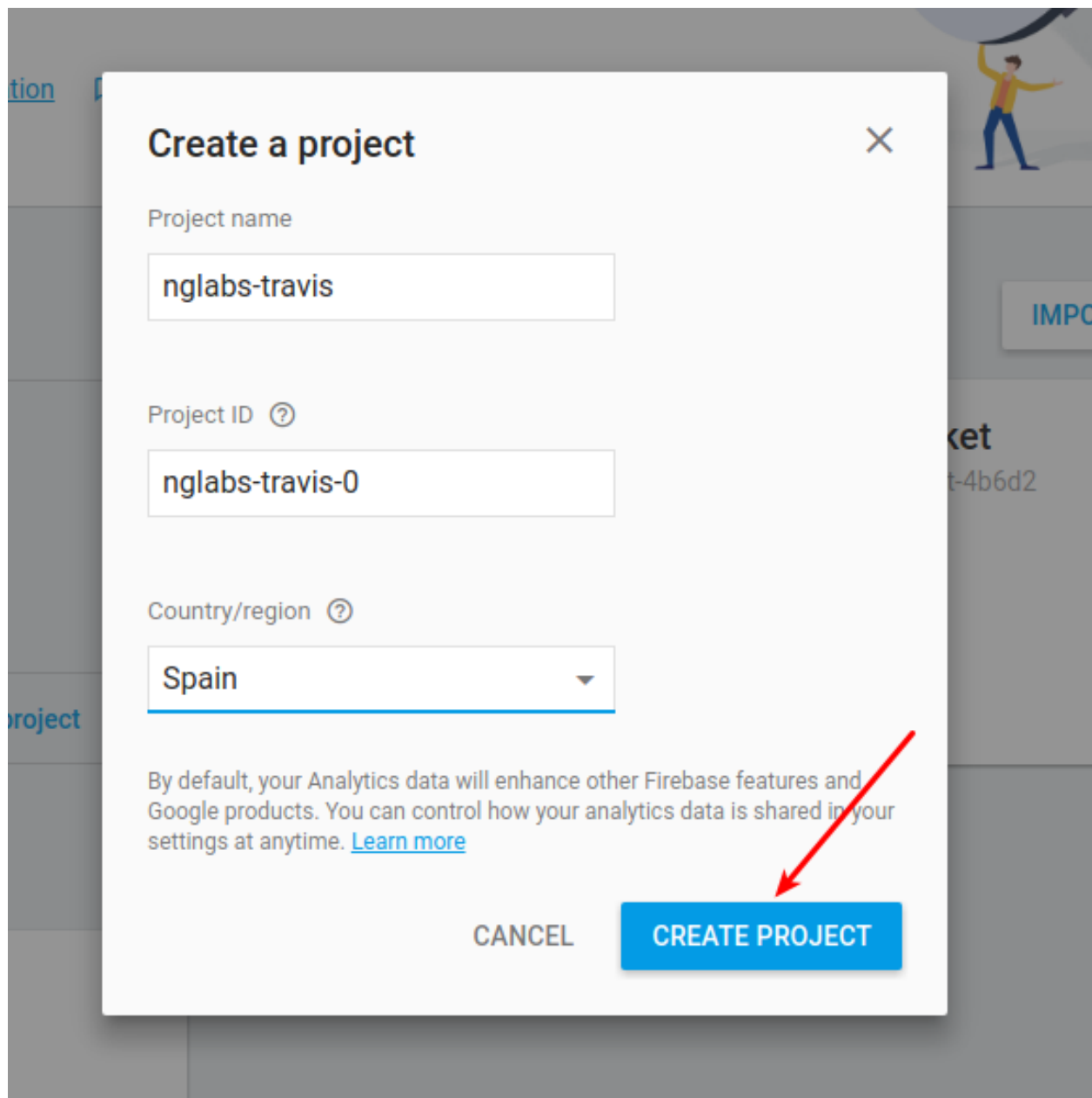
Ya tenemos nuestro proyecto integrado continuamente cada vez que hay un cambio en el repositorio.

Ahora lo normal es querer visualizar esos cambios para ello podemos implementar un despliegue continuo con TravisCI gracias a Firebase.

Para crear un proyecto en Firebase tenemos que conectarnos a la URL <http://console.firebase.google.com> y conectarnos con nuestra cuenta de Google.

En la página de inicio pulsamos en “Add Project” y establecemos un nombre de proyecto y una región geográfica.

³¹<http://console.firebase.google.com>



El siguiente paso es instalar las herramientas de Firebase en nuestra máquina de desarrollo.

Para ello simplemente ejecutamos en el terminal:

```
1 $> npm install -g firebase-tools
```

Una vez instaladas nos tenemos que logar para ello ejecutamos:

```
1 $> firebase login
```

Esto abrirá el navegador y nos pedirá las credenciales, una vez puestas correctamente ya estaremos logados.

Lo siguiente es inicializar firebase, para ello en la raíz del proyecto ejecutamos el comando:

```
1 $> firebase init
```

Desde el terminal nos solicitará que especifiquemos las funcionalidades que queremos configurar, para este caso, simplemente nos vale con seleccionar “Hosting”.

Ahora nos mostrará una lista con los proyectos que tenemos dados de alta en Firebase, seleccionamos “nglabs-travis”.

Ahora nos solicitará que especifiquemos la carpeta que queremos hacer pública, en nuestro caso, pondremos “dist”.

Ahora nos preguntará si nuestra aplicación es una SPA para automáticamente configurar la redirección con index.html, le decimos que sí.

Nos dice que index.html ya existe y si queremos sobrescribir, le decimos que sí.

Estos pasos nos van a crear los ficheros de configuración firebase.json y .firebaserc

Si vemos el contenido de firebase.json tiene que ser exactamente así:

```
1 {
2   "hosting": {
3     "public": "dist",
4     "ignore": [
5       "firebase.json",
6       "**/.*",
7       "**/node_modules/**"
8     ],
9     "rewrites": [
10      {
11        "source": "**",
12        "destination": "/index.html"
13      }
14    ]
15  }
16 }
```

Podemos probar el deploy en Firebase en local ejecutando una build y el comando Firebase:

```
1 $> npm run build -- --prod --build-optimizer
2 $> firebase deploy
```

Esto nos dará la URL donde podemos verificar que efectivamente se ha producido el despliegue.

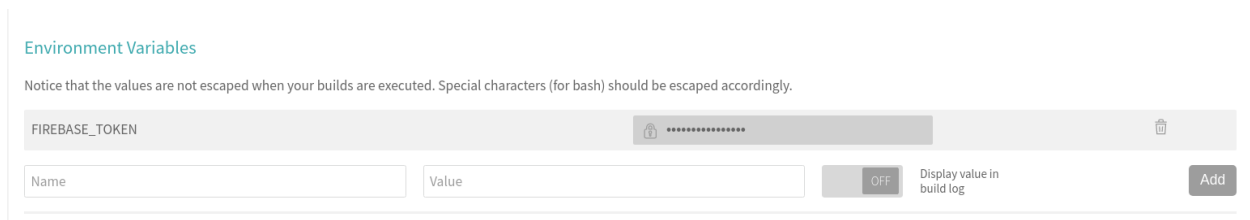
Ahora vamos a integrar esta parte con TravisCI, para ello tenemos que ejecutar el comando:

```
1 $> firebase login:ci
```

Esto abre un navegador para incluir las credenciales y dar permisos. Lo que hace que nos devuelva un token.

Ahora copiamos el token y vamos a la pestaña de “Settings” del proyecto en TravisCI que se encuentra en la opción “More options”.

En la sección de “Environment Variables” añadimos la variable FIREBASE_TOKEN con el valor del token copiado.



Ahora editamos el fichero .travis.yml para añadir la dependencia de firebase-tools y la sección de despliegue, quedando de esta forma:

```
1 language: node_js
2
3 node_js:
4   - node # will use latest node
5
6 before_script: # commands to run before the build step
7   - yarn
8   - npm install -g --silent firebase-tools
9
10 script: # the build step
11   - npm run build -- --prod
12
13 after_success:
14   - firebase deploy --token $FIREBASE_TOKEN --non-interactive
```

El parámetro `--no-interactive` es necesario para que la build se complete con éxito.

Por último, subimos los últimos cambios al repositorio:

```
1 $> git add --all
2 $> git commit -m "Añadimos la configuración de Firebase"
3 $> git push origin master
```

Si todo ha sido correcto, la build en TravisCI terminará satisfactoriamente y todos los cambios que hagamos en el repositorio a partir de ahora los veremos desplegados en Firebase.

En caso de querer ejecutar los tests unitarios, de integración y aceptación este sería el script necesario haciendo uso de xvfb:

```
1 sudo: required
2
3 dist: xenial
4
5 language: node_js
6
7 node_js:
8   - node # descargará la última versión de node
9
10 before_install:
11   - export CHROME_BIN=/usr/bin/google-chrome
12   - export DISPLAY=:99.0
13   - sh -e /etc/init.d/xvfb start
14   - sudo apt-get update
15   - sudo apt-get install -y libappindicator1 fonts-liberation
16   - wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
17   - sudo dpkg -i google-chrome*.deb
18
19 before_script:
20   - yarn
21   - npm install -g --silent firebase-tools
22
23 script:
24   - npm run test -- --single-run=true
25   - npm run e2e
26   - npm run build -- --prod
```

```
27
28 after_success:
29 - firebase deploy --token $FIREBASE_TOKEN --non-interactive
```

En el caso de querer generar un contenedor y subirlo a docker hub o cualquier otro registro donde tengamos permisos, este sería nuestro .travis.yml:

```
1  sudo: required
2
3  dist: xenial
4
5  language: node_js
6
7  node_js:
8  - node # descargará la última versión de node
9
10 before_install:
11 - export CHROME_BIN=/usr/bin/google-chrome
12 - export DISPLAY=:99.0
13 - sh -e /etc/init.d/xvfb start
14 - sudo apt-get update
15 - sudo apt-get install -y libappindicator1 fonts-liberation
16 - wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
17 - sudo dpkg -i google-chrome*.deb
18
19 before_script:
20 - yarn
21 - npm install -g --silent firebase-tools
22
23 script:
24 - PACKAGE_VERSION=$(node -p -e "require('./package.json').version")
25 - npm run test -- --single-run=true
26 - npm run e2e
27 - npm run build -- --prod --build-optimizer
28 - docker build -t $TRAVIS_REPO_SLUG:$PACKAGE_VERSION .
29 - docker login -u $DOCKER_USER -p $DOCKER_PASS $REPO_DOCKER
30 - docker tag $TRAVIS_REPO_SLUG:$PACKAGE_VERSION $REPO_DOCKER/$TRAVIS_REPO_SLUG:$\
31 PACKAGE_VERSION
32 - docker push $REPO_DOCKER/$TRAVIS_REPO_SLUG:$PACKAGE_VERSION
33
34 after_success:
35 - firebase deploy --token $FIREBASE_TOKEN --non-interactive
```

DOCKER_USER, DOCKER_HUB y DOCKER_REPO tienen que establecer su valor en la sección “Settings” del repo específico en TravisCI (donde tengamos FIREBASE_TOKEN).

El Dockerfile para la construcción de la imagen ya lo vimos en la sección Puesta en Producción: Empaquetado con Docker de esta misma guía.

Ionic

Esta tecnología se ha convertido en el estándar para la creación de aplicaciones móviles híbridas; es decir aquellas que gracias a Cordova/Phonegap se instalan como aplicaciones nativas del móvil pero que se ejecutan en un webview, lo que permite desarrollarlas con tecnologías web y se integra a la perfección con Angular.

Además el equipo de Ionic ha creado un ecosistema alrededor de esta tecnología (Ionic PRO) que permite el control de errores, la distribución de actualizaciones sin necesidad de pasar por el market, la previsualización con Ionic View, etc... de las que hablaremos en este módulo.

En este módulo vamos a dar una visión práctica y ver algunos casos de uso que nos podemos encontrar a la hora de abordar este tipo de aplicaciones.

Para una referencia completa recomiendo ver la [documentación oficial](#).³²

Como empezar a crear una aplicación (Ubuntu 16.04)

Para empezar con Ionic tenemos que instalar las siguientes dependencias, la de gradle es obligatoria a partir de la versión 7.0.0 de cordova:

```
1 $> sudo apt-get install lib32z1 lib32ncurses5 libbz2-1.0:i386 lib32stdc++6
2 $> npm install -g ionic cordova
3 $> sudo apt-get install gradle
```

Para crear un proyecto tenemos que ejecutar:

```
1 $> ionic start
```

Este comando nos lanzará una serie de preguntas:

- **What would you like to name your project:** le damos un nombre al proyecto, por ejemplo, ionic-workshop
- **What starter would you like to use:** nos ofrece una serie de plantillas con las que empezar el proyecto.
 - **tabs:** crea un layout dividido en tabs

³²<https://ionicframework.com/>

- **blank:** crea un proyecto en blanco
- **sidemenu:** crea un proyecto con un menu lateral
- **super:** crea un proyecto donde se muestran las buenas prácticas del equipo de ionic
- **conference:** crea un proyecto real de la gestión de información de una conferencia
- **tutorial:** crea un proyecto con un tutorial en base a la documentación de Ionic.
- **aws:** crea un proyecto para el AWS Mobile Hub Starter

Seleccionamos el que mejor se ajuste a las necesidades visuales de nuestra aplicación, por ejemplo, vamos a seleccionar “sidemenu”.

- **Would you like to integrate your new app with Cordova to target native iOS and Android?:** pregunta si queremos integrar la aplicación con Cordova para poder ser instalada de forma nativa en IOS y/o Android. Contestamos que si.

Es importante que nuestro equipo ya esté preparado para el desarrollo con la plataforma que se quiera.

- **Install the free Ionic Pro SDK and connect your app?:** Nos pregunta si queremos integrar la aplicación con los servicios de Ionic Pro. Le decimos que si.
- **How would you like to connect to Ionic Pro?:** Nos pregunta de que forma queremos conectar con Ionic Pro y nos da las siguientes opciones:
 - **Automatically setup new a SSH key pair for Ionic Pro:** si queremos en este momento crear una clave SSH
 - **Use an existing SSH key pair:** si queremos utilizar una clave SSH que ya tengamos generada, por lo general la tendremos generada si trabajamos con Git.
 - **Skip for now:** saltar este paso
 - **Ignore this prompt forever:** o ignorar esto para siempreSeleccionamos la primera opción para que la genere en ese momento. El asistente nos informa de los pasos que va a dar, advirtiéndolo que si ya existe una clave no la va a sobrescribir. Le decimos que proceda. En el proceso nos pedirá un “passphrase” para la clave y al terminar nos pide que confirmemos los pasos. Le decimos que si.
- **Which app would you like to link:** Ahora nos solicita que linquemos el proyecto con una aplicación en nuestra cuenta o creamos una aplicación nueva asociada. Le vamos a decir que cree una nueva aplicación llamada como el proyecto.

Finalizado el asistente nos proporciona información sobre los siguientes pasos que podemos dar y cómo desplegar los cambios haciendo push contra la rama master que gestiona ionic.

```
1 $> git push ionic master
```

Podemos asociarlo con otro repositorio de Git como GitHub o GitLab, creando el repositorio y estableciendo el nuevo origen:

```
1 $> git remote add origin url_repositorio
```

Conectar con los servicios gratuitos de Ionic PRO

Nos conectamos a la URL <https://ionicframework.com/pro/>³³ y pulsamos en “Login” introduciendo nuestras credenciales.

Inicialmente se nos muestra una lista con las aplicaciones que tenemos registradas. Debemos de ver la aplicación anterior “ionic-workshop”, pulsamos sobre ella.

A la derecha vemos una asistente de instalación con los pasos que nos quedan por hacer.

- **Connect your app:** este paso se cumple cuando ejecutamos por primera vez:

```
1 $> git push ionic master
```

- **Deploy to a Channel:** tenemos que asignar una build a un channel. Esto también se cumple haciendo el paso anterior de subida de código, dado que lo estamos subiendo al channel master.
- **Preview your app:** para cumplir con este paso tenemos que instalar en un dispositivo móvil ya sea IOS o Android a través del correspondiente market la aplicación “Ionic View - Test Ionic App”. Una vez instalada nos logamos y veremos en el listado de la aplicaciones la nuestra. Pinchamos sobre ella, podremos visualizar la aplicación en ese dispositivo y sin pasar por el market.
- **Share your app:** si pulsamos en esta opción del asistente se nos muestra una ventana donde se da a elegir entre “Private View App” (parte de pago) y “Public View App” (se mantiene gratuita). Al pinchar en la pestaña “Public View App” nos informa de un id que podemos transmitir a nuestros testers para que lo utilicen en “Ionic View” a fin de ver la aplicación en un determinado canal. Esto es muy útil para enseñar a nuestros clientes el estado de la aplicación en cualquier entorno que hayamos definido. (Nos tenemos que asegurar que el canal a compartir tiene estado público).

³³<https://ionicframework.com/pro/>

- **Deploy Live Updates:** en cada “channel” tenemos el link “Set Up Deploy” al pulsar encima nos aparece una ventana que nos ayuda a configurar el deploy entre tres opciones: descargar las actualizaciones en segundo plano e instalarlas en el siguiente arranque, descarga las actualizaciones durante el splashscreen y lo instala inmediatamente o pregunta al usuario para lo que hace falta meter código en la aplicación.
- **Track Errors in your App:** si pulsamos encima de esta opción, nos lleva a una nueva página donde pulsamos en “Set up Monitoring” que explicamos en un punto siguiente.
- **Package your App:** permite empaquetar el proyecto nativamente en Android y/o IOS sin necesidad de tener el entorno correspondiente montado. (Esta funcionalidad si es de pago).

Establecer monitoring

Para habilitar el monitoring que nos ofrece Ionic Pro tenemos que estar seguros de estar utilizando esta dependencia en nuestro proyecto. Si no es así la instalamos ejecutando:

```
1 $> npm install --save @ionic/pro
```

En el fichero `src/app.module.ts` tenemos que asegurarnos que existe la correcta inicialización de Ionic Pro.

```
1 import { Pro } from '@ionic/pro';  
2  
3 const IonicPro = Pro.init('APP_ID', {  
4   appVersion: "APP_VERSION"  
5 });
```

Donde **APP_ID** es el id que podemos encontrar en el dashboard y **APP_VERSION** se aconseja encarecidamente que mantenga la misma versión que el fichero `package.json` para mantener la trazabilidad.

Captura automática de los errores

Para la captuta automática de los errores en una aplicación con Ionic-Angular, tenemos que añadir el siguiente Handler dentro del fichero `src/app.module.ts`:

```
1 import { Pro } from '@ionic/pro';
2 import { ErrorHandler, Injectable, Injector } from '@angular/core';
3 import { IonicErrorHandler } from 'ionic-angular';
4
5 const IonicPro = Pro.init('APP_ID', {
6   appVersion: "APP_VERSION"
7 });
8
9 @Injectable()
10 export class MyErrorHandler implements ErrorHandler {
11   ionicErrorHandler: IonicErrorHandler;
12
13   constructor(injector: Injector) {
14     try {
15       this.ionicErrorHandler = injector.get(IonicErrorHandler);
16     } catch(e) {
17       // Unable to get the IonicErrorHandler provider, ensure
18       // IonicErrorHandler has been added to the providers list below
19     }
20   }
21
22   handleError(err: any): void {
23     IonicPro.monitoring.handleNewError(err);
24     // Remove this if you want to disable Ionic's auto exception handling
25     // in development mode.
26     this.ionicErrorHandler && this.ionicErrorHandler.handleError(err);
27   }
28 }
```

Y tenemos que añadirlo en la sección del providers del módulo, para sobrescribir el comportamiento de la clase ErrorHandler:

```
1 providers: [
2   // ...,
3   IonicErrorHandler,
4   [{ provide: ErrorHandler, useClass: MyErrorHandler }]
5 ]
```

Captura manual

Desde cualquier parte de la aplicación podemos establecer un log que quedará registrado en la herramienta de monitorización.

Para registrar una excepción:

```
1 Pro.getApp().monitoring.exception(new Error('error'));
```

Para registrar un mensaje de log:

```
1 Pro.getApp().monitoring.log('This happens sometimes', { level: 'error' })
```

Añadir los sourcemaps

Una de las grandes ventajas de la herramienta de monitorización de Ionic Pro es que podemos asignarle los sourcemaps de los ficheros de producción para ver los ficheros donde están los errores de igual forma que en desarrollo.

Para añadirlos de forma automática tenemos que ejecutar:

```
1 $> ionic monitoring syncmaps
```

En caso de no tener una build en producción nos preguntará si queremos hacerla en ese momento.

También se pueden añadir de forma manual en el dashboard web de la aplicación.

Crear nueva página en el sidemenu

Para la creación de una nueva página podemos hacer uso del comando “generate” de Ionic, de esta forma:

```
1 $> ionic generate page nombre_pagina --no-module
```

Esto nos va a crear una carpeta con el nombre de la página dentro del directorio “pages” y tres ficheros: un .scss para el estilo, otro .ts para la lógica y un tercero .html para el contenido.

Para integrar esta página en el sidemenu tenemos que editar el fichero app.module.ts y añadir la declaración tanto en el array de declarations, como en el array de entrycomponents.

```
1  ...
2  @NgModule({
3    declarations: [
4      MyApp,
5      HomePage,
6      ListPage,
7      NombrePaginaPage
8    ],
9    ...
10   entryComponents: [
11     MyApp,
12     HomePage,
13     ListPage,
14     NombrePaginaPage
15   ],
```

El siguiente paso es editar el fichero `src/app/app.component.ts` para añadir una nueva entrada en el array `pages` que mantiene para ser mostrado en el menu lateral.

```
1  this.pages = [
2    { title: 'Home', component: HomePage },
3    { title: 'List', component: ListPage },
4    { title: 'NombrePagina', component: NombrePaginaPage }
5  ];
```

Eventos de navegación

Dentro del ciclo de vida de una página se dispararán estos eventos:

- **ionViewLoad:** se dispara solo cuando la vista es almacenada en memoria. Se puede utilizar para registrar tareas que solo se tienen que hacer una vez.
- **ionViewWillEnter:** se dispara cuando se va a entrar a la página, antes de activarla. Se utiliza para tareas que requieren hacerse todas las veces que se entra en la página (establecer listeners, actualizar tablas, ...)
- **ionViewDidEnter:** se dispara cuando entra en la página y la activa. No hay mucha diferencia con el anterior.
- **ionViewWillLeave:** se dispara cuando se abandona la página antes de la desactivación. Se utiliza para deshacer las tareas que se han registrado como event listeners.
- **ionViewDidLeave:** se dispara cuando se abandona la página y se desactiva. Similar al anterior.

- **ionViewWillUnload:** se dispara cuando la vista se elimina completamente de memoria y de la cache.
- **ionViewCanEnter:** se dispara antes que ninguno de los anteriores cuando se va a entrar a la página, lo que permite realizar el control de acceso pertinente devolviendo true o false.
- **ionViewCanLeave:** se dispara después de todos los anteriores al abandonar la vista, permite determinar una lógica booleana para verificar si el usuario puede o no abandonar la página. Por ejemplo, si no ha dado a guardar los cambios en la pantalla.

Gestión del almacenamiento

Ionic nos ofrece una solución de almacenamiento de información en el dispositivo que es un wrapper que facilita el uso de la librería “localForage”, y nos abstrae del driver de almacenamiento que tenga disponible el dispositivo en el que vayamos a ejecutar la aplicación. Por ejemplo, para los dispositivos móviles será preferible utilizar SQLite y para cuando estemos en el navegador utilizar IndexedDB o WebSQL, esta es la configuración que ya nos ofrece por defecto. Pero como vemos en el ejemplo más abajo podemos especificar ciertos parámetros de configuración, que encontramos en la [documentación de localForage](https://github.com/localForage/localForage#configuration)³⁴

Para empezar a utilizar este servicio de Ionic lo primero que tenemos que hacer es instalar la dependencia:

```
1 $> npm install --save @ionic/storage
```

En caso de querer almacenamiento en el dispositivo móvil tendremos que instalar este plugin de Cordova:

```
1 $> ionic cordova plugin add cordova-sqlite-storage
```

Ahora solo tenemos que editar el fichero `src/app/app.module.ts` e importar el módulo correspondiente.

³⁴<https://github.com/localForage/localForage#configuration>


```
1  import { IonicStorageModule } from '@ionic/storage';
2
3  @NgModule({
4    declarations: [
5      // ...
6    ],
7    imports: [
8      BrowserModule,
9      IonicModule.forRoot(MyApp),
10     IonicStorageModule.forRoot({
11       name: '__mydb',
12       driverOrder: ['sqlite', 'indexeddb', 'websql']
13     })
14   ],
15   bootstrap: [IonicApp],
16   entryComponents: [
17     // ...
18   ],
19   providers: [
20     // ...
21   ]
22 })
23 export class AppModule {}
```

De esta forma en cualquier parte de la aplicación podremos inyectar el servicio Storage que será el encargado de mantener el almacenamiento.

```
1  import { Storage } from '@ionic/storage';
2
3  export class MyComponent {
4
5     constructor(private storage: Storage) { }
6
7     ...
8
9     // establecer clave/valor
10    storage.set('user', {name: 'Ionic'});
11
12    // obtener el valor a partir de su clave
13    storage.get('user').then((val) => {
14      console.log('Tu nombre es: ', val.name);
15    });
16  }
```

El servicio cuenta con los siguientes métodos:

- **get(key):** para recuperar un valor almacenado a través de la key. Hay que tener en cuenta que devuelve una Promesa.
- **set(key, value):** para almacenar un valor en la key indicada. Devuelve una promesa para verificar que ha ido bien.
- **remove(key):** para eliminar un valor del almacenamiento a través de la key asociada. Devuelve una promesa para verificar que ha ido bien.
- **clear():** borra todas las claves del almacenamiento. Devuelve una promesa para verificar que ha ido bien.
- **length():** devuelve una promesa retornando el número de keys almacenadas.
- **keys():** devuelve una promesa retornando todas las keys que están almacenadas.
- **forEach(iteratorCallback):** permite recorrer todas las keys almacenadas devolviendo clave y valor. Devuelve una promesa que resuelve cuando finaliza la iteración.
- **driver:** devuelve un string con el nombre del driver que se está utilizando.
- **ready():** devuelve una promesa que resuelve cuando el almacenamiento está preparado para ser gestionado.

Declaración de múltiples almacenamientos

Para soportar múltiples almacenamientos en la aplicación tenemos que crear un Provider que los gestione y establezca un API de uso.

Este podría ser un ejemplo:

```
1 import { Injectable } from '@angular/core';
2 import { Storage } from '@ionic/storage';
3
4 @Injectable()
5 export class ApiServiceProvider {
6
7   private contacts: Array<Object>;
8   private medias: Array<Object>;
9
10  private contactsDb : any;
11  private mediaDb : any;
12
13  constructor() {
14    console.log('Hello ApiServiceProvider Provider');
15  }
```

```
16     this.contactsDb = new Storage({
17       name: '__my_custom_db',
18       storeName: '_contacts',
19       driverOrder: ['sqlite', 'indexeddb', 'websql', 'localstorage']
20     });
21     this.mediaDb = new Storage({
22       name: '__my_custom_db',
23       storeName: '_media',
24       driverOrder: ['sqlite', 'indexeddb', 'websql', 'localstorage']
25     });
26
27   }
28
29   public getContacts(): Promise<any> {
30     return this.contactsDb.get("contacts");
31   }
32
33   public getMedia(): Promise<any> {
34     return this.mediaDb.get("media");
35   }
36
37   public addDemoContact() {
38     let contact = {
39       firstname: 'Jack',
40       lastname: 'Johnson',
41       emails: [
42         'jj@gmail.com',
43         'jackjohnson@yahoo.cn'
44       ],
45       phones: []
46     };
47
48     this.contacts = [...this.contacts, contact];
49     this.contactsDb.set("contacts", this.contacts).then(
50       () => {
51         console.log("Saved contact to database");
52       }
53     );
54   }
55
56   public addDemoMedia() {
57     let media = {
```

```
58     type: 'video',
59     url: '#'
60   };
61
62   this.medias = [this.medias, media];
63   this.mediaDb.set("media", this.medias).then(
64     () => {
65       console.log("Saved media to database");
66     }
67   );
68
69 }
70
71 }
```

Gestión de las variables de entorno

Para la gestión efectiva de las variables de entorno vamos a modificar la configuración del proyecto para añadir webpack.

Para ello vamos a editar el package.json incluyendo la siguiente sección:

```
1  "config": {
2    "ionic_webpack": "./config/webpack.config.js"
3  }
```

Luego vamos a editar el fichero tsconfig.json y dentro de la sección “compilerOptions” añadimos para que el compilador de TypeScript sepa que tiene una nueva ruta con la información de configuración:

```
1  "baseUrl": "./src",
2  "paths": {
3    "@app/env": [
4      "environments/environment"
5    ]
6  }
```

Ahora creamos el fichero config/webpack.config.js con el siguiente contenido:

```
1  var chalk = require("chalk");
2  var fs = require('fs');
3  var path = require('path');
4  var useDefaultConfig = require('@ionic/app-scripts/config/webpack.config.js');
5
6  var env = process.env.PHASE;
7
8  useDefaultConfig[env] = useDefaultConfig.dev;
9  useDefaultConfig[env].resolve.alias = {
10    "@app/env": path.resolve(environmentPath(env))
11  };
12
13  function environmentPath(env) {
14    var filePath = './src/environments/environment' + (env === 'prod' ? '' : '.' + \
15    env) + '.ts';
16    if (!fs.existsSync(filePath)) {
17      console.log(chalk.red('\n' + filePath + ' does not exist!'));
18    } else {
19      return filePath;
20    }
21  }
22
23  module.exports = function () {
24    return useDefaultConfig;
25  };
```

En resumen estamos estableciendo que vamos a tener una variable de entorno “PHASE” que va a indicar para que entorno queremos cargar la configuración.

Es muy útil modificar la sección scripts del fichero package.json para incluir un script por entorno, si vamos a mantener tres entornos: prod, dev y qa, este podría ser el resultado:

```
1  ...
2  "scripts": {
3    "clean": "ionic-app-scripts clean",
4    "build": "PHASE=dev ionic-app-scripts build",
5    "serve": "PHASE=dev ionic-app-scripts serve",
6    "build:prod": "PHASE=prod ionic-app-scripts build --prod",
7    "serve:prod": "PHASE=prod ionic-app-scripts serve --prod",
8    "build:qa": "PHASE=qa ionic-app-scripts build",
9    "serve:qa": "PHASE=qa ionic-app-scripts serve",
10   "lint": "ionic-app-scripts lint"
```

```
11
12     },
13     ...
```

Ahora solo resta crear la carpeta `src/environments` con los tres ficheros de configuración: `environment.ts` para producción, `environment.dev.ts` para desarrollo (será el de por defecto) y `environment.qa.ts` para el entorno de qa. Es importante que los tres mantengan la misma estructura aunque con diferentes valores. Por ejemplo, para desarrollo sería:

```
export const ENV = { mode: 'Development', api: 'http://dev.dominio.com' }
```

Para poder utilizar esta configuración en cualquier parte de la aplicación solo tenemos que hacer uso de la constante `ENV` importándola de `@app/env` (cuidado con los auto imports que juegan malas pasadas)

```
1  import { ENV } from '@app/env';
2  ...
3  getServerAPI(): string {
4    return ENV.api;
5  }
6  ...
```

Creación y carga dinámica de temas

Una de las prioridades de toda aplicación móvil es que resulte atractiva para el usuario. Por tanto se hace imprescindible que el framework te facilite la manera de crear y cargar los distintos temas en la aplicación, incluso en tiempo de ejecución.

Dentro de una aplicación de Ionic podemos crear todos los temas que necesitemos alojándolos en la carpeta `theme`.

En esta [página³⁵](https://ionicframework.com/docs/theming/overriding-ionic-variables/) de la documentación de Ionic tenemos todas las variables que pueden ser utilizadas en nuestros temas.

Por ejemplo, si queremos crear un tema para darle un toque oscuro a la aplicación, crearíamos el fichero `src/theme/theme.dark.scss` con el siguiente contenido:

³⁵<https://ionicframework.com/docs/theming/overriding-ionic-variables/>

```
1 .dark-theme {
2   ion-content {
3     background-color: #090f2f;
4     color: #fff;
5   }
6
7   .toolbar-title {
8     color: #fff;
9   }
10
11  .header .toolbar-background {
12    border-color: #ff0fff;
13    background-color: #090f2f;
14  }
15 }
```

Y por el contrario si le queremos dar un toque más luminoso tendríamos el siguiente contenido en un fichero “src/theme/theme.light.scss”:

```
1 .light-theme {
2   ion-content {
3     background-color: #fff;
4   }
5   .toolbar-background {
6     background-color: #fff;
7   }
8 }
```

Para que estos temas puedan ser tenidos en cuenta tenemos que importarlos dentro del fichero “src/theme/variables.scss”.

Ahora para aplicar uno u otro tema en nuestra aplicación solo tenemos que editar el fichero “app.html” para añadir dentro del atributo “class” el nombre del tema con un guión. Este sería el ejemplo para aplicar el tema dark:

```
1 <ion-nav [root]="rootPage" #content swipeBackEnabled="false" [class]=" 'dark-them\
2 e' "></ion-nav>
```

Esto esta bien si queremos tener aplicado un único tema. Pero tambien podemos querer cambiar el tema en tiempo de ejecución, para darle la posibilidad al usuario para aplicar uno u otro en función de cierta lógica.

Para ello recurrimos al bus de eventos de Ionic donde en cualquier parte de la aplicación puedo publicar en un topic llamado ‘set:theme’ cual es el tema que quiero aplicar:

```
1 this.events.publish('set:theme', 'light-theme');
```

En el fichero “src/app/app.component.ts” tener la suscripción a dicho evento que modifique el atributo “selectedTheme” de tipo string.

```
1 this.events.subscribe('set:theme', (theme) => {  
2   this.selectedTheme = theme  
3 });
```

De forma que apliquemos este valor al atributo class dentro del fichero “src/app/app.html”.

```
1 <ion-nav [root]="rootPage" #content swipeBackEnabled="false" [class]="selectedTh\  
2 eme"></ion-nav>
```

De esta forma cada vez que se publique el evento con un nuevo nombre de tema, previamente registrado, veremos como se aplica automáticamente en tiempo de ejecución a toda la aplicación.

Implementación de hot deploy

¿Qué es el hot deploy?

Básicamente es hacer una actualización de tu aplicación móvil sin pasar por el market, lo que se traduce en mucho menos tiempo para poner en producción un cambio, al no tener que volver a pasar por el market de turno.

Y ahora es cuando dices, !Esto no lo validan en el Apple Store ni hoy ni mañana! No es así pero si que hay que tener en cuenta una serie de limitaciones.

- La primera validación para entrar en el market no te la quita nadie.
- Si en tu actualización tienes que modificar algún permiso, esta actualización la tienes que hacer a través del market.
- Tampoco vale modificar de forma sustancial el cometido original de la aplicación. Ejemplo claro, si tu has subido al market una calculadora, con una actualización no puedes convertirla en un reproductor de música.
- No puedes mostrar al usuario un mensaje de advertencia de que vas a actualizar la aplicación; lo tienes que hacer en el siguiente arranque de la aplicación.
- La comunicación con el servidor de actualizaciones tiene que ser HTTPS.

Es decir, esto es muy útil en aplicaciones híbridas donde queremos hacer un cambio que implica cambio en el proyecto web asociado (HTML, CSS, Angular) y poder ponerlo en producción en segundos y no en la semanas que pueden tardar los markets en validar la aplicación nuevamente.

Como aplicarlo a los proyectos

En esta sección vamos a explicar los pasos a seguir para permitir hot deploy en nuestras aplicaciones sin hacer uso de Ionic Deploy.

En primer lugar vamos a instalar las dependencias y plugins necesarios.

```
1 $> ionic cordova plugin add cordova-hot-code-push-plugin
2 $> npm install -g cordova-hot-code-push-cli
```

Es necesario crear el fichero “cordova-hcp.json” en la raíz del proyecto con las siguientes propiedades:

```
1 {
2   "name": "nombre-proyecto",
3   "ios_identifier": "",
4   "android_identifier": "",
5   "update": "now",
6   "content_url": "http://server/updates"
7 }
```

Las propiedades son:

- **name:** es el nombre del proyecto
- **ios_identifier:** es el identificador de la aplicación dentro del Apple Store; es necesario para poder redireccionar a la instalación a través de este market.
- **android_identifier:** es el identificado de la aplicación dentro del Google Play Store; es necesario para redireccionar a la instalacion desde este market.
- **update:** indica el tipo de actualización que queremos que se produzca cuando haya un cambio, los posibles valores son:
 - **start:** instala la actualización en el nuevo arranque de la aplicación. Es la de por defecto.
 - **resume:** instala la actualización cuando la aplicación pasa a background.
 - **now:** instala la actualización tan pronto como se termina de descargar.
- **content_url:** es la URL del servidor donde están almacenados todos los ficheros de la aplicación. Es una propiedad obligatoria, puede autogenerarse para desarrollo con el comando `cordova-hcp server`.
- **release:** es cualquier string que indique una versión única. Sirve para determinar si la aplicación tiene una nueva actualización. Es obligatorio pero puede autogene-
rarse con el comando `build` o `server` de `cordova-hcp`.

Solo para facilitar el desarrollo en local tenemos el comando:

```
1 $> cordova-hcp server
```

Que genera un servidor de pruebas con el contenido de la carpeta `www` publicándolo en el `localhost:31284` y, gracias a [ngrok](https://ngrok.com/)³⁶ nos facilita una URL para que nuestra aplicación sea accesible desde Internet por cualquier máquina.

Este comando también genera el fichero necesario `www/chcp.json` que toma todos los valores del fichero `cordova-hcp.json`, estableciendo un string autogenerado para la propiedad `"release"` y sobrescribiendo la propiedad `"content_url"` con la URL de ngrok facilitada.

Otro paso necesario es editar el fichero `config.xml` que tenemos en la raíz del proyecto para establecer la configuración de `chcp` donde vamos a especificar:

- **config-file:** indicando la URL del fichero `chcp.json` donde el plugin que se ejecuta en la aplicación tiene que verificar si existe una nueva actualización.
- **auto-download:** tiene una propiedad `"enabled"` que permite determinar si se quiere hacer o no la descarga automática.
- **auto-install:** tiene una propiedad `"enabled"` que permite determinar si se quiere o no hacer la instalación automática.

Un ejemplo de configuración podría ser:

```
1 ...
2 <name>ionic-workshop</name>
3 <chcp>
4   <config-file url="https://6dfde99e.ngrok.io/chcp.json" />
5   <auto-download enabled="true" />
6   <auto-install enabled="true" />
7 </chcp>
8 <description>Ionic Workshop</description>
9 <author email="raguilera@autentia.com" href="http://www.autentia.com">Autentia T\
10 eam</author>
11 <content src="index.html" />
12 ...
```

Ahora podemos trabajar de la forma habitual y veremos que cada modificación se actualiza en los dispositivos que tengamos configurados. Es importante mantener el servidor de pruebas activo, y saber que si lo reiniciamos nos cambiará la URL de ngrok por lo que tendremos que actualizar el fichero `config.xml` con la nueva URL.

³⁶<https://ngrok.com/>

Configuración para distintos entornos de ejecución

El servidor de pruebas está bien para facilitar el desarrollo pero no es viable tenerlo activo permanentemente para las actualizaciones de las aplicaciones en producción.

Además lo normal es que tengamos distintos entornos de ejecución (dev, prod, qa, uat, ...) por lo que necesitaremos un servidor por cada entorno de ejecución que tengamos.

Para facilitar el uso de distintos entornos podemos crear el fichero “chcpbuild.options” en la raíz del proyecto y establecer las distintas opciones que configuramos en el fichero config.xml especificando el entorno, por ejemplo:

```
1 {  
2   "dev": {  
3     "config-file": "https://dev.server/chcp.json"  
4   },  
5   "prod": {  
6     "config-file": "https://prod.server/chcp.json"  
7   },  
8   "qa": {  
9     "config-file": "https://qa.server/chcp.json"  
10  }  
11 }
```

De forma que podamos compilar para uno u otro entorno especificándolo de este modo para desarrollo:

```
1 $> cordova build -- chcp-dev
```

O para QA:

```
1 $> cordova build -- chcp-qa
```

Previamente debemos subir la versión adecuada del proyecto al servidor del entorno de ejecución especificado.

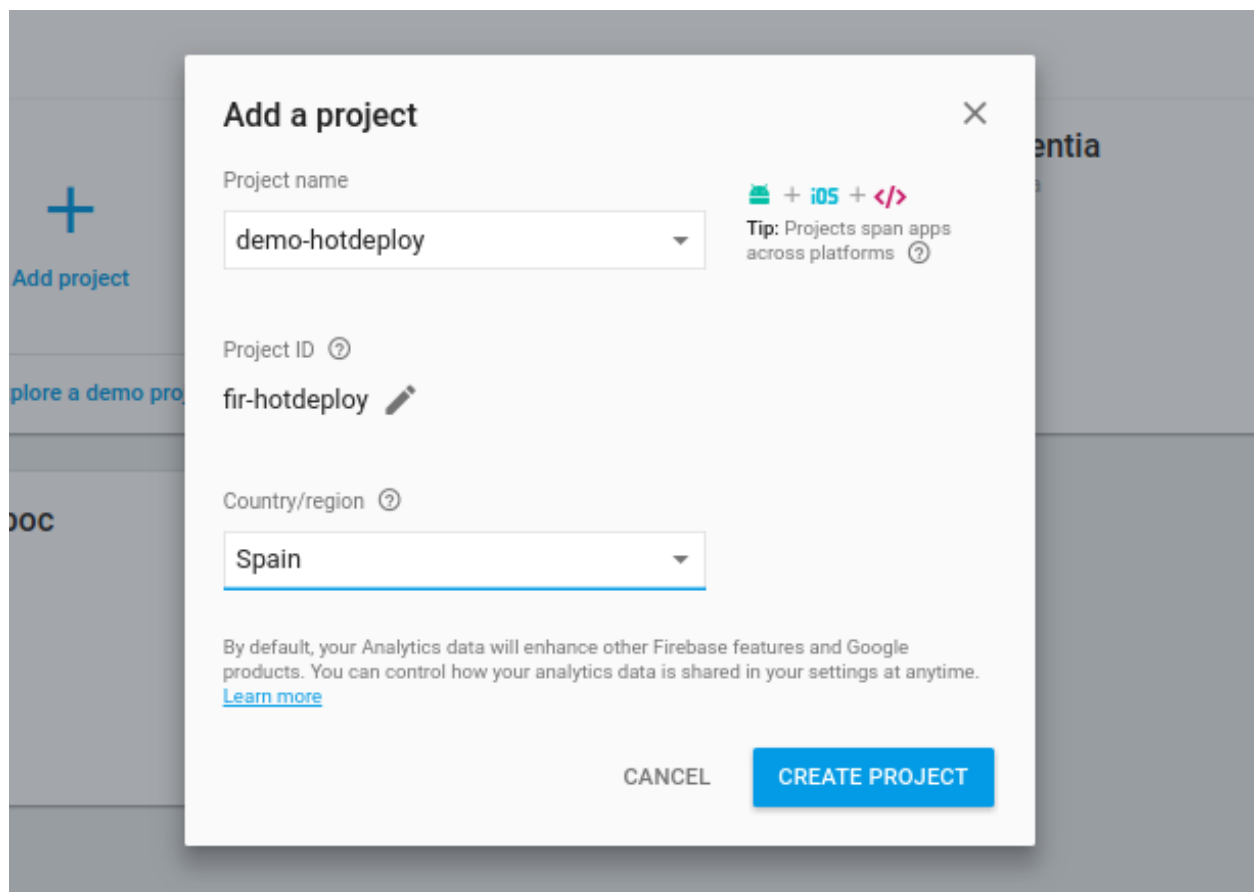
Este comando no modifica directamente el fichero config.xml, sino que establece el valor en el fichero final de la plataforma específica, así por ejemplo, en Android podemos ver el cambio de URL en el path nombre-proyecto/platforms/android/res/xml/config.xml

Usando Firebase como servidor de producción

Como dejar levantado el servidor en nuestra máquina de desarrollo no es viable, vamos a mostrar una solución haciendo uso de Firebase para dejar en “producción” el servidor donde registrar los cambios y al que los dispositivos, que tengan la aplicación instalada, van a consultar para actualizarse.

Lo primero es crear un proyecto dentro de Firebase, para lo que nos conectamos a <https://console.firebase.google.com>³⁷ con nuestro usuario de Google.

Pulsamos en “Add project” y le damos un nombre y una ubicación al proyecto.



Pulsamos en “Create Project” para finalizar el proceso.

Ahora vamos a instalar las dependencias necesarias de Firebase, esto es:

```
1 $> npm install -g firebase-tools
```

Lo siguiente es logarnos con nuestra cuenta de Google desde el terminal, para ello:

³⁷<https://console.firebase.google.com>

```
1 $> firebase login
```

Esto abrirá un navegador donde incluir las credenciales de nuestra cuenta de Google.

El siguiente paso es inicializar nuestro proyecto para utilizar Firebase, esto es:

```
1 $> firebase init
```

Este proceso nos preguntará una serie de aspectos relativos a la configuración. El primero es relativo a los servicios de Firebase que queremos incluir en nuestra aplicación, donde únicamente seleccionamos el de “Hosting”.

Inmediatamente nos devolverá la lista de proyectos que están registrados en Firebase con nuestra cuenta, aquí vamos a seleccionar el proyecto creado anteriormente (demo-hotdeploy (fir-hotdeploy))

La siguiente pregunta es relativa al directorio público de nuestra aplicación que queremos utilizar para publicar, al tratarse de un proyecto Ionic los ficheros de producción a publicar se encuentran en el directorio “www”.

Lo siguiente que nos pregunta es si queremos configurar la aplicación como una SPA de forma que siempre redirecciona hacia index.html, esto realmente no es muy importante en aplicaciones de Ionic, ya que utiliza el hash para las URLs, es importante decir que sí cuando la tecnología (como las aplicaciones de Angular) utilizan urls de tipo HTML5. Contestamos que no.

Nos informa de que va a sobrescribir el fichero www/index.html, es importante decirle que NO lo haga.

Y la configuración ya está completada, podemos ver el resultado en el fichero “firebase.json” de la raíz del proyecto.

Esto ya nos permite publicar nuestra aplicación en Firebase, para ello vamos a generar una build para producción:

```
1 $> npm run ionic:build -- --prod
```

Y, seguidamente, publicamos el contenido de “www” con el comando:

```
1 $> firebase deploy
```

El resultado de este comando nos informa de la URL que podemos utilizar para acceder a nuestra aplicación dentro de Firebase. En mi caso, <https://fir-hotdeploy.firebaseio.com>³⁸

³⁸<https://fir-hotdeploy.firebaseio.com>

Esta es la URL que vamos a establecer en el fichero “chcpbuild.options” para producción a fin de que al compilar la aplicación con los comandos de producción el fichero config.xml tome este valor.

También hay que modificar el valor de la propiedad content_url del fichero “cordova-hcp.json” con el valor de esta URL y ejecutar:

```
1 $> cordova-hcp build
```

Con este comando se actualiza apropiadamente el fichero “www/chcp.json”.

Probando el despliegue en caliente

Para probar el despliegue en caliente vamos a generar una build para producción del código:

```
1 $> npm run ionic:build --prod
```

Vamos a actualizar el fichero chcp.manifest necesario para que el plugin sepa los cambios que se tienen que aplicar:

```
1 $> cordova-hcp build
```

Subimos esta versión a Firebase:

```
1 $> firebase deploy
```

Vamos a instalar una build para producción en nuestro dispositivo Android, estableciendo el parámetro de compilación de producción (si tenemos configurados varios entornos de ejecución):

```
1 $> cordova run android -- chcp-prod
```

Y verificamos que la aplicación se ejecuta normalmente.

Ahora hacemos un cambio en el código y volvemos a ejecutar:

```
1 $> cordova-hcp build && cordova-hcp build && firebase deploy
```

Dependiendo del valor de la propiedad “update” veremos que la aplicación tiene el cambio en el dispositivo móvil, pasados unos segundos (now), al reiniciar la aplicación (start) o al poner en background (resume)

NX: cómo abordar proyectos complejos

Hasta el momento nos hemos centrado en conocer la mayoría de aspectos que nos ofrece el framework de una forma muy básica y con ejemplos sencillos y didácticos. Pero en el mundo real las aplicaciones que hacemos suelen ser mucho más complejas y requieren que aportemos organización y estructura para que sean fácilmente mantenibles.

El CLI de Angular ya nos ofrece un esqueleto de aplicación con todo configurado pero se queda en lo básico, es el punto de partida.

Una forma de estructurar adecuadamente una aplicación con Angular es dividir la aplicación en distintos módulos por funcionalidad que sean fácilmente “enganchables” con el módulo principal de forma lazy, para no aumentar el tamaño general del bundle.

Cuando hacemos esta división nos podemos dar cuenta que existen elementos que se repiten en los distintos módulos. Es algo muy común establecer un módulo “shared” que almacene todos estos elementos transversales como: pipes, directivas, componentes, servicios, modelo, validadores y guardas del router.

Además podemos darnos cuenta de que algunos de estos elementos nos conviene extraerlos a una librería para que puedan ser importados en otros proyectos de Angular.

Esto requiere de mucha disciplina por parte de los desarrolladores, y sobre todo, de los arquitectos y es fácil encontrarse con la excusa de que estas refactorizaciones llevan un tiempo que muchas veces no se tiene.

NX al rescate

Es por ello que la empresa Nrwl formado por ex-miembros del equipo de Angular y con Victor Savkin, el responsable de no tener Angular 3, a la cabeza; han creado la tecnología Nx que se integra con el CLI de Angular para añadir nuevos comandos (gracias a schematics) que facilitan la organización y el desarrollo de proyectos complejos en un único espacio de trabajo.

Una de las principales características de esta tecnología es que facilita el uso del mono-repo, es decir, que todas nuestras aplicaciones y librerías relacionadas con Angular estén en un único repositorio de Git, al estilo de Google, Facebook, o Uber. Las principales ventajas que encontramos son:

- No enmascara ninguna pieza de código, todo lo que hay se ve en el mismo repositorio, así es más fácil darse cuenta de qué piezas ya están implementadas evitando las re-implementaciones.
- Permite ejecutar los tests en todo el repositorio, lo que hace que fallen si hay cualquier cambio en alguna librería compartida que afecta a otra, evitando el exceso de código por programación defensiva.
- Crear una nueva pieza (aplicación/librería) solo supone crear una nueva carpeta en el proyecto, lo que favorece que los desarrolladores queramos modularizar más nuestro código, al no tener que esperar al típico trámite burocrático de crear un nuevo repositorio en la organización y hacer la configuración inicial, lo que puede llevar días, si estamos en el mismo repo, lo podemos hacer en minutos.
- Las refactorizaciones de código que implican cambios en aplicaciones y librerías también se hacen de forma más sencilla y efectiva al no tener que cambiar el contexto y poder ejecutar los tests de forma conjunta.
- Solo tenemos una única versión para todos los proyectos por lo que las migraciones a versiones superiores del framework se hacen de manera más eficiente; como Angular saca versión cada 6 meses y deja compatibilidad con la versión anterior, tenemos un año para modificar nuestro código si es que existe algún breaking change.

Empezando con NX

Toda la información se encuentra en su [documentación oficial](#)³⁹

Lo primero que necesitamos es instalar de forma global la siguiente dependencia que nos va a permitir poder ejecutar los comandos de nx desde el CLI de Angular:

```
1 $> npm install -g @nrwl/schematics
```

Una vez termine el proceso de instalación tendremos a nuestra disposición el siguiente comando para la creación del workspace:

```
1 $> create-nx-workspace nombre-workspace [--directory=nombre-directory] [--npm-sc\
2 ope=nombre-scope-sin-@]
```

Este comando tiene dos parámetros que son opcionales:

- **directory**: para definir un nombre de carpeta distinto al del nombre de workspace.

³⁹<https://nrwl.io/nx>

- **npmScope:** para definir un scope de npm distinto al nombre del workspace, del tipo @nombreScope, es importante definirlo sin la @, ya que al tratarse de un scope de npm ya se lo pondrá internamente.

Dentro de este espacio de trabajo recién creado se encuentran dos directorios: uno “apps” que almacena el módulo principal de cada una de las aplicaciones de Angular y otro “libs” que almacena los módulos secundarios y librerías compartidas.

Creación de una nueva aplicación

Para crear una nueva aplicación hacemos uso del comando generate de Angular pasándole el nombre de la aplicación.

```
1 $> npm run ng -- generate app nombre-app
```

Este comando genera la aplicación con el nombre especificado dentro de la carpeta “apps” y modifica automáticamente el fichero .angular-cli.json para registrar la nueva aplicación.

En caso de querer añadir automáticamente el módulo de routing tendremos que añadir el siguiente flag:

```
1 $> npm run ng -- generate app nombre-app --routing
```

Si entramos dentro del directorio podemos ver que la estructura de esta aplicación no difiere en nada de un proyecto creado con Angular CLI.

Estas aplicaciones serán las encargadas de generar el esqueleto visual de la aplicación, con el módulo principal, la gestión del router para la carga lazy del resto de módulos y el aspecto estético de la aplicación.

Es importante para el rendimiento general que esta aplicación solo contenga lo imprescindible; el grueso de nuestro código tiene que recaer en las librerías.

Creación de una librería como módulo secundario

Para la creación de una librería como módulo secundario tenemos que hacer uso del siguiente comando:

```
1 $> npm run ng -- generate lib nombre-module-lib
```

o si queremos añadir la gestión del router:

```
1 $> npm run ng -- generate lib nombre-module-lib --routing
```

y si queremos que se configure asociado al router de una aplicación y se cargue de forma lazy, tenemos que ejecutarlo con el flag “lazy” e indicando la ruta del módulo principal asociado:

```
1 $> npm run ng -- generate lib nombre-module-lib --routing --lazy --parentModule=\
2 apps/nombre-app/src/nombre-app.module.ts
```

Este tipo de librerías van a contener la lógica de las distintas funcionalidades sin preocuparse por el aspecto estético.

Creación de una librería

Para la creación de una librería hacemos uso del flag `--nomodule`:

```
1 $> npm run ng -- generate lib nombre-lib --nomodule
```

Este comando crea una simple librería sin módulo asociado, la cual se va a encargar de implementar todas aquellas funcionalidades transversales al resto de módulos: como el sistema de notificación de mensajes, el logging, acceso a datos externos, etc... serán susceptibles de extraerse como librerías independientes para su uso en otros proyectos con Angular.

Arranque/distribución de la aplicación

Para poder arrancar o distribuir la aplicación tenemos que especificar con el flag “app” el nombre de la aplicación:

```
1 $> npm run start -- --app=nombre-app
2 $> npm run build -- --app=nombre-app
```

Publicación de una librería

En caso de querer publicar una librería en un repositorio de NPM podemos hacer uso del `ng-packagr` igual que vimos en el módulo de creación de una librería.

Añadimos la dependencia a nuestro workspace:

```
1 $> npm install --save-dev ng-packagr
```

Creamos un fichero `package.json` dentro del directorio de la librería que queremos publicar con el siguiente contenido:

```
1 {
2   "$schema": "../../node_modules/ng-packagr/package.schema.json",
3   "name": "nombre-lib",
4   "version": "1.0.0",
5   "ngPackage": {
6     "lib": {
7       "entryFile": "index.ts"
8     },
9     "dest": "distributions/nombre-lib"
10  }
11 }
```

Tenemos que añadir todas las dependencias necesarias dentro de la sección “peerDependencies”

Ahora ejecutamos el empaquetador con el comando:

```
1 $> ./node_modules/.bin/ng-packagr -p libs/nombre-lib/package.json
```

Este comando genera la carpeta “distributions/nombre-lib” en el raíz del workspace donde almacena los binarios y el `package.json` adecuado para la publicación de la librería. Ahora solo tenemos que situarnos en este directorio y ejecutar:

```
1 $> npm publish
```

En caso de tener los permisos adecuados veremos que la librería se publica correctamente en el repositorio npm configurado.

Cosas a tener en cuenta

En este punto donde ya tenemos varios proyectos creados en nuestro workspace, tenemos que tener en cuenta algunos aspectos:

- Todos los ficheros se referencian con el nombre del workspace.

- Es importante que a medida que vayamos incluyendo elementos en los proyectos nos acordemos de actualizar el `index.ts` asociado, a fin de que las importaciones automáticas se realicen de forma correcta.
- Cuando ejecutamos el comando “`npm run test`” lo estamos haciendo de todos los tests del workspace, no solo del proyecto en el que estemos trabajando, si queremos ejecutar solo los de nuestro proyecto, tendremos que utilizar `fdescribe` o bien cambiar la ruta del fichero `test.ts` especificando solo la ruta a nuestro proyecto.

Actualizar el workspace a la última versión de Angular

Como ya hemos mencionado una de las grandes ventajas de trabajar con un único repositorio es que las actualizaciones se pueden efectuar de forma más rápida, pero si además Nx lo hace por nosotros pues mejor que mejor. Para ello tenemos que seguir los siguientes pasos:

Instalar la última versión de estas dependencias de forma local al proyecto:

```
1 $> npm install --save-dev @nrwl/nx@latest @nrwl/schematics@latest --save-exact
```

Luego ejecutar el comando de npm habilitado por el workspace:

```
1 $> npm run update
```

En caso de no tenerlo (por tener una versión más antigua de nx) debemos ejecutar:

```
1 $> ./node_modules/.bin/nx update
```

Esto va a recorrer todo nuestro workspace realizando todas las actualizaciones necesarias de forma automática. Incluso nos actualiza el fichero `package.json` con las tareas de npm que nos falten.

Terminado el proceso instalamos todas las dependencias por si hubiera necesitado introducir alguna nueva:

```
1 $> npm install
```

Visualizar en un gráfico las dependencias entre aplicaciones y librerías

Otra de las cosas que nos ofrece Nx es una manera muy visual de ver las dependencias que tenemos dentro de nuestro workspace.

Para probarlo puedes hacer clone de mi proyecto:

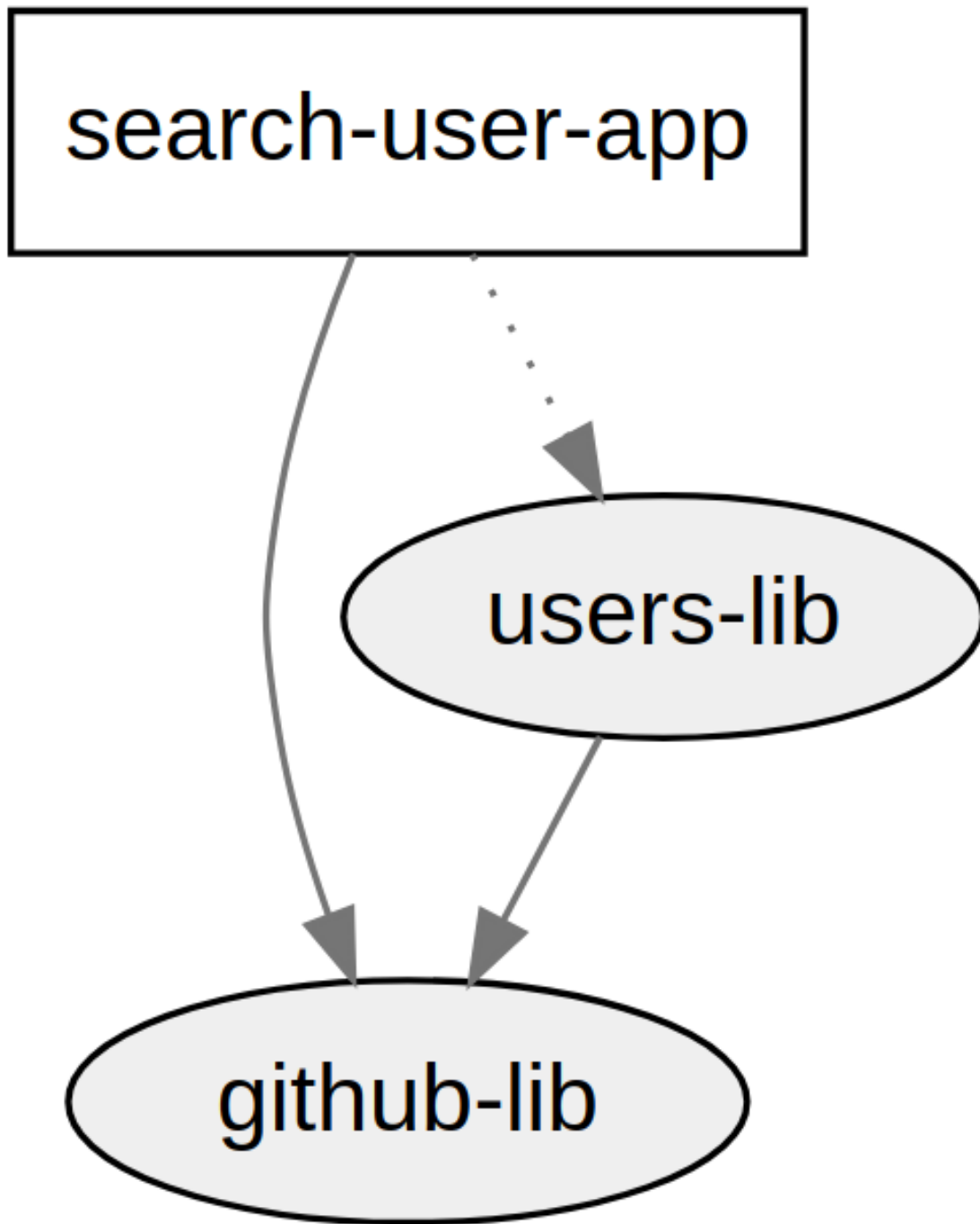
```
1 $> git clone https://github.com/raguilera82/nx-radh.git
```

No olvides hacer `npm install` para instalar todas las dependencias necesarias.

Para obtener el gráfico simplemente tenemos que ejecutar el script de npm asociado:

```
1 $> npm run dep-graph
```

Y a día de hoy podemos ver algo parecido a esto:



Los puntos suspensivos indican que la librería se está cargando en modo lazy.

Integración de Ionic en un workspace de NX

Hasta ahora hemos visto el soporte de NX para manejar aplicaciones de Angular y librerías. Seguro que en este punto te estarás preguntando cómo poder reutilizar esas librerías en proyectos con Ionic sin perder las facilidades y servicios que te da Ionic, y la respuesta inicialmente es que NX no tiene soporte para Ionic.

Pero no te preocupes porque tienes dos opciones: la primera sería publicar las librerías como paquetes npm en el repositorio corporativo y tener los proyectos de Ionic de forma independiente, esta opción requiere más DevOps pero es perfectamente factible.

La segunda es integrar los proyectos de Ionic en el propio workspace de NX, y ahora veremos cómo hacerlo.

Partimos de que ya tenemos creado un workspace con NX con una aplicación y una librería; ésta última queremos compartirla con nuestra aplicación Ionic.

Lo primero que vamos a hacer es crear un proyecto de Ionic de la forma habitual dentro de la carpeta “apps”.

Nota: si en el proceso de creación de la aplicación seleccionamos la opción de integrar el proyecto con Ionic PRO es mejor eliminar la carpeta .git que te genera a fin de que no entre en conflicto con nuestro workspace. Esto hará que no podamos utilizar ciertos servicios de Ionic PRO como la gestión de crashes.

Para poder hacer uso de nuestras librerías en este proyecto y permitir el live reload cuando se modifique alguna librería, tenemos que editar el fichero “apps/app-mobile/-package.json” para añadir donde nuevos ficheros de configuración:

```
1 {  
2   ...  
3   "description": "An Ionic project",  
4   "config": {  
5     "ionic_webpack": "./config/webpack.config.js",  
6     "ionic_watch": "./config/watch.config.js"  
7   },  
8   ...  
9 }
```

Ahora creamos el fichero “apps/app-mobile/config/webpack.config.js” con el siguiente contenido:

```

1  const path = require('path')
2  const TsconfigPathsPlugin = require('tsconfig-paths-webpack-plugin');
3  const resolveConfig = {
4    extensions: ['.ts', '.js', '.json'],
5    modules: [path.resolve('node_modules')],
6    plugins: [
7      new TsconfigPathsPlugin({})
8    ]
9  };
10 // Default config update
11 const webpackConfig = require('../node_modules/@ionic/app-scripts/config/webpack\
12 .config');
13 webpackConfig.dev.resolve = resolveConfig;
14 webpackConfig.prod.resolve = resolveConfig;

```

En este fichero estamos añadiendo al fichero de webpack que viene por defecto con Ionic la posibilidad de cargar el fichero tsconfig.json a través del plugin “tsconfig-paths-webpack-plugin” que tenemos que instalar como dependencia de desarrollo de nuestro proyecto.

```
1 $> npm install --save-dev tsconfig-paths-webpack-plugin
```

Hecho esto el siguiente paso será añadir el path hacia de las librerías en las opciones de compilación del fichero tsconfig.json de nuestro proyecto con Ionic.

```

1  {
2    "compilerOptions": {
3      "allowSyntheticDefaultImports": true,
4      "declaration": false,
5      "emitDecoratorMetadata": true,
6      "experimentalDecorators": true,
7      "lib": [
8        "dom",
9        "es2015"
10     ],
11     "module": "es2015",
12     "moduleResolution": "node",
13     "sourceMap": true,
14     "target": "es5",
15     "baseUrl": ".",
16     "paths": {
17       "@nx-namews/*": [

```



```
18     "../../libs/*"
19   ]
20 }
21 },
22 "include": [
23   "src/**/*.ts"
24 ],
25 "exclude": [
26   "node_modules",
27   "src/**/*.spec.ts",
28   "src/**/*.__tests__/*.ts"
29 ],
30 "compileOnSave": false,
31 "atom": {
32   "rewriteTsconfig": false
33 }
34 }
```

De esta forma ya podemos hacer uso de las librerías de nuestro workspace en nuestro proyecto de Ionic. Ahora creamos el otro fichero “apps/mobile-app/config/watch.config.js” con el siguiente contenido:

```
1  const watchConfig = require('../node_modules/@ionic/app-scripts/config/watch.con\
2  fig');
3  watchConfig.srcFiles.paths = [
4    '{SRC}/**/*.ts|html|s(c|a)ss',
5    '../../libs/**/*.ts|html|s(c|a)ss'
6  ];
```

De esta forma le estamos indicando que también tenga en cuenta los cambios que se produzcan en las librerías a la hora de hacer live-reload de nuestra aplicación de Ionic.

Realizando esta configuración podemos hacer uso de las librerías del workspace de NX en nuestra aplicación de Ionic, siempre y cuando los elementos estén correctamente exportados en el fichero index.ts correspondiente. Además no perdemos la ventajas de trabajar con el CLI de Ionic.

Principios de diseño y buenas prácticas

Hay una serie de principios de diseño en Angular que conviene tener en mente a la hora de ir desarrollando la aplicación, independientemente de la estructura de nuestros proyectos.

Componentes pequeños y tontos

Uno de los más importantes, relacionado con los componentes, es que un componente debe ser siempre lo más pequeño posible y lo más tonto posible. Debemos evitar, en la medida de lo posible, que un componente tenga un template muy grande, debemos pensar si ese template no se puede dividir en componentes más pequeños y más tontos (dummy). Esto va a favorecer la reutilización y el testing de los componentes.

No subestimes el poder de los pipes

Los pipes de Angular es de las cosas más potentes y menos explotadas del framework. Su uso nos permite desacoplar lógica de presentación y poder testearla de forma independiente. Debemos escanear nuestros componentes con la idea de intentar sacar toda la lógica posible de ellos ya que son un elemento difícilmente testeable debido a que está pegado al DOM.

Uso de la herencia en los componentes

Como ya se ha comentado una de las importantes ventajas que presenta las novedades de ECMAScript 6 y que están disponibles en TypeScript es la herencia entre clases.

Uno de los mejores usos que se le puede dar a la herencia en Angular es para resolver la situación de tener varias vistas partiendo de la lógica de un mismo componente.

Si no tuviéramos herencia haríamos un copy & paste del componente modificando el template para mostrarlo de otra forma, pero podemos estar duplicando mucho código.

Lo mejor es hacer que el nuevo componente extienda el componente existente y modificar solo las partes distintas en el nuevo componente.

Por ejemplo, tenemos una situación en la que tenemos un componente que muestra todos los datos de un determinado usuario, con el siguiente código:

```

1  import { Component, OnInit } from '@angular/core';
2  import { Observable } from 'rxjs/Observable';
3
4  import { CurrentUserModelService } from '../current-user-model.service';
5  import { User } from '../list-users/user';
6
7  @Component({
8    selector: 'app-user',
9    template: `<div *ngIf="user$ | async as user">
10      <p>{{user.login}}</p>
11      <p>{{user.url}}</p>
12      <p>{{user.admin}}</p>
13      <img [src]="user.avatar" alt="avatar" width="200">
14    </div>`
15  })
16  export class UserComponent implements OnInit {
17
18    user$: Observable<User>;
19
20    constructor(private serviceModel: CurrentUserModelService) { }
21
22    ngOnInit() {
23      this.user$ = this.serviceModel.user$;
24    }
25
26  }

```

Y ahora queremos un componente solo para mostrar el login y otro solo para mostrar el avatar. En vez de copiar y modificar el template lo que hacemos es extender UserComponent de esta forma para mostrar el login:

```

1  import { Component } from '@angular/core';
2
3  import { UserComponent } from '../user/user.component';
4
5  @Component({
6    selector: 'app-user-login',
7    template: `<h1 *ngIf="user$ | async as user">{{user.login}}</h1>`
8  })
9  export class UserLoginComponent extends UserComponent {
10

```

Y de esta forma para mostrar solo el avatar:

```
1 import { Component } from '@angular/core';
2
3 import { UserComponent } from '../user/user.component';
4
5 @Component({
6   selector: 'app-user-avatar',
7   template: `<img *ngIf="user$ | async as user" [src]="user.avatar" width="200">`
8 })
9 export class UserAvatarComponent extends UserComponent {
10 }
```

No uses funciones para calcular datos dentro de los templates

En algunas ocasiones puedes tener la tentación de hacer uso de una función de cálculo dentro del template de un componente. ¡No lo hagas! Te puedes llevar la “sorpresa” de esa función se llama *n* veces más de las que piensas, ya que cada elemento que se interpola en un template está sometido al change detection de Angular y escapa de nuestro control el número de veces que se produce dentro del ciclo de vida de un componente.

La solución es sencilla, si tienes que hacer un cálculo, hazlo en la lógica del componente y solo interpola en el template un atributo con el resultado del cálculo.

¡No uses funciones de cálculo dentro de los templates!

Los datos siempre deben fluir en una única dirección

Es decir, en la medida de lo posible debemos evitar el uso de la directiva `ngModel` que nos proporciona el two-way databinding pero que complica y hace más pesado el proceso de change detection de Angular, lo que repercute en el rendimiento global de nuestra aplicación.

Es por ello que la comunicación de datos y la gestión del estado debemos mantenerla lo más simple posible. En una aproximación progresiva primero veríamos si no nos vale con la comunicación padre-hijo, en caso contrario pasaríamos a implementar una solución Angular Model Pattern y solo en caso extremo implementar una librería que haga uso de `ngrx`. (Ver tema dedicado a la gestión de estado)

No dejes a tus componentes jugar con todos los juguetes ni saber nada del estado de la aplicación

Este principio se refiere a los “Container Components” donde el constructor inyecta multitud de servicios, lo que lo acopla fuertemente con multitud de capas como la de estado, la de servicios asíncronos, etc... Lo primero que nos tenemos que preguntar es si realmente nuestro componente está cumpliendo el principio de responsabilidad única y, en caso negativo, dividirlo en varios componentes.

En caso de no poder dividirse más aplicaríamos un sandbox (caja de arena) que abstraer al componente del resto de capas de la aplicación, el componente solo se comunica con el sandbox y es el sandbox el que se comunica con el resto de capas. El sandbox implementa el patrón “Facade” que nos proporciona una fachada en forma de API público que permite interactuar con el resto de la aplicación sin conocer los detalles. Lo normal es tener un servicio sandbox por módulo; aunque en algunos casos se puede tener por “Container Component” dentro de un mismo módulo.

Estos principios facilitan en buena medida los tests del componente ya que solo tenemos que hacer un fake de la fachada y/o el sandbox y no de los distintos módulos y servicios de la aplicación.

Utiliza servicios proxy para acceder a los datos de un API sin nada más de lógica

Toda aplicación cliente tiene que comunicarse con fuentes externas de datos, generalmente, a través de API REST. En muchas implementaciones nos encontramos con que estos servicios además de acceder a los datos también los transforman o peor aún cambian el estado de la aplicación, con lo que nos encontramos que hacer tests de estos servicios es muy complicado teniendo que recurrir a clases para mockear las llamadas o directamente no haciendo tests.

En este caso lo mejor es que nuestro servicio Proxy simplemente devuelva un Observable de `HttpResponse`, en caso de ser una llamada GET, y nada más. De forma que podamos crear un test de integración sencillo, y hacer un fake del servicio que poder utilizar por el resto de clases de la cadena que hacen uso de él en los tests eliminando el problema de la asincronía.

Este tipo de clases las podemos nombrar con el sufijo Proxy, y tendrían un contenido similar al siguiente:

```
1 import { ConfigService } from '../config/config.service';
2 import { Observable } from 'rxjs/Observable';
3 import { HttpClient } from '@angular/common/http';
4 import { Injectable } from '@angular/core';
5
6 import { Airport } from '../api/models';
7
8 @Injectable()
9 export class AirportsProxyService {
10
11     constructor(private httpClient: HttpClient,
12                 private configService: ConfigService) { }
13
14     getAirports(): Observable<Airport[]> {
15         return this.httpClient.get<Airport[]>(`${this.configService.config.api}airpo\
16 rt`);
17     }
18
19 }
```

Nota: La entidad Airport se obtiene de la definición del API y, a fin de evitar acoplamientos entre capas, no debe sustituir a la entidad Airport (u otra equivalente) del modelo de negocio de la aplicación.

Pasos para abordar una aplicación del mundo real

En este punto conocemos los conceptos básicos y no tan básicos del framework, las integraciones con otras tecnologías y los principios de diseño, arquitectura y buenas prácticas. Tenemos un montón de herramientas pero la pregunta es ahora, tengo que implementar un proyecto, ¿por dónde empezamos?

Seguramente que el cliente o nuestro departamento de diseño y UX nos haya facilitado alguna especie de boceto o proyecto de cartón piedra funcional con alguna herramienta, y nos haya dicho: ¡allí que te va!

Aquí es cuando nos acordamos de lo simpáticos que son nuestros compañeros o el cliente y que maravilloso sería nuestro trabajo si los diseñadores supieran StencilJS y nos proporcionarían una librería que poder utilizar directamente en nuestras aplicaciones con el diseño pensado.

Como hoy por hoy esto es pura utopía, yo diría que, en la mayor parte de los casos, lo mejor es empezar por crear el “App Shell” de la aplicación, aplicar una plantilla si es que la tenemos e ir creando la estructura de la aplicación.

Empezar a desarrollar las funcionalidades que más valor aportan utilizando un “Container Component” inicialmente sin lógica que haga uso de distintos “Dummy Components”, es decir, componentes que solo trabajan con estructuras de datos proporcionadas a través de `@Input()` por el “Container Component” de forma estática y que si se quieren comunicar con el “Container Component” lo hacen a través de un `EventEmitter` definido con `@Output()`; sin preocuparse por la lógica de negocio de la aplicación solo por la parte visual.

En estos “Dummy Components” la estrategia de “change detection” se define a `OnPush` para aumentar el rendimiento, y es importante que tengamos en cuenta el respetar la inmutabilidad de los datos para que el “change detection” no se vea afectado. A medida vamos necesitando de estructura de datos, vamos almacenando las distintas interfaces en una carpeta “model” que ira formando el dominio de la aplicación.

La ventaja principal de comenzar por este punto es que vamos dando tiempo a la parte de back del proyecto a madurar el API y vamos teniendo claro cuál es nuestro dominio, independientemente del API de turno. Además vamos recibiendo el feedback temprano del usuario de la parte visual que es lo que más llama la atención y en lo que primero se fija, y es más susceptible a recibir cambios.

Otra ventaja de esta aproximación es que delegamos para más adelante los problemas típicos de integración con el API, como las caídas frecuentes, el CORS, los distintos entornos, etc... También empezamos a conocer más la aplicación lo que facilita la posterior modularización para aprovechar el lazy loading y podemos darnos cuenta si las necesidades de gestión del estado nos van a llevar hacia `ngrx` o nos basta con implementar el Angular Model Pattern.

Debemos siempre tener en mente el principio KISS (Keep It Simple and Stupid) y dejar que la arquitectura de nuestra aplicación con Angular vaya emergiendo en base a las necesidades que se vayan descubriendo aplicando otro principio de diseño muy conocido como es YAGNI (You aren't Gonna Need It), es decir, no pretendas hacer la arquitectura y el diseño antes de conocer las historias de usuario.

Teniendo estos principios en mente y habiendo empezado por el “App Shell” de la aplicación ya tendremos una serie de componentes en nuestro módulo principal. Lo primero es asegurarse que todos los elementos del módulo principal son necesarios para la primera carga de la aplicación dado que el tamaño del bundle principal va a determinar el tiempo de carga de nuestra aplicación y la primera sensación de si la aplicación es lenta o va como un tiro.

En caso de detectar elementos que no tengan que estar en el módulo principal vamos a crear módulos secundarios organizados por funcionalidad y respetando la norma de estilo de que la estructura de carpetas sea lo más plana posible.

Estos módulos secundarios serán los que podamos cargar de forma lazy sin repercutir en el tamaño inicial de la aplicación. En caso de detectar elementos que se puedan utilizar en más de un módulo es conveniente crear un nuevo módulo “shared” que los contenga;

estos elementos pueden ser componentes, validadores, pipes que se vayan a usar a lo largo de la aplicación por lo que estarán referenciados en el módulo principal y, por tanto, su tamaño repercutirá en el tiempo de carga inicial de la aplicación así que hay que tener cuidado de que este módulo contenga lo imprescindible.

A la hora de organizar los componentes en los distintos módulos los vamos a clasificar en dos tipos: “Container Components” y “Presentational Components”.

Los “Container Components” van a tener la misión de hablar con el resto de servicios, manejar la asincronía con Observables y hacer la gestión del estado de la aplicación. Aplican el patrón “Sandbox” para no tener más que hacer un fake de una dependencia a la hora de implementar tests unitarios.

Los “Presentational Components” se van a encargar de mostrar los datos en pantalla que van a recibir a través de propiedades @Input y solo en el caso de querer comunicar algo a su “Container Components” lo van a hacer con propiedades @Output de tipo EventEmitter. Estos componentes son susceptibles de utilizar dentro de su template librerías de terceros como PrimeNg, o hacer uso de otras tecnologías de Web Components como Polymer o, lo que se prefiere ahora, StencilJS. Todos ellos deberían estar marcados con la estrategia de change detection OnPush. Este tipo de componentes se testean con tests e2e implementados con Protractor.

Los distintos procesos de negocio se implementan en clases del dominio como servicios evitando lo más que se pueda que se vean afectados por la asincronía. En caso de requerir de acceso al servidor se implementan como un proxy que solo tiene la lógica de uso del servicio HttpClient.